University of Alberta

Queries and Query Processing
in
Object-Oriented Database Systems

by

Dave D. Straube

Technical Report TR90-33
December 1990

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada

University of Alberta

Queries and Query Processing in Object-Oriented Database Systems

by

Dave D. Straube

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta
Spring 1991

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Queries and Query Processing in Object-Oriented Database Systems** submitted by **Dave D. Straube** in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

. . . . . .
M. Tamer Özsu
(Supervisor)

. . . . . .
H.J. Hoover

. . . . . .
L.Y. Yuan

. . . . . .
W. Joerg

. . . . . .
S.L. Osborn

Date: . . . . . .

Abstract

Object-oriented database systems have been proposed as an effective solution for providing the data management facilities of complex applications. Proving this claim and the investigation of related issues such as query processing have been hampered by the absence of a formal object-oriented data and query model.

This thesis presents a model of queries for object-oriented databases and uses it to develop a query processing methodology. Two formal query languages are developed: a declarative object calculus and a procedural object algebra. The query processing methodology assumes that queries are initially specified as object calculus expressions. Algorithms are developed to prove the safety of calculus expressions and to translate them to their object algebra equivalents. Object algebra expressions represent sets of objects which may not all be of the same type. This can cause type violations when the expressions are nested. A set of type inference rules is presented which determines the type consistency of algebra expressions. The next step of the query processing methodology is logical optimization. Algebra expressions are optimized by applying equivalence preserving rewrite rules. Both algebraic and semantic rewrite rules are developed. Applicability conditions for algebraic rules are determined by pattern matching of query subexpressions while semantic rules additionally require that various conditions on the database schema be met. Thus the semantic rewrite rules are unique to a specific application. The final step in query processing is generation of access plans. The interface to an object manager subsystem which performs primitive operations on streams of objects is defined. Join enumeration algorithms from the relational model are adapted and extended to translate algebra expressions into access plans which are sequences of object manager operations.

Acknowledgements

I would like to express my thanks to Dr. Özsu, my supervisor, for his invaluable assistance. He used the right balance between carrot and stick to keep me focused and motivated. Thanks also to the members of the examining committee, Jim Hoover, Li Yan Yuan, Warner Joerg and Sylvia Osborn for their insights and comments.

To the many raquetball, squash, hockey, football and softball teammates and opponents, thank you for being my pressure relief valve. Special thanks go to fellow cyclists Rod Johnson, Franco Carlacci and Dave Hohm. Some of my fondest Edmonton memories are of our rides together.

The Power Plant regulars also had a hand in this effort, although I'm not necessarily sure it was a constructive one ! Nonetheless, thanks for the good company and stimulating discussion.

None of this would have been possible, though, without the help of my wife Liz. Her unwavering support, both financial and emotional, made the dream turn into reality. I am lucky to have such a friend as my partner in life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Object-oriented database systems have been proposed as a solution well suited to providing the data management facilities for applications such as computer aided design, office information and multi-media systems. These applications can be characterized as complex; supporting multiple users concurrently and manipulating large amounts of data. They require that features of programming languages and databases be combined in a coherent, consistent fashion and embody aspects which have traditionally belonged to separate research areas: languages and databases. Object-oriented databases bridge these two worlds by combining the data abstraction and computation model of object-oriented languages with the performance and consistency features of databases. Figure 1.1 illustrates this overlap and categorizes several well known languages and databases.

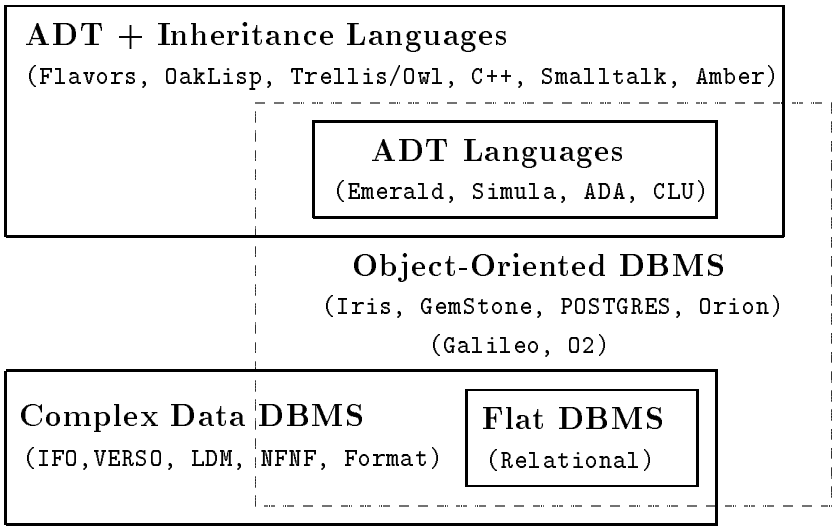


Figure 1.1: Object-Oriented databases bridge traditional DBMSs and languages.

The extensive overlapping of areas in Figure 1.1 shows that many database models have concepts in common. In this respect, each "new" database model is not new

at all, it is merely a new combination and interpretation of pre-existing concepts. A pre-requisite for performing research on any particular model then, is agreement on which set of features the model includes, and their interpretation. Much of the work to date on object-oriented databases has been restricted to these data model issues. The many possible interpretations of features in the object-oriented model (e.g., polymorphism, inheritance, etc.) makes it unlikely that a succinct and formal definition of the model will be agreed upon in the near future. Rather than contribute to the model debate, the goal of this thesis is to formally define an interpretation for a set of object-oriented features and investigate query processing issues within this context. Queries are an important component of database systems as query languages define the user interface (both syntactically and functionally) and the query processing techniques affect performance.

The distinction between defining a data model and investigating issues within the context of that model is important. For example, Codd's specification of the relational model [Cod70, Cod71] provided a solid foundation for the investigation of issues such as dependency theory, null value semantics, query processing and computational completeness precisely because it was concise and formal. Figure 1.2 illustrates this notion of a data model as an interpretation of known concepts (the dashed circle) and as a means of providing the foundation for investigation of related issues. The thesis follows this approach by formally defining an object-oriented database model and using it as the foundation for an investigation of query processing.

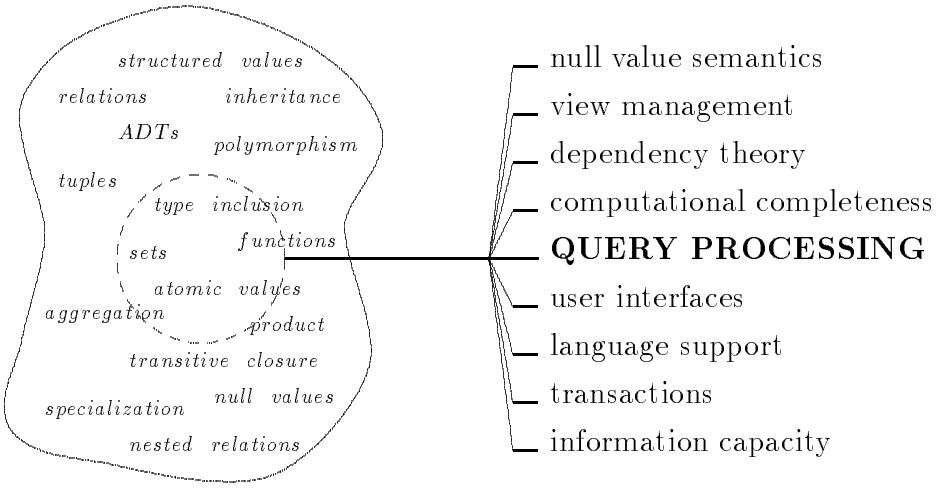


Figure 1.2: The object-oriented model as a collection of known concepts.

1.2 Thesis Scope and Contribution

One of the primary distinctions between an object-oriented programming environment and an object-oriented database is that the database supports efficient associative access, i.e., queries. In particular, a declarative query language is desired because it allows the programmer to focus on *what* the query means as opposed to *how* the

query is implemented. Almost all object-oriented programming languages provide some capability to apply a block of code (a predicate or function) to a collection of objects and return those objects which satisfy the predicate or the results of applying the function to each member of the collection. I maintain that this does not constitute a query facility for several reasons.

Iterating through a collection of objects and applying the code block in its entirety to each successive object is an inefficient paradigm. It requires that each object (and perhaps intermediate values) be evaluated on an individual basis. In contrast, traditional query processing techniques break down the query condition, i.e., the code block, into primitive data manipulation steps and then apply each step to all items being queried at once. Buffering can be used in this scenario to reduce the cost of handling intermediate and final results. In addition, the effort expended in improving the engine which performs the data manipulation primitives is highly leveraged as it improves the performance of all queries.

Object-oriented programming languages permit constructs which do not terminate or which create infinite output. Code blocks which specify such constructs would not return a result while still consuming system resources to evaluate them. A standard user requirement is that the database system return query results in finite time and reject those queries for which this is not possible. Arbitrary code blocks do not have this property.

An object-oriented language compiler or interpreter can use traditional code optimization techniques to improve the run time of a block of code. But such techniques typically do not utilize equivalent primitive data manipulation constructs to rewrite expressions in a more efficient form. In other words, formal representations of query conditions provide more opportunity for rewrites and optimization than the code representation.

The problem this thesis investigates can be stated as follows: What is an appropriate query model for object-oriented systems and how are queries processed in an object-oriented DBMS? The question can not be answered without addressing four more fundamental problems related to object-oriented database systems:

1. What languages can be used to express queries and what defines an acceptable, or computable query?

2. How do the semantics of inheritance affect query processing?

3. Can logically equivalent forms of a query be identified and used as a basis for query optimization?

4. What functionality should an "object management subsystem" provide such that queries can be evaluated efficiently?

The approach proposed in this thesis is depicted in Figure 1.3. Queries are expressed in a declarative language, the object calculus, which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first reduced to a normalized form and then tested for safety. The

normalized calculus expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent instances of classes in the database. The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested functions. This is not as simple as type checking in general programming languages since intermediate results, which are sets of objects, may be composed of heterogeneous types. The next step in query processing is logical optimization achieved by applying equivalence preserving rewrite rules to the type consistent algebra expression. Lastly, an access plan which represents a sequence of primitive data manipulation operations is generated from the optimized object algebra expression.

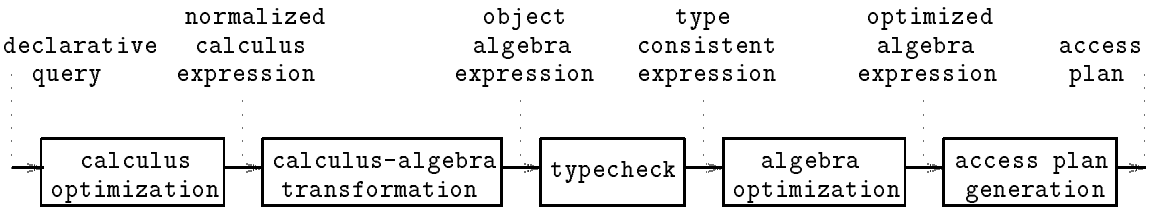


Figure 1.3: Query processing methodology.

The main contributions of this thesis are the following:

1. Formalization of a data model which captures many commonly accepted features of object-oriented database systems and the development of two query languages for that model: a declarative object calculus and a procedural object algebra.

2. A safety criterion for identifying computable queries.

3. A new definition of type consistency for query expressions which insures type consistent bindings of program variables to query results.

4. The specification of an object manager subsystem interface which performs primitive operations on streams of objects and algorithms for generating access plans which are sequences of object manager operations.

1.3 Related Work

This thesis draws from several areas of computer science: advanced database models, object-oriented programming languages, query processing and query languages. The following sections review relevant aspects of these areas and identify the specific ideas which have influenced the work. Several implementations of object-oriented programming systems and databases have influenced the work as well. These are summarized in Appendix A.

1.3.1 Advanced Database Models

It is commonly recognized that the relational data model [Cod70], with its flat representation of data, results in a semantic mismatch between the entities being modeled and the underlying DBMS [Ken79]. A number of approaches have been followed to remedy this. One approach has been to modify the relational model to provide it with more power [Cod79]. More recent work has aimed at including user defined data types within the relational model [OT86, RS87, WSSH88]. Others have allowed for non-normalized relations [OY87, Rot86, RK87, SS86] and developed languages for them [AB84, JS82, Sch85].

An orthogonal approach has been the development of new and more powerful data models generally classified as semantic data models. Most semantic models are based on the concepts of aggregation, generalization and classification [SS77]. Well known examples are the E-R model [Che76], LDM [KV84], the Format model [HY84], Daplex [Shi81], IFO [AH84] and Taxis [MBW80]. Overviews of the entire field can be found in [Hul87, HK87].

The data model developed in this thesis includes the common semantic data model features of aggregation, generalization and classification. The function-based action paradigm of Daplex (without inverse functions) is used as the underlying mechanism for sending a message to an object[1]. The consistent use of algebras in processing queries for the normalized and non-normalized relational models provided motivation to do the same for the data model presented in this thesis.

1.3.2 Object-Oriented Concepts

An overwhelming number of papers have been written in justification of some interpretation of the term "object-oriented". However, although a complete notion of what it means to be object-oriented is not agreed upon, some common themes have emerged. Proposals have been made to define terms such as object, class, identity, behavior, representation, method and inheritance [OOD90, SB85, Weg87]. An object is considered to be an instance of some class which defines its representation and behavior. Identity is that property of an object which distinguishes it from all other objects. It is separate from addressability and value. The object equality operators *identity*, *shallow equal* and *deep equal* defined in [KC86] are present in almost all object-oriented programming languages. Methods define the behavior of objects and when applied to a specific object return another object; either a subobject or some other object in the system.

Representation and/or behavior may be inherited. Inheritance defines a directed graph whose nodes are classes and edges denote inheritance relationships. In the context of databases, this graph represents the database *schema*. Early systems such as Smalltalk [GR85] did not distinguish between inheritance of representation and inheritance of behavior. Thus there was only one inheritance graph. LaLonde and

[1] Chapter 2 clarifies how the message sending paradigm of object-oriented systems is modeled by applying the function associated with a method to an object.

Zdonik investigate the situation where there are distinct graphs for each form of inheritance within the same system [LTP86, Zdo86]. An orthogonal issue is whether a class may inherit from one or many parent classes. Single inheritance forces the inheritance graph to be a strict hierarchy while multiple inheritance organizes the classes into a directed acyclic graph.

Proponents of object orientation claim that the approach is superior to other paradigms for modeling real world entities. But the model may at times be overly flexible while at others overly restrictive. For example, a common error is to use subtyping (subclassing) as a means of composition rather than to define a sum of behaviors. Design rules which guard against misapplication of object-oriented principles during system design are given in [HO87]. The requirement that all concepts be organized into classes can be overly restrictive. For example, Borgida has investigated how non-strict taxonomies of classes can be used to accommodate exceptional entities [Bor88], i.e., those which differ only slightly from the class definition.

An alternative to classification schemes, yet still termed object-oriented, is the notion of delegation. In this paradigm each object defines some behavior itself and delegates additional behavior to other prototypical objects as required [Agh87, Lie86]. Strong arguments have been made to show that delegation and inheritance are both viable methods for incremental definition and sharing of behavior [Ste87]. Recent models have attempted to combine the two in an integrated fashion [Sci89].

Some more formal approaches to inheritance and classification have also helped define object-oriented concepts. Cardelli defines a set inclusion semantics for multiple inheritance by interpreting objects as records with function components [Car84]. Cardelli and Wegner use a typed λ-calculus augmented with quantification to model abstract types, subtypes and type inheritance [CW85]. They also make the distinction between ad hoc, inclusion and parametric polymorphism which correspond to operator overloading, subtyping and generic functions respectively. A survey of typing in various object-oriented programming languages is given in [DT88]. Other formalization attempts have focused on providing a posteriori specifications of Smalltalk-80 using VDM and other specification languages [CP89, Wol87].

This thesis incorporates many of the concepts presented above. We use a multiple inheritance model which does not support any form of delegation. Inheritance without delegation gives the schema a regular structure which can be exploited during query processing. The data model defines a mechanism for behavioral inheritance only and makes no restriction on object representations. This allows objects within a class to have different representations and permits the representation inheritance graph to be distinct from the behavioral inheritance graph if desired. Representation independence also permits treating objects as abstract instances of a class whose methods define a behavior. We support inclusion polymorphism as defined by Cardelli and Wegner [CW85] and base the form of our type inference rules on those proposed by Cardelli in [Car84].

1.3.3 Query Processing

Query processing has many components including query representation, transformation, optimization and evaluation. A good overview of these issues and their interrelationships in the context of the relational model is provided by Jarke [JK84].

Two types of query transformations are common. One generates query expressions for which it is easier to prove certain properties such as safety. Methods for rewriting relational calculus expressions to obtain domain independent formulas are presented in [Bry89, GT87]. Domain independence implies safety by insuring that a query can be answered by examining only the relations it references, not the (possibly infinite) domains their attributes are drawn from [Fag82, GT87, Nic82]. A similar technique using set operators instead of universal quantification and negation is given in [OW89].

The second type of transformation generates query expressions which are less costly to evaluate. These are often referred to as logical transformations. Jarke [JK84] and Talbot [Tal84] identify many such transformations for the relational data model. Shaw identifies a limited number of logical transformations for an object algebra [SZ90].

The ability to precisely define the nature and pre-conditions of logical transformations has led to their application using rule based systems. This technique can be used both to ameliorate queries [GD87, HP88] and to translate them into executable access plans [Fre87, GD87, Loh88].

Generating access plans requires that (1) a well defined set of primitive query operations be implemented, and that (2) techniques exist for enumerating and evaluating alternate sequences of operations. The relational storage subsystem (RSS) of System R [Ast76] defined the standard for relational low level query interfaces. Similar low level interfaces have not been defined for other data models because they often are mapped to the relational model. Enumeration of alternate operation sequences has received significant attention in the context of join ordering [OL88, RR82, SAC+79].

Almost all of the referenced work is applied in this thesis. The proposed query processing methodology, with the exception of the type consistency step, is similar to that proposed by Jarke [JK84]. The notion of domain independence in the relational model is extended to define finite and infinite classes in Chapter 3 and their relationship to the safety of queries. We use logical, equivalence preserving transformations to improve query expressions and define them with conditions which would be suitable for a rule based transformation system. The join enumeration algorithm of [OL88] and join template representation of [RR82] are modified and extended for generating our access plans.

1.3.4 Query Languages

Many query languages have been developed for databases. I make the distinction between "user" and "formal" languages. Languages such as QUEL [Sto76] and SQL [Dat87] are user languages while calculi and algebras are formal. User languages associated with a particular implementation are discussed in Appendix A.

The most well known calculus and algebra are those for the relational model

[Cod70, Cod71, Klu82, OW89, Ull82]. An algebra exists for the ¬1NF model [Rot86, Sch85] and for general non-normalized relations [AB84]. Complex object manipulation languages have also been developed [BK86].

Osborn [Osb88, Osb89] describes an object algebra which views objects as passive data and does not support encapsulation. Her algebra allows aggregate or set objects to be disassembled and recombined and then reused in later stages of a query.

Zdonik and Shaw [SZ89, Zdo88] developed an object algebra which supports encapsulation but only allows unary methods to be used in qualifying algebra operators. They also introduce an *id-equal at depth i* comparison operator which allows structural comparisons of objects. Results of queries are always new objects. Thus the same query run twice may not pass some equality tests which use the structural comparison operators. This complicates using query results as views since the database user must insure that only the proper types of comparisons are performed on the results of a previous query.

The user language OQL [ASL89] specifies queries as a subschema of the database combined with selection criteria on the objects which are part of the subschema. Objects whose components extend past the query defined subschema are 'truncated' in the sense that those components are not available in the query result. This allows definition of views which hide a part of the objects' representation while allowing free access to the remainder of the objects. OQL considers objects strictly as aggregations, not ADTs, and no discussion is presented on how to process and optimize such queries.

This thesis focuses on formal query languages by developing both an object calculus and an object algebra. Our query languages differ from those of Osborn [Osb88], Alashqur [ASL89] and Shaw [SZ89] in that we support strict encapsulation, i.e., operators which manipulate or depend on the object representation are not allowed. Chapter 8 addresses how the addition of new operators affects the overall query processing methodology.

1.4 Organization of this Thesis

This thesis is divided into eight chapters almost along the lines of Figure 1.3. The object-oriented data model that forms the basis of investigation is introduced in Chapter 2. The model formalizes objects, methods, classes, inheritance, legal database states and operations. A sample hypertext database is defined which is used in examples throughout the thesis.

The following two chapters formalize the notion of queries against the model of Chapter 2. Chapter 3 defines a declarative query language: the object calculus. The syntax and semantics of low level query primitives are specified and a safety criterion for object calculus expressions is developed. Chapter 4 presents a procedural query language, the object algebra, which implements a subset of the full object calculus. An algorithm for translating calculus expressions to their algebra equivalent is given.

Traditional query languages, with their limited data types and operations, have required only simple checks for type consistency. In contrast, the data abstraction and multiple inheritance present in the object-oriented model do not permit such trivial

typechecking. Chapter 5 presents a set of type inference rules for insuring the type consistency of object algebra expressions. Type consistency is based on the notion that an expression may conform to several types concurrently due to inheritance. Several algorithms are developed for manipulating the class inheritance graph and determining type inclusion relationships.

Chapter 6 presents a suite of equivalence preserving transformation rules for object algebra expressions. The rules are intended to serve as the rule base for a query rewrite system similar to that of Starburst [HP88]. Three types of transformation rules are presented: identities, conditional rules and semantic rules. Identities are always applicable while conditional rules must meet a set of well defined conditions to be eligible. Semantic rules additionally utilize the semantics of the object-oriented data model, class inheritance graph and type consistency to determine rule eligibility.

The last step in query processing, execution (access) plan generation, is addressed in Chapter 7. Access plan generation is the process of translating object algebra expressions into sequences of executable, primitive data manipulation operations. An object manager interface which provides these primitive data manipulation operations is defined followed by two algorithms for generating access plans for the interface. The first algorithm takes a naive approach and generates sub-optimal plans. The second algorithm generates a family of access plans (which includes the optimal plan) by applying a modified relational join enumeration technique.

Conclusions and contributions of this work are presented in Chapter 8. The results are summarized and future research directions that this work suggests are discussed.

Chapter 2

The Object-Oriented Database Model

2.1 Introduction

The data model attempts to meet several objectives, the most important of which is to provide a query formalism which can be used to investigate properties of queries such as safety, expressiveness and equivalence preserving transformations. Another objective is to accurately reflect common object-oriented concepts without setting any restrictions on implementations or physical object representations. For example, relational query languages do not reflect whether relations are vertically or horizontally partitioned in the physical system. Such knowledge is required for physical data access but is not required in order to completely and unambiguously specify a query. In an analogous fashion, the object-oriented data model and query formalism should have a logical view of data which is consistent for all physical implementations.

The data model should additionally support both a set oriented query formalism as well as a general purpose programming language with assignment. The common problem of 'impedance mismatch' [Ban88a, MSOP86] occurs when a set oriented query language and an imperative programming language with different computational paradigms are used together. Removing this mismatch makes it easier to implement complex applications which operate on persistent data.

These objectives are best achieved by modeling objects as instances of abstract data types (ADTs). This interpretation can be explained in the traditional context of objects which are modeled as instances of record types [Car84]. Consider that the fields of a record may hold data or functions. Restricting the fields of a record type to functions on types results in a true ADT where the components of the record implement the ADT abstraction. Functions in the mathematical sense are not dependent on local data, but implementations of functions using imperative programming languages are. As a result, implementations will often allow both data and functional fields with the data fields corresponding to the object state, and the function fields implementating the object behavior.

Defining objects as combinations of data and functions raises the question of visibility. Is the data component visible only to an object's functions or to any user

of an object? The issue takes on added significance when we speak of *inheritance*. When we say that type A inherits from type B we could mean:

1. **inheritance of representation** – type A objects have the same data components as type B objects (and possibly additional ones);

2. **inheritance of behavioral implementation** – code implementing the functions of type B objects can be applied to the data structures representing type A objects;

3. **inheritance of behavioral specification** – type A objects provide functions with the same names, arguments and semantics as those of type B objects.

Note that inheritance of behavioral specification (3) is a concept while items (1) and (2) are realizations of that concept.

The inheritance relationships among types can be represented by a graph. This graph is highly dependent on the form(s) of inheritance that a system supports. For example, Smalltalk-80 [GR85] enforces (1) and (2) but allows inheriting types (subtypes) to change the arguments and semantics of inherited functions. The *public* and *private* declarations of Owl [SCW85] and C++ [Str86] allow (1) and (2) to be chosen for each data component and function on an individual basis. A system can have multiple inheritance graphs. LaLonde [LTP86] describes a Smalltalk-80 variant which has one inheritance graph for implementations and a second for behavioral specifications. It is also possible to have (3) without either (1) or (2). In this case each type must reimplement the functional specifications and semantics it has inherited.

By separating behavior from implementation, the data model places no restrictions on implementations [DT88, Sny87]. This allows

- a type to have multiple representations and implementations,

- subtypes to have representations and implementations different from their supertypes.

The model can be used consistently for distributed systems with heterogeneous processing elements, systems where functionality is distributed by type, and systems where multiple representations of a type are required.

Modeling all object-oriented concepts is not possible as some are in conflict with each other. For example, this thesis restricts objects to be instances of only one type while other models [Fis87] allow an object to be a member of multiple types concurrently. However, an object may exhibit the behavior of several types via multiple inheritance. Lastly, the model presented here does not address primarily implementation issues such as object migration, exception handling, schema evolution [BKKK87], and saving queries as views [TYI88].

2.2 Values and Objects

All definitions in the model are based on the existence of the following sets:

- A finite set of *basic domains* $D_1, \ldots, D_n$ where $\mathcal{D} = \cup_{i=1}^{n} D_i$.
- A countably infinite set A of symbols called *attributes*.
- A countably infinite set ID of *identifiers*.
- A finite set CN of *class names*.
- A finite set MN of *method names*.

Definition 2.1 *Values:* There are three types of *values*:

1. Every $v \in \mathcal{D}$ is an *atomic value* for which there exists a textual representation.
2. Every finite subset of ID is a *set value*.
3. Every element in $I\!P(A) \times I\!P(\mathcal{D}) \times I\!P(ID)$[1] is a *structural value*.

The symbol $\mathcal{V}$ denotes the set of all values. □

Definition 2.2 *Objects:* An *object* is a triple $o = (id, cn, val)$ where $id \in ID$, $cn \in CN$ and $val \in \mathcal{V}$. $O = ID \times CN \times \mathcal{V}$ is the set of all objects. We will use the notation $o.id$, $o.cn$, and $o.val$, respectively, to denote the identifier, the class and the value of object o. □

Based on an object, the function $ref(o)$ is defined which associates to an object o the set of all identifiers referenced within the value part of the object. This is a recursive function which includes all references of objects nested within o.

Definition 2.3 *Consistent set of objects:* A set of objects Θ is *consistent* iff:

- $\forall o, p \in \Theta, o.id \neq p.id$ (No two objects in Θ have the same identifier – *unique identifier assumption* [KC86].)
- $\forall o \in \Theta, ref(o) \subseteq \bigcup_{p \in \Theta} p.id$ (Each identifier in $ref(o)$ is an object in Θ – *no dangling identifier assumption* [KC86].) □

2.3 Classes and Inheritance

Classes are abstract types whose instances are objects. In our model, a class is a type definition as well as a synonym for all objects which are instances of the type [Ban88a]. A class defines the interface and semantics of its instances by making public a set of methods and their signatures. The method signature specifies the method name, and the number and type of the arguments and result. For example, the class *Person* might define the method

$$salary \quad : \quad Person \times Date \rightarrow Number$$

[1] $I\!P(X)$ denotes the powerset of X.

whose semantics are to provide the salary values over a person's lifetime.

Signatures provide the basis for a strongly typed, compile time-checkable system. Type checking is based upon the notion of *conformity*. A type P *conforms* [BHJL86, BHJ+87] to a type Q if

1. P provides at least the operations of Q;
2. The types of the results of P's operations conform to the types of the results of the corresponding operations of Q;
3. The types of the arguments of Q's operations conform to the types of the arguments of the corresponding operations of P.

The conformity relation is reflexive but not necessarily symmetric or transitive. Assuming a set of primitive types and their known conformity, any non-primitive type can be tested for conformity by recursively examining the types referenced by signatures of its methods until only primitive types remain. An assignment is legal if the type of an object conforms to the declared type of the identifier to which it is being assigned. A formal definition of the conformity relation between types is given in [BHJ+87] and is sufficient for developing a type checking algorithm.

A class which conforms to another and additionally adds behavior is said to be a *specialization* of that class. This new class, termed a *subclass*, 'behaves like' [Zdo86] its parent class in the sense that it exhibits all the behavior of its parent(s) in addition to defining new behaviors. For example, we could define *Student* as a subclass of *Person*. Since a student object implements the behavior of a person object, the *Student* class provides the method

$$salary \quad : \quad Student \times Date \rightarrow Number$$

As well, it could specialize *Person* by adding new behavior defined by the methods

$$\begin{aligned} grade &: Student \times Course \rightarrow Number \\ thesis\_topic &: Student \rightarrow String \end{aligned}$$

The concept of a subclass 'behaving like' its parent is called *behavioral inheritance* and corresponds to the third form of inheritance described on page 11. In our example, *Student* inherits the behavior of *salary* from its parent class. There is no restriction on the number of classes a subclass may inherit behavior from. The case where a class inherits from multiple parents is termed *multiple inheritance*. Our use of the term inheritance will refer specifically to behavioral inheritance in the remainder of the paper.

Since each class may have parents from which it inherits behavior, we can define a *class inheritance lattice* as the directed graph $G = (V, E)$ where vertices in V represent classes and an edge (v_i, v_j) exists in E if v_j is the parent of v_i. We say v_i is a *subclass* of v_j if there exists a path from v_i to v_j in E. Conversely, v_j is a *superclass* of v_i if a path from v_i to v_j exists.

Multiple inheritance mechanisms require a conflict resolution protocol to deal with cases where a class inherits the same named behavior from a number of classes. We

consider the definition of this protocol to be an implementation issue and, therefore, do not specify it explicitly as part of the formal data model. However, either one of the two alternatives that have been proposed in the literature – namely requiring the user to explicitly define the multiple inheritance semantics or resolving the conflict according to a predetermined ordering of the class lattice – is appropriate.

The behavioral inheritance described above insures that a subclass always conforms to all of its superclasses. Others have called this *substitutability* [SZ89]. Substitutability insures that an object of class P can be used in any context specifying a superclass of P. Note that a subclass could redefine the class of the result of an inherited method to be a subclass of the original result class without violating substitutability. This is because rule 3 in the definition of conformity is not violated by such a change. In order that the behavioral inheritance terminates, the class lattice has a *root* class which is the parent of all other classes.

We call the instances of a class c the *extent* of c and denote it as $ext(c)$. It is also convenient to have a notation for referring to all the instances in the extent of all classes rooted at a specific node in the class lattice. For a class c, the *deep extent* of c, denoted $ext^*(c)$, is the union of the extent of each class in the subtree of the class lattice rooted at class c. Figure 2.1 illustrates a sample class lattice with rectangles representing classes and circles denoting objects. Solid lines indicate a subclass relationship and dotted lines indicate class membership. In this example $ext(c_1)$ denotes the objects $\{o_1, o_2\}$ while $ext^*(c_1)$ denotes the objects $\{o_1, \ldots, o_{10}\}$.

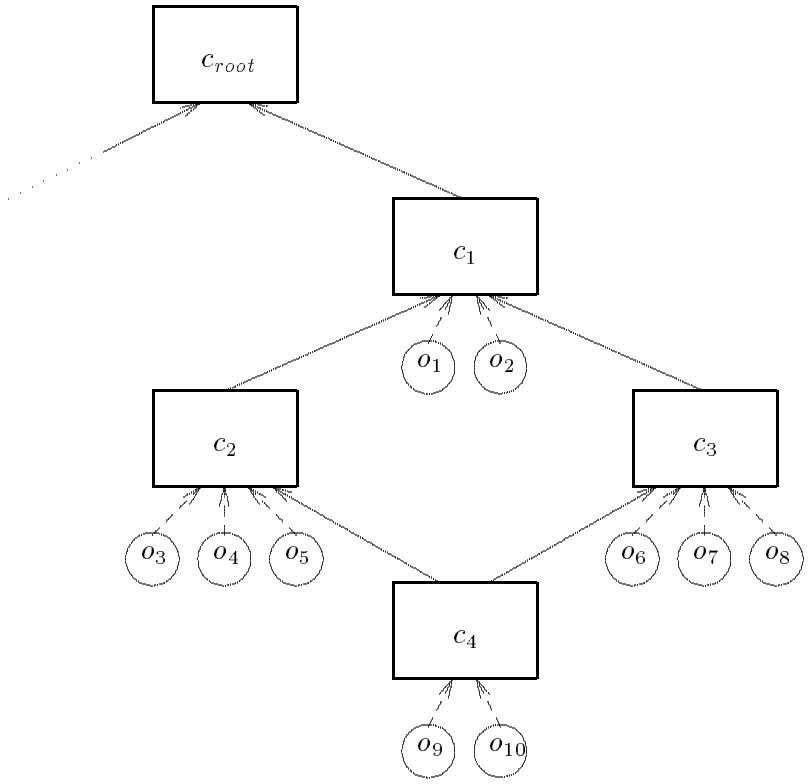


Figure 2.1: Class lattice with extents.

The definition of methods, classes and inheritance are formalized next. As indicated earlier, methods define the behavior of objects. In the object-oriented paradigm, a method is *applied* to an object or a message identifying a method is *sent* to an object and a result object is returned [SB85]. Conceptually, a method can be thought of as providing a mapping from a *source* object to a *target* object. If, in the process, the value of the source object is changed, we say that the method has 'side effects'. Methods with side effects can adversely affect query processing. Consider a method whose side effect is to create new objects in the class being queried. A query using such a method would never terminate since the query set grows as each member of the query set is evaluated. This thesis considers only side effect free methods.

Definition 2.4 *Methods:* A *method* is a triple $m = (nm, f : S \rightarrow T, b)$ where:

- $nm \in MN$ is the name of the method.
- $f : S \rightarrow T$ is a function mapping a product of source domains to a target domain of the form
$$f : S_1 \times S_2 \times \ldots \times S_n \rightarrow T$$
where $S_1, \ldots, S_n$ and T are sets of objects.
- b denotes the semantics (behavior) of function f. □

The following functions will be useful in the upcoming discussion:

- $source(m)$ denotes S_1, the first source domain in S of method m.
- $target(m)$ denotes the target domain, T, of method m.
- $name(m)$ denotes the name mn of method m.

Function f is n-ary in order to allow parameterized methods. For example, $f(s_1, \ldots, s_n) = t$ when $s_1 \in S_1, \ldots, s_n \in S_n$ and $t \in T$. Using the object-oriented paradigm, function f is applied to object s_1 using objects $s_2, \ldots, s_n$ as parameters resulting in object t. Subsequent definitions will restrict $S_1, \ldots, S_n$ and T to insure substitutability and a strongly typed system. Later, sets of methods will be associated to classes and all methods associated with a class will be required to have unique names. The following definition formalizes this constraint.

Definition 2.5 *Consistent set of methods:* A set of methods M is *consistent* iff:

$$\forall m_i, m_j \in M, name(m_i) \neq name(m_j), i \neq j \quad \square$$

A class defines the behavior of its instances by providing methods which operate on those instances. However, a class may additionally inherit behavior from its parents. The functions which implement the methods of the parent class are only applicable to instances of the parent class, they can not be applied to instances of a subclass. The transformation operator '$\circ$' modifies inherited methods such that they are applicable to instances of a subclass.

Definition 2.6 *Method Transformation:* Given a method m and a set of objects Ω:

$$m \circ \Omega \quad \equiv m' = (n', f' : S' \times T', b') \mid [\quad \begin{array}{l} n' = n, \\ S' = \Omega \times S_2 \times \ldots \times S_n, \\ T' = T, \\ b' = b \quad] \end{array} \quad \square$$

The transformation operator takes a method m and a set of objects Ω and creates a new method m' with the same name, behavior, parameters and target domain but which operates on the objects in Ω instead of the source domain of the original method. In the case of an interpreted language with dynamic binding like Smalltalk-80 where representations and implementations are inherited, the transformation operator is a no-op. In compiled languages, the transformation operator is an abstraction of that aspect of the compiler which implements the language's inheritance mechanisms.

The following definition formalizes the notion of a class and shows how the data model implements behavioral inheritance. The transformation operator is used to build a set of *applicable methods* from locally defined and inherited methods.

Definition 2.7 *Classes:* A class c is defined as the tuple $c = (n, p, MT, AM)$ where:

- $n \in CN$ is the class name.
- p is a sequence of parent classes of the form $< c_1, \ldots, c_k >$.
- MT is a consistent set of methods with the added restrictions:
  - $\forall m \in MT, source(m) = ext(c)$. Each method in MT must operate on all instances of the class, and no other objects. In other words, the class' methods are defined only on the objects in the extent of the class.
  - $\forall m \in MT$, source domains $S_2(m), \ldots, S_i(m), \ldots, S_n(m)$ are the deep extent of some class c'_i, $ext^*(c'_i)$, in the database. This restriction is based on the principle of substitutability implied by the conformity relationship. Substitutability states that an instance of a subclass is acceptable in any context specifying an instance of the parent class. Here, the parent class is c'_i. In order to meet the substitutability criterion then, the function which implements a method must accept all instances of c'_i and all subclasses of c'_i as arguments, i.e., $ext^*(c'_i)$.
  - $\forall m \in MT$, the target domain $T(m)$ is the deep extent of some class c', $ext^*(c')$, in the database. This restriction insures that the result of a method is well defined thereby allowing a strongly typed system, i.e., objects returned by method m are guaranteed to minimally conform to class c'.
- AM is a set of *applicable methods* formed by:

$$MT(c) \cup (AM(c_1) \circ ext(c)) \cup \ldots \cup (AM(c_k) \circ ext(c))$$

The applicable methods of a class consists of the methods the class provides, MT, plus the methods from its parents transformed to operate on its own instances[2]. □

2.4 Databases

Definition 2.8 *Database:* A database is a pair $db = (\mathcal{H}, \Theta)$ where $\mathcal{H}$ is a set of classes denoting a class hierarchy and Θ is a consistent set of objects meeting the following constraints:

1. $\forall c \in \mathcal{H}$, the name of c is unique.
2. $\forall o \in \Theta$, $o.cn$ is the name of a class in $\mathcal{H}$.
3. $c_{root} \in \mathcal{H}$.
4. $\forall c \in \mathcal{H}$, c is a subclass of c_{root}. □

The first two conditions insure that each object in the database belongs to only one class which is itself in the database. The third and fourth conditions are required to insure that the recursive definition for behavioral inheritance terminates for every class in the database.

A legal database operation can be thought of as applying the function associated with a method to a sequence of objects where the first object in the sequence is in the extent of the class on which the method is defined.

Definition 2.9 *Operation:* A database *operation* is defined as the function $op : db\times < o_1, \ldots, o_n > \times mn \rightarrow r$ where:

- db is a database.
- $< o_1, \ldots, o_n >$ is a sequence of objects in db with at least one member.
- mn is the name of a method defined on the class of o_1.
- r is the resulting object obtained when the function associated with the named method, $f : S_1 \times S_2 \times \ldots \times S_n \rightarrow T$, is applied to $< o_1, \ldots, o_n >$. Intuitively, the method with name mn is being applied to object o_1 with parameters $o_2, \ldots, o_n$. □

We introduce the notation $op(db, < o_1, \ldots, o_n >, mn)$ to indicate the result of the operation $db\times < o_1, \ldots, o_n > \times mn$.

It is often desirable to perform a sequence of operations with the implicit assumption that the result of the first operation is to be used as the input to the second, etc. This is analogous to composition of the functions associated with methods. We call such a sequence of *m*ultiple *op*erations a *mop* and introduce the notation $<o_1, \ldots, o_n>.mn_1, \ldots, mn_m$ where:

[2]This assumes that name conflicts between methods in MT and any parent class are already resolved.

- a database db is implied.
- $<o_1, \ldots, o_n>$ is a sequence of objects in db.
- $<mn_1, \ldots, mn_m>$ is a sequence of method names where $mn_i \in MN$.
- Let k_i denote the number of parameters[3] required by method mn_i where $1 \leq i \leq m$, then r is the result of the multi-operation obtained by:

$$\begin{aligned}
r_1 &== op_1(db, <o_1, \ldots, o_{k_1+1}>, mn_1) \\
r_2 &== op_2(db, <r_1, o_{k_1+2}, \ldots, o_{(k_1+k_2+1)}>, mn_2) \\
&\vdots \qquad\qquad \vdots \\
r_i &== op_i(db, <r_{i-1}, o_{(\sum_{j=1}^{i-1} k_j)+2}, \ldots, o_{(\sum_{j=1}^{i} k_j)+1}>, mn_i) \\
&\vdots \qquad\qquad \vdots \\
r = r_m &== op_m(db, <r_{m-1}, o_{(\sum_{j=1}^{m-1} k_j)+2}, \ldots, o_n>, mn_m)
\end{aligned}$$

This sequence of operations is illustrated in Figure 2.2. □

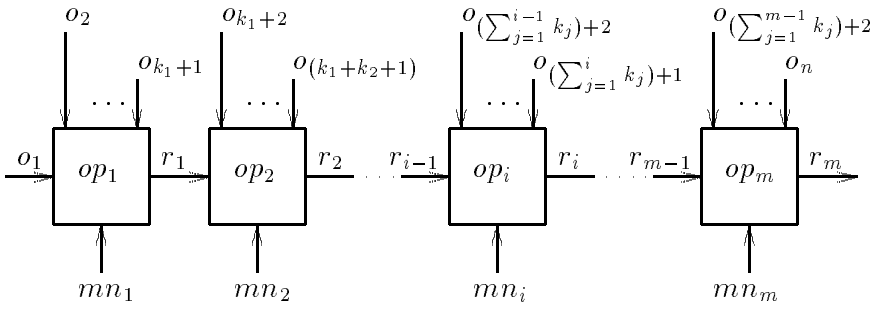


Figure 2.2: Sequence of operations making up a *mop*.

The multi-operation dot notation will be used in the remainder of the document. $<o_1, \ldots, o_n>.mlist$ will be used to denote a multi-operation where the number of method names in the method list is immaterial.

2.5 Example Database

The hypertext application is selected as an example because it belongs to an application domain (office information systems) that is claimed to potentially benefit from the object-oriented database technology. Specifically, a hypertext system requires persistent data, has a large number of data types and many types of ad hoc queries can be posed. The basic underlying concept is a simple one. Windows on the screen are associated with units of information stored in a database [Con87]. Information

[3]The parameters are those objects supplied as arguments to a method in addition to the object the method is applied to. In other words, a unary method has zero parameters.

units are related to one another via *links*. Links are typed in the sense that some may be used to specify the structural composition of a document (structural links), some may point to related information which supports the primary theme (referential links), and some may point to comments made by reviewers (note links). Users of the hypertext system browse through documents by traversing links and examining nodes of interest. This approach is a powerful communications tool as documents do not need to be structured linearly and users can sidetrack to follow related trails of information in whatever order they desire.

Information units are referred to as *nodes* and can encompass text, graphics, computer generated sound, and even executable programs. The example will be restricted to textual nodes. A *document* is a set of nodes connected by links with one node designated as the *root* node. Figure 2.3 depicts a hypertext system with structural links shown as solid lines and referential links shown as dotted lines. The nodes labeled **A** and **B** are root nodes. Documents can have any structure desired. Here the documents rooted at **A** and **B** are linear and hierarchical respectively. In general, there are no restrictions on links thereby allowing nodes to be a part of multiple documents, as in the case of **C**, or to exist outside of a document as in the case of nodes **D**, **E** and **F**. The forest of links associated with a document or group of documents is called a *web*.

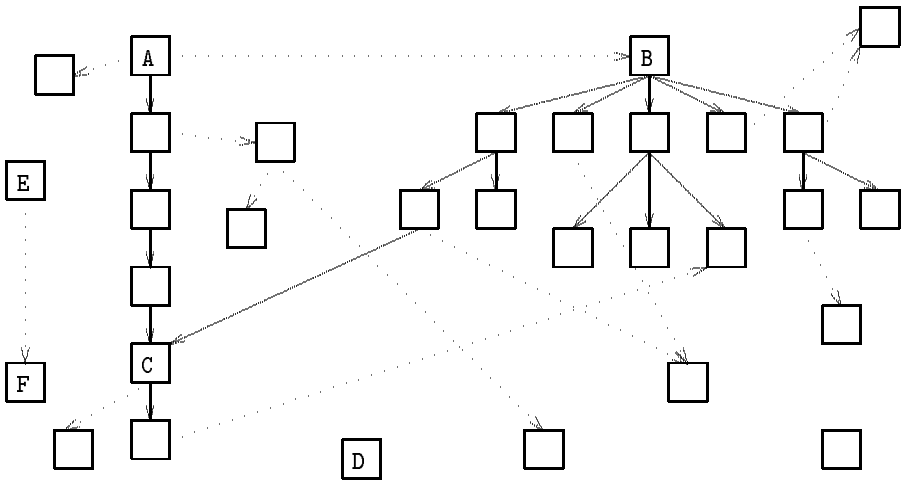


Figure 2.3: A *web* of hypertext nodes.

The hypertext database can be browsed in three ways. One method is to follow links and to open windows on nodes to examine their contents[4]. Another method is to graphically display the web associated with a document and selectively examine nodes of interest. Third, the database can be queried to identify nodes meeting some criteria. Nodes are qualified using selection criteria appropriate to the node type. For example, textual nodes may be selected based upon a keyword search while graphics nodes are selected based upon pattern recognition. The query mechanism can also be used to filter the nodes and links presented to the user when viewing the web of a

[4]Most systems implement link following and window invocation as a single mouse command.

document. Schatz and Caplinger [SC88] note that as a hypertext system grows, its web becomes less connected. This is due to the existence of documents which do not reference one another. In this situation, link following and web display as methods for finding related units of information are of limited usefulness. As a result, the ad hoc query capabilities become more important as the hypertext system grows in size.

The design of the user interface contributes greatly to the usefulness of a hypertext system. The ease and speed with which links can be followed and windows opened on information units can make the difference between a system which augments concurrent thought processes and one which merely stores large amounts of related data. Although implementations such as KMS [AMY88], Notecards [Hal88] and Intermedia [Con87] each have a unique user interface, a common, low level architecture can be identified. Campbell and Goodman [CG88] call this common set of features the *Hypertext Abstract Machine* (HAM) and show how several well known systems can be implemented on the standardized hypertext subsystem. The example implements a subset of the HAM using the object-oriented database model presented in this chapter.

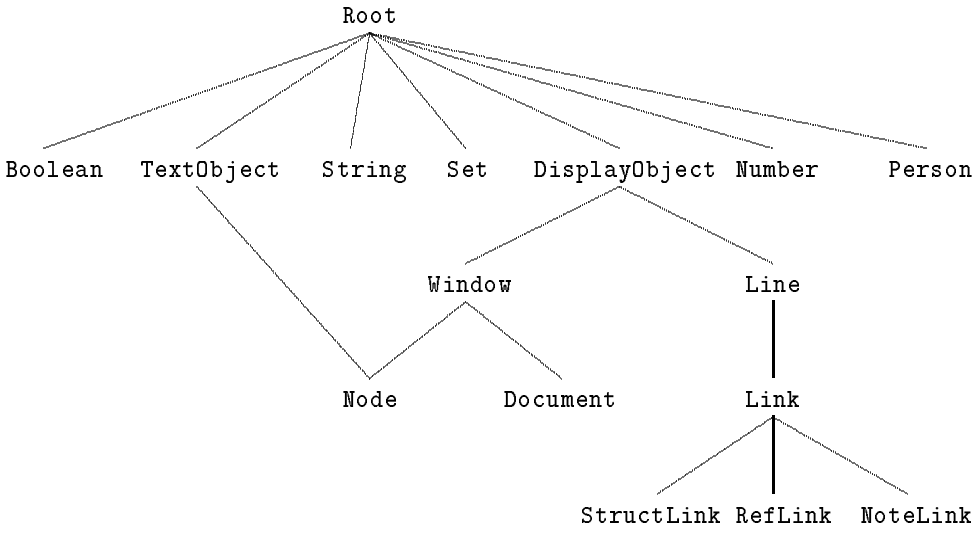


Figure 2.4: Classification graph for a hypertext system.

The class lattice for the hypertext database is given in Figure 2.4. The classes *Boolean*, *String*, *Set*, and *Number* should be considered as being predefined by the database management system while all other classes are defined by the hypertext database implementor. The signatures of methods defined by each class are given in Table 2.1. The classes *Node*, *Link*, and *Document* closely reflect the logical hypertext structure described earlier. However, some implementation details are significant. Since a node may belong to several documents concurrently, the links emanating from it belong to several documents as well. The method $links : Node \times Doc \rightarrow Set$ returns the set of links emanating from a node for a given document.

Links identify their source and destination node by means of the methods *from* and *to*. Unlike the HAM which tags links with an attribute, we have chosen to separate link types by defining the subclasses *StructLink*, *RefLink* and *NoteLink*. This provides

Table 2.1: Method signatures for classes of the hypertext database.

Name	Method Signatures
Root	
Boolean (Bool)	*negate:* $Bool \rightarrow Bool$
TextObject (TO)	*contains:* $TO \times Str \rightarrow Bool$ *creator:* $TO \rightarrow Person$ *keywords:* $TO \rightarrow Set$ *edit:* $TO \rightarrow TO$
String (Str)	*contains:* $Str \times Str \rightarrow Bool$ *concat:* $Str \times Str \rightarrow Str$ *within:* $Str \times Str \times Num \rightarrow Bool$
Set	*size:* $Set \rightarrow Num$ *add:* $Set \times Root \rightarrow Set$
DisplayObject (DO)	*display:* $DO \rightarrow DO$ *isColor:* $DO \rightarrow Bool$
Number (Num)	*add:* $Num \times Num \rightarrow Num$ *greater:* $Num \times Num \rightarrow Bool$
Person	*age:* $Person \rightarrow Num$ *expertise:* $Person \rightarrow Set$ *mother:* $Person \rightarrow Person$ *father:* $Person \rightarrow Person$ *children:* $Person \times Person \rightarrow Set$
Window	
Link	*creator:* $Link \rightarrow Person$ *from:* $Link \rightarrow Node$ *to:* $Link \rightarrow Node$ *part_of:* $Link \rightarrow Doc$
Node	*links:* $Node \times Doc \rightarrow Set$
Document (Doc)	*author:* $Doc \rightarrow Person$ *co-authors:* $Doc \rightarrow Set$ *title:* $Doc \rightarrow Str$ *keywords:* $Doc \rightarrow Set$ *rootnode:* $Doc \rightarrow Node$
RefLink	
NoteLink	
StructLink	

the opportunity to a-priori restrict the scope of a query by defining it to range over the appropriate subclass of *Link*. *Link* is a specialization of the graphical object class *Line*.

Documents and nodes both require a display ability and thus are a subclass of *Window*. *Node* additionally inherits its text handling behavior from *TextObject* which defines methods such as *contains* for testing substring containment and *edit*. Although a document is a collection of nodes, it would be incorrect to implement *Document* as a specialization of *Node*. Instead, instances of *Document* have a structural value which captures the document structure but is hidden from users. Access to this structure is provided by methods on the *Document* class thereby preserving the ADT abstraction.

Chapter 3

The Object Calculus

3.1 Introduction

Three key trade-offs of OODB query facilities can be identified: (1) formal *vs* ad hoc query languages, (2) predicates based upon structure *vs* behavior, and (3) object-preserving *vs* object-generating operations. The goal is not to argue either side of these trade-offs. Compromises need to be made with respect to each of these features when designing a practical, user query language. However, a formal, behaviorally based, object-preserving query language provides the best foundation for formal analysis of OODB query processing issues and results of this analysis should apply to many other OODB query models.

Formal query languages [Osb88, SZ90] have several properties not found in ad hoc query languages [Fis87, Kim87, MSOP86] which make them more suitable for formal analysis. Their semantics are well defined which simplifies formal proofs about their properties. Common types of formal query languages are a calculus or an algebra. A calculus allows queries to be specified declaratively without any concern for processing details. Queries expressed in an algebra are procedural in nature but can be optimized. Algebras provide a sound foundation for rule-based transformation systems [Fre87, GD87, HFLP89] which allow experimentation with various optimization strategies. A large body of work exists on algebras for other data models [AB84, JS82, Ull82]. Defining OODB query requirements in terms of an algebra facilitates comparisons with these other models.

Some models implement complex objects whose internal structure is visible [BK86, LRV88, Osb88] while others view objects as instances of abstract data types (ADT) [ACO85, SZ90]. Access to objects which are instances of an ADT is through a public interface. This interface defines the behavior of the object. Although the two views of objects appear incompatible, the ADT approach can effectively model complex objects by including *get* and *put* methods for each of the components of the internal structure [Zdo86]. Thus, a query language which supports predicates based on object behavior is more general while still allowing knowledge of object representations to be introduced in a later stage of query processing.

A distinction can be made between object-preserving and object-creating query operations [SS90]. Object-preserving query languages [ASL89, ACO85, MSOP86]

return objects which exist in the original database. Object-creating languages [Kim89, LRV88, Osb88, SZ90] answer queries by creating new objects from other objects. The new objects have a unique identity and some criterion is used to appropriately place them in the type inheritance graph. In one sense this violates the integrity afforded by objects with identity as objects with no apparent relation to each other can be combined and presented as a new object which (presumably) encapsulates some well defined behavior. But the requirement for combining objects into new relationships does exist; either for output purposes or for further processing as in knowledge bases where knowledge is acquired by forming new relationships among existing facts.

This thesis restricts the query formalism to object-preserving operations for two reasons. First, any OODB query language must have a complete object-preserving query facility independent of whether it additionally creates new objects. The ability to retrieve any object in the database utilizing relationships defined by the type inheritance graph or defined by ADT operations on objects is a fundamental requirement. Second, object-creating operations raise a number of issues which are not the focus of this research, e.g., what is the class of the created objects and what operations do they support.

The remainder of this chapter develops an object calculus for the model of Chapter 2. Primitive query operations and their semantics are developed in Section 3.1. The calculus is formally defined in Section 3.2. Sections 3.3 and 3.4 examine several characteristics of object calculus expressions and provide a definition of safety for queries.

3.2 Query Primitives

In principle, maintaining the data abstraction paradigm would require querying the database based on object behaviors, not their values. However, since every information system actually stores values and uses functions to implement behavior, queries need to specify a combination of values and functions. Four comparison operators which can be used in queries are defined: $==$, $\in$, $=_{\{\}}$, and $=$ whose semantics are shown in Tables 3.1 and 3.2. The $==$ operator tests for object identity equality; i.e., $o_i == o_j$ evaluates to true when o_i and o_j denote the same object. The $\in$ and $=_{\{\}}$ operators apply to set valued objects and denote set value inclusion and set value equality respectively. As shown in the tables, one of the operands can denote a value. The last operator, $=$, can only be used to test the value of an atomic object. In order to maintain data abstraction, no primitives are provided for querying structural values. Any aspect of structural values which are required by users of an object should be made available via methods by the class implementor.

Atoms are the building blocks of calculus expressions and predicates for qualifying algebra operators. They represent the primitive query operations of the data model and return a boolean result. The legal atoms are as follows:

- $o_i \theta o_j$ where
  - o_i and o_j are object variables or denote an operation of the form

Table 3.1: Semantics of $o_i \theta o_j$ as a function of the object value type.

		$o_i \theta o_j$			
o_i	o_j	==	=	$\in$	$=_{\{\}}$
atomic	atomic	T/F	T/F	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	undefined
structural	atomic	T/F	undefined	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	undefined
set	atomic	T/F	undefined	undefined	undefined
	structural	T/F	undefined	undefined	undefined
	set	T/F	undefined	T/F	T/F

Table 3.2: Semantics of $a \theta o_i$ as a function of the object value type.

		$a \theta o_i$			
a	o_i	==	=	$\in$	$=_{\{\}}$
val_1	atomic	undefined	T/F	undefined	undefined
	structural	undefined	undefined	undefined	undefined
	set	undefined	undefined	T/F	undefined
$\{val_1, \ldots, val_n\}$	atomic	undefined	undefined	undefined	undefined
	structural	undefined	undefined	undefined	undefined
	set	undefined	undefined	undefined	T/F

  $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables,
- θ is one of the operators ==, =, $\in$ or $=_{\{\}}$.

- $a \theta o_i$ where
  - o_i is an object variable or denotes an operation of the form $<o_1 \ldots o_n>.mlist$ where $o_1 \ldots o_n$ are object variables,
  - a is the textual representation of an atomic value or a set of atomic values,
  - θ is one of the operators =, $\in$ or $=_{\{\}}$.

Example 3.1 Let p, q and r be object variables. Then the following are examples of legal atoms and their semantics:

1. $(p == q)$ – Are the objects denoted by p and q the same object?
2. $(p \in <q, r>.mlist)$ – Is the identifier of p contained in the set value of the object obtained by applying the methods in $mlist$ to the objects $<q, r>$?
3. $(<p, q>.mlist =_{\{\}} r)$ – Is the set value of the object obtained by applying the methods in $mlist$ to the objects $<p, q>$ pairwise equal to the set value of the object denoted by r?
4. ("59" $= p$) – Is "59" the atomic value of the object denoted by p?
5. ("59" $\in p$) – Does the set value of the object denoted by p include an identifier for the object whose atomic value is "59"?

6. ({"59", "61"} $=_{\{\}}$<p, q, r>.$mlist$) – Does the set value of the object obtained by applying the methods in $mlist$ to the objects <p, q, r> contain only two identifiers for objects whose atomic values are "59" and "61"? $\diamond$

3.3 The Object Calculus

The format of the object calculus definition is similar to the tuple relational calculus definition provided in [Ull82]. A query in the object calculus is of the form $\{o \mid \psi(o)\}$, where o is an object variable denoting some objects in the database and ψ is a formula built from atoms. The result of the query is the set of objects o which satisfy the predicate formed by $\psi(o)$. We introduce a third atom, specific only to calculus expressions, in addition to those defined in the previous section.

Range Atom: $C(o)$ or $C^*(o)$ where C is the name of a class and o is an object variable ranging over the instances of class C. $C(o)$ refers to the objects in the extent of C, i.e., $ext(C)$, whereas $C^*(o)$ refers to the objects in the deep extent of C, i.e., $ext^*(C)$.

Formulas depend on the notion of *free* and *bound* variables. A variable is said to be bound in a formula if it has been previously introduced using a quantifier such as $\exists$ or $\forall$. If the variable has not been introduced using a quantifier it is free in the formula. Formulas are defined as follows:

1. Every atom is a formula. All object variables in the atom are free in the formula.

2. If ψ_1 and ψ_2 are formulas, then $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$ and $\neg\psi_1$ are formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$ and $\neg\psi_1$ as they are free or bound in ψ_1 or ψ_2 depending on where they occur.

3. If ψ is a formula, then $(\exists o)(\psi)$ is a formula. Free occurrences of o in ψ are bound to $(\exists o)$ in $(\exists o)(\psi)$.

4. If ψ is a formula, then $(\forall o)(\psi)$ is a formula. Free occurrences of o in ψ are bound to $(\forall o)$ in $(\forall o)(\psi)$.

5. Formulas may be enclosed in parenthesis. In the absence of parenthesis, the decreasing order of precedence is $\in$, $=$, $=_{\{\}}$, $==$, $\exists$, $\forall$, $\neg$, $\wedge$ and $\vee$, in that order.

A query is an *object calculus expression* of the form $\{o \mid \psi(o)\}$ where o is the only free variable in ψ.

Example 3.2 Using the database of Figure 2.4, the following sample queries can be formulated as object calculus expressions.

1. Author of the document titled 'As We May Think' [Bus45]:

$$\{\ o \quad | \quad \exists p(Doc(p) \wedge o ==<p>.author \\ \wedge\ \text{"}As\ We\ May\ Think\text{"} =<p>.title)\ \}$$

2. Nodes belonging to the document titled 'As We May Think':

$$\{\ o \ \mid\ \exists p(Doc(p) \wedge \text{“}As\ We\ May\ Think\text{”} = <p>.title \wedge \exists q(Link^*(q) \wedge p == <q>.part\_of \wedge (o == <q>.from \vee o == <q>.to)))\ \}$$

3. Documents co-authored by a person's father:

$$\{\ o \ \mid\ \exists p(Doc(o) \wedge p == <o>.author \wedge \exists q(q \in <o>.co\_authors \wedge q == <p>.father))\ \}$$

4. Authors who only write about boating:

$$\{\ o \ \mid\ \forall p(Doc(p) \ \wedge\ o == <p>.author \ \wedge\ \text{“}Boating\text{”} \in <p>.keywords)\ \} \quad \diamond$$

At this point it is appropriate to comment on the choice of atoms for the object calculus as some of them seem quite restrictive as compared to the tuple relational calculus. For example, the tuple relational calculus allows the operator θ to be one of $=$, $<$, $\leq$, $>$ or $\geq$ whereas the object calculus restricts θ to $=$, $==$, $=_{\{\}}$ or $\in$. The object calculus allows a value based equality comparison of atomic objects only, not of complex objects.

Some researchers have proposed *shallow* and *deep* equality operators which can be applied to objects of any class [GR85, KC86, LRV88]. Two objects are said to be shallow-equal if their values are identical. Two objects are said to be deep-equal if (1) they are atomic objects and their values are equal, or (2) they are set objects and their elements are pairwise deep-equal, or (3) they are tuple objects and the values they take on the same attributes are deep-equal.

The object calculus defined here avoids these operators for two reasons. First, a value based comparison of complex objects, such as $o_i = o_j$, where o_i and o_j are complex objects, violates the principle of abstract data types whose instances are solely defined by their behavior. In order to completely support encapsulation, one can not allow query expressions whose results are dependent on equivalence of structure as opposed to equivalence of behavior. Second, the model should allow various objects of the same class to be implemented differently to take advantage of their environment. For example, different representations may be used when objects are in main memory versus when they are stored on secondary storage. Furthermore, if distribution and heterogeneity is considered, then objects may be represented differently on different machines. Therefore, the notion of an equivalence test which depends on representation is inappropriate.

A similar argument can be made for prohibiting the use of comparison operators other than $=$ on atomic objects. User knowledge of the values in a domain does not necessarily imply knowledge about their ordering. As an example, consider the case of a Ceasar cipher where all letters are shifted by n characters. With $n = 5$, the encoded form of 'hello world' would be 'czggj rjmgy'. A database might contain the class *CipherAlphabet* whose value domain is the letters of the alphabet and whose total ordering is $< v, \ldots, z, a, \ldots, u >$. Obviously the $<$ relation on members of

CipherAlphabet is not the same as the $<$ relation on the standard alphabet even though the value domains are identical. For this reason, all value comparison operations other than $=$ must be implemented by a class in accordance with the total ordering the class defines.

3.4 Finiteness

A traditional notion in relational databases is that relations are finite even if the attribute values within the relation may be drawn from an infinite domain. This restriction is the basis for defining the *domain independent* class of queries [Fag82, GT87, Nic82]. A relational query is domain independent if it can be answered by simply considering the collections of tuples of the relations named in the query and ignoring the underlying domains of the attributes. The corresponding notion in object-oriented databases would be that the extents of classes are finite while the value components of objects may be drawn from infinite domains. Definitions 2.1 and 2.2 indeed allow for the value component of objects to be drawn from infinite domains. However, whether a class extent is finite depends on the methods defined by that class.

Consider the class *Number* from the example in the previous section. The signature of the method *add* was given as:

$$add : Num \times Num \rightarrow Num$$

A problem arises due to the restriction on classes (Definition 2.7) which states that any method defined on the class must apply to all objects in the extent of the class. Thus if 1 is in the database, 2 needs to be in the database as well $(1 + 1)$. This scenario is not limited to mathematical concepts such as closure under addition, it can occur in any class which has a method which maps to members of its own class. For example, the method on *String*

$$within : Str \times Str \times Num \rightarrow Bool$$

presents no problems while

$$concat : Str \times Str \rightarrow Str$$

implies the existence of an infinite number of strings ('a', 'aa', 'aaa', etc).

Note that not every method which maps to members of the class on which it is defined causes problems. For example, in the class *Person*, the method with signature

$$father : Person \rightarrow Person$$

does not imply an infinite number of person objects as long as we allow some persons to have a NULL father or to return themselves when the *father* method is applied.

Clearly the intended semantics of the class and associated methods determine whether a class is finite or not. We assume in the following that all infinite classes (and their associated methods) are implemented as primitive classes, i.e., provided by the database system. User defined classes are restricted to finite classes only. In the sample hypertext database, *String* and *Number* are infinite classes.

3.5 Safety of Object Calculus Expressions

The object calculus is extremely expressive and allows the formulation of queries with no reasonable interpretation or which do not have finite output. For example, $\{ o \mid \neg C(o) \}$ denotes all possible objects which are not in the extent of class C. Similarly, $\{ o \mid C^{\infty}(o) \}$ where C^{∞} is an infinite class such as *Number* denotes a well defined, although infinite set of objects. To avoid such meaningless constructs, we restrict ourselves to expressions considered safe.

Definition 3.1 *Safety:* An object calculus expression is considered *safe* if it can be evaluated in finite time and produces finite output [OW89]. □

The previous definition is a semantic one. The following provides a syntactic definition such that any object calculus expression can easily be tested for safety.

Definition 3.2 *Restricted variable:* Let $F(x_1, \ldots, x_k)$ be a conjunction of (possibly negated) atomic formulas. We say that a variable x_i is *restricted* in F if one of the following appears in F:

1. $C(x_i)$ or $C^*(x_i)$ without a negation to its immediate left where C is a finite class.
2. $(x_i\ \theta\ o)$ with positive polarity where $\theta \in \{==, \in\}$ and o is an object variable restricted in F.
3. $(x_i\ \theta\ <o_1, \ldots, o_n>.mlist)$ with positive polarity where $\theta \in \{==, \in\}$ and $o_1, \ldots, o_n$ are restricted and $x_i \notin \{o_1, \ldots, o_n\}$.
4. $(a = x_i)$ with positive polarity where a is the textual representation of an atomic value or a set of atomic values. □

The atoms defined by cases 2–4 above are called *generating atoms* for the variable x and are enumerated in Table 3.3. They are so named because they generate objects for x from a constant value, the content of other objects, or by applying methods to other objects. Atom 1 in the table generates values which are identical to values for o. In practice, all occurrences of x are replaced by o. In atom 2, x ranges over the identifiers contained within the set value of o. Atoms 3 and 4 are similar to 1 and 2 except that the sequence of methods *mlist* is applied to object o_1 prior to generating values for x. Atom 5 generates a value for x from the textual representation of the atomic value a.

Generating atoms are important because they can be used to generate responses to queries without directly ranging over the classes which x is an instance of. Instead, the query ranges over the classes of other variables in the generating atom.

Table 3.3: Generating atoms for x.

1	$x == o$
2	$x \in o$
3	$x ==<o_1, \ldots, o_n>.mlist$
4	$x \in<o_1, \ldots, o_n>.mlist$
5	$x = a$

Example 3.3 Consider the query which returns those children who have a geneticist and poet for parents:

$$\{\ o \ \mid\ \exists p(Person(p) \wedge \text{“}genetics\text{”} \in<p>.expertise \wedge \exists q(Person(q) \wedge \text{“}poetry\text{”} \in<q>.expertise \wedge o \in<p, q>.children))\ \}$$

In this query $o \in<p, q>.children$ is a generating atom as o does not range over a class in the database. Instead, o ranges over the set valued objects returned by the method application $<p, q>.children$. $\diamond$

Theorem 3.1 Let ψ' be the prenex disjunctive normal form (PDNF) of ψ. The object calculus expression $\{\ o \mid \psi(o)\ \}$ is safe iff o is the only free variable in $\psi(o)$ and all other variables are bound and restricted.

Proof: We first show by case analysis that x_i in Definition 3.2 always refers to a finite set of objects.

Case 1: $C(x_i)$ or $C^*(x_i)$

By definition of finite classes, the (deep) extent of C is a finite set of objects.

Case 2: $x_i == o$

By definition of identity equality ($==$) x_i and o denote the same object, thus if o is restricted, x_i denotes a finite set of objects as well.

Case 3: $x_i \in o$

Since o is restricted, it ranges over a finite set of objects. Each set value in the database is a finite set of objects. Thus x_i ranges over the members of finite sets within a finite set which is itself a finite set.

Case 4: $x_i ==<o_1, \ldots, o_n>.mlist$

Since $o_1, \ldots, o_n$ are all restricted, the operation $<o_1, \ldots, o_n>.mlist$ denotes a finite set of objects. By definition of identity equality ($==$) then, x_i denotes a finite set as well.

Case 5: $x_i \in<o_1, \ldots, o_n>.mlist$

Since $o_1, \ldots, o_n$ are all restricted, the operation $<o_1, \ldots, o_k>.mlist$ denotes a finite set of objects. Thus x_i denotes a finite set of objects as in case 3.

Case 6: $a = x_i$

x_i denotes the single object whose value is a.

Since x_i is restricted, the output of each disjunct is finite and since the query has a finite number of disjuncts it can be evaluated in finite time. $\square$

Example 3.4 The following queries illustrate the nature of safe and unsafe queries.

1. Although o ranges over an infinite class in the following query, the query is safe since o is restricted by the atom $o ==<p>.age$.

$$\{\ o \quad | \quad Number(o) \wedge \exists p(Person(p) \wedge o ==<p>.age)\ \}$$

2. The following query is not safe as o is not restricted by any atom and $Number$ is an infinite class.

$$\{\ o \quad | \quad Number(o) \wedge \exists p(\text{“}65\text{”} = p \wedge \text{“}True\text{”} =<o, p>.greater)\ \}$$

3. Let % stand for either the universal ($\forall$) or existential ($\exists$) quantifier and C stand for a finite class. The following query is safe as x_1 is restricted by ranging over the extent of a finite class and all other terms correspond to case 4 in Theorem 3.1.

$$\begin{aligned}\{\ o \quad | \quad (\%x_2 \ldots \%x_n)(&C(x_1)\\ &\wedge\ x_2 ==<x_1>.mlist_1\\ &\wedge\ x_3 ==<x_2>.mlist_2\\ &\qquad\vdots\\ &\wedge\ o ==<x_n>.mlist_n\ \}\end{aligned}$$ $\diamond$

Chapter 4

The Object Algebra

4.1 Introduction

This chapter[1] presents an object algebra which implements a subset of the object calculus. The algebra operators and their semantics are defined in Section 4.1. Section 4.2 presents an algorithm for translating a restricted class of object calculus expressions into object algebra expressions.

4.2 The Object Algebra

Operands and results in the object algebra are sets of objects. Thus the algebra maintains the closure property [ASL89] where the result of a query can be used as the input to another. Some of the operators accept more than two operands. Let Θ be an operator in the algebra. The notation $P\ \Theta\ \langle Q_1 \ldots Q_k \rangle$ will be used for algebra expressions where P and Q_i denote sets of objects which are arguments to the operator Θ. In the case where $k = 1$ we will use $P\ \Theta\ Q$ and where $k = 0$ we will use $P\ \Theta\ \langle\ \rangle$ without loss of generality.

Some of the algebra operators are qualified by a predicate. Such operators will be written $P\ \Theta_F\ \langle Q_1 \ldots Q_k \rangle$ where F is a formula consisting of one or more atoms connected by $\wedge$, $\vee$, or $\neg$ using parenthesis as required. Atoms reference lower case, single letter variables which range over objects in the input set named with the corresponding upper case letter. For example, the object variables p, q_1 and q_2 in the predicate of $P\ \Theta_{F(p,q_1,q_2)}\ \langle Q_1, Q_2 \rangle$ range over the sets of objects denoted by P, Q_1 and Q_2 respectively.

The algebra defines five operators.

Union (denoted $P \cup Q$): The union is the set of objects which are in P or Q or both. An equivalent expression for union is $\{\ o \mid P(o) \vee Q(o)\ \}$.

[1]Portions of this chapter have been accepted for publication in the *Proc. of the X3/SPARC/DBSSG/OODB Standardization Workshop* held in conjunction with the May, 1990 SIGMOD conference in Atlantic City, New Jersey.

Difference (denoted $P - Q$): The difference is the set of objects which are in P and not in Q. An equivalent expression for difference is $\{\ o \mid P(o) \wedge \neg Q(o)\ \}$. The intersection operator, $P \cap Q$, can be derived by $P - (P - Q)$.

Select (denoted $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$): Select returns the objects denoted by p for each vector $< p, q_1, \ldots, q_k > \in P \times Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for select is $\{\ p \mid P(p) \wedge Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge F(p, q_1, \ldots, q_k)\ \}$.

The select is similar to, but more powerful than, that of [SZ90] which allows only one operand. Multiple operands permit explicit joins as described in [Kim89]. An explicit join is a join between arbitrary classes which support (a sequence of) method applications resulting in comparable objects.

Example 4.1 Find all documents about cars by persons over 50 years of age. Let d range over *Doc* and p range over *Person*, then

$$Doc\ \sigma\left[\begin{array}{c}\text{“}car\text{”} \in <d>.keywords\ \wedge \\ p == <d>.author\ \wedge \\ \text{“}50\text{”} = x\ \wedge\ \text{“}True\text{”} = <p,x>.age.greater\end{array}\right]\ \langle Person \rangle\ \ \diamond$$

The result of this expression is a set of *Document* objects, not sets of $< Document, Person >$ objects. This is due to the 'object preserving' nature of the algebra which does not support creation of new objects. In this sense then, the select is most like the traditional semi-join operator. As a result, the selection $P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$ always returns a subset of P.

Generate (denoted $Q_1\ \gamma_F^t\ \langle Q_2 \ldots Q_k \rangle$): F is a predicate with the condition that it must contain one or more generating atoms for the target variable t and t does not range over any of the argument sets. The operation returns the objects denoted by t in F for each vector $< q_1, \ldots, q_k > \in Q_1 \times \ldots \times Q_k$ which satisfies the predicate F. An equivalent expression for generate is $\{\ t \mid Q_1\,(q_1) \wedge \ldots \wedge Q_k\,(Q_k) \wedge F\,(t, q_1, \ldots, q_k))\ \}$.

Two common uses of the generate operator are to collect results of method applications or to iterate over the content of set valued objects.

Example 4.2 Return all co-authors of the document 'My Cat is Object-Oriented' [Kin89]. Let t be the target variable and d range over *Doc*, then

$$Doc\ \gamma^t\left[\begin{array}{c}\text{“}MyCat...\text{”} = <d>.title\ \wedge \\ t \in <d>.co\_authors\end{array}\right]\ \langle\ \rangle\ \ \diamond$$

Map (denoted $Q_1\ \mapsto_{mlist}\ \langle Q_2 \ldots Q_k \rangle$): Let *mlist* be a list of method names of the form $m_1 \cdots m_m$. Map applies the sequence of methods in *mlist* to each object $q_1 \in Q_1$ using objects in $\langle Q_2 \ldots Q_k \rangle$ as parameters to the methods in *mlist*. This returns the set of objects resulting from each sequence application. If no method in *mlist* requires any parameters, then $\langle Q_2 \ldots Q_k \rangle$ is the empty

sequence $\langle\ \rangle$. Map is a special case of the generate operator whose equivalent is $\{\ t \mid Q_1(q_1) \wedge \ldots \wedge Q_k(q_k) \wedge t == <q_1, \ldots, q_k>.mlist\ \}$. This form of the generate operation warrants its own definition as it occurs frequently and supports several useful optimizations. Map is similar to the **image** operator of [SZ90], except that it is not restricted to unary methods.

4.3 Calculus-Algebra Translation

This section presents an algorithm for translating a subset of object calculus queries to the object algebra.

Definition 4.1 *Restricted Query:* A *restricted query* is any safe object calculus query of the form $\{\ o \mid \exists p \exists q \exists r \ldots \psi(o, p, q, r, \ldots)\ \}$ where ψ does not contain any occurrences of either $\exists$ or $\forall$. $\square$

Restricted queries are similar in power to the select–project–join class of queries in the relational model. Even though this class contains a large number of practical queries, it does not include those that require universal quantification and recursion such as those often required in knowledge base applications. This restriction can be overcome by introducing additional algebra operators and will be a topic of future research.

We show, by presenting an algorithm for translating restricted object calculus queries to object algebra operator trees, that the algebra is capable of representing the entire class of restricted queries. The algorithm inserts matched pairs of parentheses into the disjuncts of the prenex disjunctive normal form (PDNF) of a query. The algorithm terminates successfully when the inner most nested expressions correspond to the range atoms of the query. If this step is not possible, then the query is unsafe. After rewriting, nested subexpressions are mapped directly to their object algebra counterparts.

We first introduce a simple notational convenience for describing atoms. Atoms are written as $a_i(g \mid r_1, r_2, \ldots, r_n)$ where i is the atom number, g represents a variable for which this atom generates values, and each r_j represents a variable restricted by the atom. We say that the atom *generates* g and *restricts* r_i. However, the atom *references* g and all r_i.

Example 4.3 The following expressions show the correspondence between atoms and their shorthand notation.

1	$Class\_Name(o) \rightarrow a_1(o \mid \_)$	range atom
2	$p \in <q_1, q_2>.mlist \rightarrow a_2(p \mid q_1, q_2)$	set inclusion
3	$<q_1, q_2>.a.b.c == <q_3, q_4>.d.e.f \rightarrow a_3(\_ \mid q_1, q_2, q_3, q_4)$	

$\diamond$

The calculus equivalents of the object algebra operators can now be easily expressed using this notation.

Table 4.1: Calculus and algebra equivalents using abbreviated notation.

$Q_1 \cup Q_2 \cup \ldots \cup Q_n$	$\equiv$	$\{\ q \mid Q_1(q \mid \_) \vee Q_2(q \mid \_) \vee \ldots \vee Q_n(q \mid \_)\ \}$
$Q_1 \cap Q_2 \cap \ldots \cap Q_n$	$\equiv$	$\{\ q \mid Q_1(q \mid \_) \wedge Q_2(q \mid \_) \wedge \ldots \wedge Q_n(q \mid \_)\ \}$
$P - Q$	$\equiv$	$\{\ p \mid P(p \mid \_) \wedge \neg Q(p \mid \_)\ \}$
$P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle$	$\equiv$	$\{\ p \mid F(\_ \mid p, q_1, \ldots, q_k) \wedge P(p \mid \_) \wedge Q_1(q_1 \mid \_) \wedge \ldots \wedge Q_k(q_k \mid \_)\ \}$
$Q_1\ \gamma_F^t\ \langle Q_2 \ldots Q_k \rangle$	$\equiv$	$\{\ t \mid F(t \mid q_1, \ldots, q_k) \wedge Q_1(q_1 \mid \_) \wedge \ldots \wedge Q_k(q_k \mid \_)\ \}$

The key to the translation algorithm is a recursive routine which given a target variable, an empty expression and a list of unused atoms will selectively place atoms in the empty expression. The routine additionally defines new, empty expressions which denote ranges for the variables referenced in the atoms just placed. The routine is called recursively on the newly defined empty expressions until all atoms have been used. Empty expressions are written as $(\ldots)_o$ where the subscript o indicates that the expression defines a set of objects for variable o.

Atoms will be selected for placement in the following order: (1) those which restrict the target variable, (2) those which generate the target variable, and (3) range atoms for the target variable. We introduce the notion of a *correlating variable* and *correlated atoms*. Consider an undirected bipartite graph G where one set of nodes represents variables, the other atoms, and edges connect atoms to the variables they reference. Two atoms a_i and a_j are said to be correlated by variable v if there exists a cycle in G which traverses a_i, a_j and v, not necessarily exclusively.

We also define a *conjunct dependency graph* (CDG) for a conjunct of atoms $a_1 \wedge a_2 \wedge \ldots \wedge a_n$. The graph contains a node for each variable v referenced by any atom a_i and a directed edge $v_1 \rightarrow v_2$ if there exists an atom $a_i(v_2 \mid \ldots v_1 \ldots)$, i.e., a_i generates values for v_2 using v_1 as input. We call v_2 the head and v_1 the tail respectively. Later algorithms will require that a CDG be *safe*. Assume that some nodes in a CDG are marked indicating that a set of values exists which the node's variable ranges over. Recalling that nodes with incoming edges represent variables whose values are generated from others, we can mark such nodes if the tails of all incoming edges are marked as well. In other words, all inputs are present for generating values for the head variable. We call the exhaustive propagation of node markings *completing* the CDG. If after completion there exist any unmarked nodes we say that the CDG is *unsafe*.

The atom placement routine is recursive and defined as follows:
$Place(v)$
Inputs:

- target variable v
- list of atoms $L(a_1, \ldots, a_n)$

Outputs:

- success/failure indicator
- nested expression of atoms

```
begin
    let BG be the bipartite graph for atoms in L
    select an atom a_i ∈ L which references v as per criterea given above
    if ( appropriate a_i not found ) then
        return(UNSAFE QUERY)
    else
        let C be the conjunct containing only a_i
        remove a_i from L
    endif
    if (a_i is a range atom for v (i.e., a_i(v | _) ) then
        while ∃ range atom a_j ∈ L for v do
            extend C with a_j via conjunction
            remove a_j from L
        endwhile
        return(OK)
    else
        for each variable v_j referenced by a_i do
            if BG defines v_j to be a correlating variable then
                for each a_k ∈ L correlated to a_i by v_j do
                    extend C with a_k via conjunction
                    remove a_k from L
                endfor
            endif
        endfor
        let CDG be the conjunct dependency graph for atoms in C
        for each node v in CDG do
            if ( v has no incoming edges ) or ( ∃ an atom a_k ∈ L which references v ) then
                mark node v
                extend C with an empty expression for v, (...)_v
            endif
        endfor
        complete CDG by propagating markings
        if ( ∃ any unmarked nodes in CDG ) then
            return(UNSAFE QUERY)
        endif
        for each empty expression (...)_v in C do
            if Place(v) ≠ OK then
                return(UNSAFE QUERY)
            endif
        endfor
        return(OK)
    endif
end.
```

We are now ready to specify the entire calculus to algebra translation algorithm.

Algorithm 4.1 *Translate*

Input: Object calculus expression : C
Output: Equivalent object algebra expression : A
begin
convert calculus expression to prenex disjunctive normal form (1)
for each disjunct **do** (2)
for each constant defining atom of the form $v = const$ **do** (3)
delete the atom and replace all other occurrences of v by $const$ (4)
endfor (5)
rewrite all atoms using the shorthand notation developed above (6)
for each atom $a(v \mid r_1, \ldots, r_n)$ for which there exists a range atom for v of the form $a(v \mid \_)$ **do** (7)
move v to the right hand side of the atom (8)
(i.e., $a(v \mid r_1, \ldots, r_n) \Rightarrow a(\_ \mid v, r_1, \ldots, r_n)$)
endfor (9)
for each atom of the form $a(g \mid g, r_1, \ldots, r_n)$ **do** (10)
rewrite the atom as $a(\_ \mid g, r_1, \ldots, r_n)$ (11)
endfor (12)
call $Place(t)$ where t is the target variable of C resulting in C' (13)
from innermost **to** outermost parenthesis nesting of C' **do** (14)
map the expression to equivalent algebra operators as per Table 4.3 (15)
endfrom (16)
endfor (17)
combine the algebra operator trees for each disjunct using Union (18)
end.

Example 4.4 We next illustrate Algorithm 4.1 via an extended example using the query

> Find all nodes belonging to the structural part of a document authored by a person who is retired, i.e., we want only the *StructLinks* of the document as opposed to *RefLinks* or *NoteLinks*.

This query can be expressed in the object calculus as:

$$\begin{aligned}\{\ o \ \mid\ & \exists p(Doc(p) \\ & \wedge \quad \exists q(\text{“}65\text{”} = q \wedge \text{“}True\text{”} = <p,q>.author.age.greater \\ & \wedge \quad\ \exists r(StructLink(r) \wedge p == <r>.part\_of \\ & \wedge \quad\ \ (o == <r>.from \vee o == <r>.to))) \ \}\end{aligned}$$

The atoms are numbered uniquely as follows:

$a_1 \quad Doc(p)$

a_2 "65" $= q$
a_3 "$True$" $=<p,q>.author.age.greater$
a_4 $StructLink(r)$
a_5 $p ==<r>.part\_of$
a_6 $o ==<r>.from$
a_7 $o ==<r>.to$

We first delete the constant defining atom a_2 and replace all occurrences of q with "65" (Steps 3–5). The two disjuncts of the prenex normal form of the query now are (Steps 1–5):

$$\begin{aligned} D_1 &\equiv a_1 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_6 \\ D_2 &\equiv a_1 \wedge a_3 \wedge a_4 \wedge a_5 \wedge a_7 \end{aligned}$$

We will perform the steps of the algorithm for disjunct D_1. The atoms are first written in the shorthand form introduced earlier (Step 5) resulting in:

$$a_1(p \mid \_) \wedge a_3(\_ \mid p) \wedge a_4(r \mid \_) \wedge a_5(p \mid r) \wedge a_6(o \mid r)$$

One of the rewrites identified in steps 7-12 of the algorithm is possible. Recognizing that a range atom exists for p in the query we can replace $a_5(p \mid r)$ by $a_5(\_ \mid p, r)$ resulting in the expression:

$$a_1(p \mid \_) \wedge a_3(\_ \mid p) \wedge a_4(r \mid \_) \wedge a_5(\_ \mid p, r) \wedge a_6(o \mid r)$$

Next, $Place$ is called for the target variable o. Atom a_6 is selected first since it is the only atom referencing variable o. The bipartite graph for this disjunct (see Figure 4.1) has no cycles, thus the atom does not need to be combined with any others. The CDG for a_6 is:

$$r \rightarrow o$$

Since r has no incoming edges we extend a_6 with an empty expression for r, $(\ldots)_r$. Variable o does not require an empty expression as it is generated by a_6 and no other atoms reference it. Our query expression is now:

$$a_6(o \mid r) \wedge (\ldots)_r$$

Due to its recursive nature, $Place$ is now called for the empty expression denoting r. Two unused atoms reference r, $a_4(r \mid \_)$ and $a_5(\_ \mid p, r)$. Using the selection criterea given earlier we select a_5 since it restricts r and a_4, which is a range atom for r, is saved for later. The CDG for this atom is:

$$r \rightarrow p$$

Both variables require empty expressions since unused atoms reference them. Our query expression is now:

$$a_6(o \mid r) \wedge (a_5(\_ \mid p, r) \wedge (\ldots)_p \wedge (\ldots)_r)_r$$

Place is called again, once each to expand the empty expressions for p and r. The expansion for p results in:

$$a_3(\_ \mid p) \wedge (\ldots)_p$$

and that for r:

$$a_4(r \mid \_)$$

Substituting into the partial query expression gives:

$$a_6(o \mid r) \wedge (a_5(\_ \mid p, r) \wedge (a_3(\_ \mid p) \wedge (\ldots)_p)_p \wedge (a_4(r \mid \_))_r)_r$$

Place is called one last time for p resulting in the final query expression:

$$a_6(o \mid r) \wedge (a_5(\_ \mid p, r) \wedge (a_3(\_ \mid p) \wedge (a_1(p \mid \_))_p)_p \wedge (a_4(r \mid \_))_r)_r$$

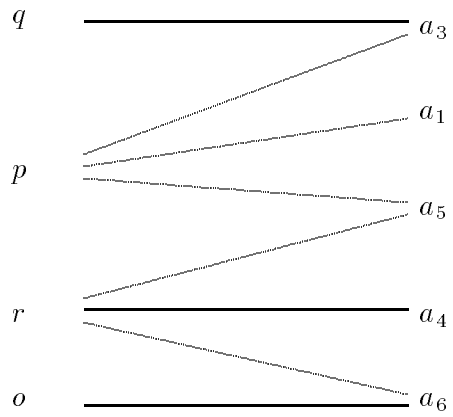


Figure 4.1: Query graphs for the sample query of Example 4.4.

The nested subsexpressions of the query can now be mapped to their object algebra counterparts of Table 4.3. Working from innermost to outermost nesting levels results in the object algebra expression:

$$(StructLink\ \sigma_{a_5}\ \langle (Doc\ \sigma_{a_3}\ \langle\ \rangle)\rangle)\ \gamma^t_{a_6}\ \langle\ \rangle \quad \diamond$$

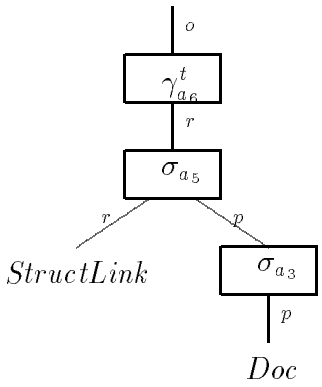


Figure 4.2: Mapping the query graph of Example 4.4 to an algebra operator tree.

Chapter 5

Type Consistency of Algebra Expressions

5.1 Introduction

Database query languages have traditionally had only minimal type checking requirements. In the relational model, for example, type checking insures that relation schemes are compatible and that only appropriate comparison operations are performed on tuple fields. The limited number of primitive domains supported by the model (e.g., integer, string, boolean) makes this a straightforward task. OODB query languages introduce complexity into this process as query results may be non-homogeneous sets of objects, i.e., all objects in the query result are not the same type[1].

Consider the case where the result of one query is used as the input to another. How can we insure that methods referenced in the predicate of the second query are defined on all objects in the result of the first? Previous algebras have imposed type restrictions such as *union compatibility* [SZ90, Zdo88] on the algebra operators to insure the type consistency of the result. Union compatibility states that members of the sets being operated on must be instances of types which are in a subtype relationship with one another. The type of the result is considered to be the most general supertype of the types involved in the operation. Such restrictions are too strong and can be avoided by a notion of type consistency which permits types to encompass a set of type specifications.

For example, what is the objection to taking the union of a set of *Apple* objects and a set of *Orange* objects? The result of the union is a set of non-homogeneous objects in the sense that they are not necessarily related by type inclusion. The real problem lies not with the union operation, but with operators which consume the result of this union and attempt to apply methods to each object therein.

Another problem, termed *impedance mismatch* [MSOP86], occurs when an appli-

[1] An object is an instance of a single class but supports the type specifications of this class and all its superclasses due to behavioral inheritance. Since this chapter investigates issues related to support of type specifications the term *type* is used throughout.

cation programming language must interface with a database query language. The two languages often have (partially) incompatible data types, e.g., union types in C and relations in SQL. A common requirement, independent of any particular language, is that a program variable be iteratively bound to each element in the set of objects returned by a query, e.g., portals [SR86] and cursors [Ast76]. Ideally, a compiler should insure that this binding is type consistent in order to detect improper use of data as early during query processing as possible. This problem becomes more complex when the query results are not homogeneous.

This chapter[2] proposes a type consistency theory for object algebra expressions with multiple types which resolves these issues. The theory is developed as a series of type inference rules. First the notion of a *conformance* as the set of type specifications an object supports is developed in Section 5.2 and an algorithm to derive conformances is presented. After defining a notation for the type inference rules, Section 5.3 develops rules for object algebra expressions in detail.

5.2 Types in Query Expressions

Black et. al. [BHJ+87] show how the conformity relationship is sufficient for developing a type checking algorithm for expressions denoting single objects and variable assignment. As demonstrated in the introduction, object-oriented database query languages introduce a new problem in that the result of a query is a set of objects which may not be homogeneous. In this case, what can be said about the types that each member of the query result supports?

Example 5.1 Consider the fragment of a type lattice in Figure 5.1 where types are labeled t_i. Assume we wish to take the union of the instances of types t_8 and t_9. The following can be said about the objects in $(ext(t_8) \cup ext(t_9))$.

1. Some objects conform to t_3 (immediate supertype of t_8),
2. Some objects conform to t_5 (immediate supertype of t_9),
3. All objects conform to t_4 (immediate supertype of both t_8 and t_9).

Intuitively then, we may say that the type of $(ext(t_8) \cup ext(t_9))$ is t_4 since this is the only type that all objects in the union conform to. This case is somewhat trivial as all objects in the query result conform to just one class. Referring again to Figure 5.1, assume we wish to take the union of the instances of types t_{10} and t_{11}. In this case the following can be said about the objects in $(ext(t_{10}) \cup ext(t_{11}))$.

1. Some objects conform to $\{t_3, t_4, t_2\}$ (immediate supertypes of t_{10}),
2. Some objects conform to $\{t_5, t_6, t_7\}$ (immediate supertypes of t_{11}),
3. All objects conform to $\{t_1, t_2\}$ (not necessarily immediate supertypes).

[2]Portions of this chapter have been published in the *Proc. of the Object-Oriented Programming Systems and Languages Conference*, October 1990.

The last statement holds because an object conforms to the type it is an instance of, and via inheritance, any of its supertypes. $\diamond$

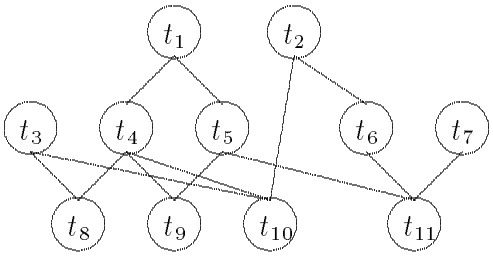


Figure 5.1: A type lattice fragment.

Definition 5.1 *Conformance:* A *conformance* is a set of types. A set of objects O has conformance $\{t_1, \ldots, t_n\}$, denoted by $O{:}\{t_1, \ldots, t_n\}$, when each object $o \in O$ conforms to every type $t_i \in \{t_1, \ldots, t_n\}$. $\square$

Definition 5.2 *Conformance Inclusion Relationship:* The conformance inclusion relationship on two sets of types C_1 and C_2 is defined as $C_1 \sqsubseteq C_2$ iff $\forall t_i \in C_2, \exists t_j \in C_1 \mid t_j \preceq t_i$. In other words, $C_1 \sqsubseteq C_2$, if for every type in C_2 there is a conforming type in C_1. $\square$

Note that C_1 may contain types which do not conform to any type in C_2 under this definition.

The notion of finding the set of types to which all members of a second set of types conforms to is central to determining the type consistency of operations on sets of objects. However, we do not always want to know all the types which are conformed to as this set would contain redundant information. In Example 5.1 the conformance of $(ext(t_{10}) \cup ext(t_{11}))$ was determined to be $\{t_1, t_2\}$. Including parents of t_1 and t_2 in the conformance would add no new type information since t_1 and t_2 define at least, if not more than, the behavior of their parents, i.e., t_1 and t_2 are specializations of their parent types. Similarly, placing more general types in the conformance, for example parents of t_1 and t_2 but not t_1 or t_2 themselves, introduces a loss of type information.

Loss of type information is undesirable when type checking a query. Consider again the type lattice fragment of Figure 5.1. Assume all objects in a query result conform to both t_{10} and t_{11} but the conformance was nonetheless specified as $\{t_1, t_2\}$. This would correspond to the case where types more general than necessary are placed into the conformance. It is possible that the query in question was just a subquery and that further operations are to be performed on its result. Some of the object algebra operators are qualified by predicates. One form of predicate involves applying a method to each member in the query set. If the method referenced in the query is defined on t_{11} but not on t_2, the query will fail during type checking when in fact each member of the query set does support that method. Thus we have the requirement that the conformance of a set of objects used in type checking include only the most specific types which satisfy the conformance definition.

Definition 5.3 *Most Specific Conformance:* The conformance of a set of objects O, $O: \{t_1, \ldots, t_n\}$, is defined to be the *most specific conformance* when there does not exist a subtype $s \preceq t_i$ such that all elements of O conform to s. $\Box$

The function $MSC(t_1, \ldots, t_n)$ is defined to return the most specific conformance of the types $t_1, \ldots, t_n$.

Example 5.2 Referring to Figure 5.1:

$$\begin{aligned} MSC(t_{10}) &\equiv \{t_{10}\} \\ MSC(t_{10}, t_{11}) &\equiv \{t_1, t_2\} \\ MSC(t_{10}, t_6) &\equiv \{t_2\} \ \diamond \end{aligned}$$

The need will arise during type checking to determine the inverse MSC relationship. Letting s and t refer to subtypes and types respectively, the function MSC^{-1} is defined as

$$MSC^{-1}(t_1, \ldots, t_n) \equiv \{ s_1, \ldots, s_k \mid MSC(s_1, \ldots, s_k) = \{t_1, \ldots, t_n\} \}$$

In other words, the inverse function MSC^{-1} returns the most general set of subtypes all of whom conform to $t_1, \ldots, t_n$.

Example 5.3 Referring to Figure 5.1:

$$\begin{aligned} MSC^{-1}(t_1) &\equiv \{t_1\} \\ MSC^{-1}(t_1, t_2) &\equiv \{t_{10}, t_{11}\} \\ MSC^{-1}(t_5, t_7) &\equiv \{t_{11}\} \ \diamond \end{aligned}$$

5.2.1 Determining the Most Specific Conformance

This section presents an algorithm for determining the most specific conformance of a set of types. The algorithm assumes the type lattice is represented as a *type dependency matrix*. The type dependency matrix for a type lattice with n types (including *Root*) is of order $n \times n$, i.e., there is a row and a column for each type in the lattice. Using $T(i, j)$ to denote the type dependency matrix, cells in T can be defined as:

$$T(i,j) = \begin{cases} 1 & \text{if } j \text{ is an immediate parent of } i \\ 0 & \text{otherwise} \end{cases}$$

The algorithm first applies Floyd's algorithm [Gou84] to determine the transitive closure of T, denoted T_c where:

$$T_c(i,j) = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Floyd's algorithm actually computes the matrix of shortest paths between all vertices of a graph but computes the transitive closure in the process.

The remainder of the algorithm performs bit-vector operations on the columns and rows of T_c to determine the MSC. Let $\vec{d}$ define the types of which the MSC is desired:

$$\vec{d}_j = \begin{cases} 1 & \text{if } MSC \text{ of type } t_j \text{ is desired} \\ 0 & \text{otherwise} \end{cases}$$

We define the common ancestor vector $\vec{c}$ where:

$$c_j = \begin{cases} 1 & \text{if } t_j \text{ is a common ancestor of all types denoted by } \vec{d} \\ 0 & \text{otherwise} \end{cases}$$

by the bitwise `AND`:

$$c_j = \bigwedge_{i=1}^{n} (T_c(i,j) \vee \neg d_j)$$

We next define the induced subgraph of T_c, called S as the subgraph of T_c which contains only the types which are common ancestors to all types in $\vec{d}$. By definition, the $MSC(\vec{d})$ is restricted to types in $\vec{c}$ and by extension, types in S. Observing that the "most specific" aspect of the types in the MSC can be interpreted as those types in the induced subgraph S which have no incoming edges allows us to derive the vector $\vec{m}$ where:

$$\vec{m}_j = \begin{cases} 1 & \text{if } t_j \text{ is in the } MSC \text{ of } \vec{d} \\ 0 & \text{otherwise} \end{cases}$$

by:

$$m_j = (\bigwedge_{i=1}^{n} \neg S_{ij}) \wedge c_j$$

In other words, the $MSC(\vec{d})$ includes only those types in the common ancestor subgraph of T_c which have no children.

Example 5.4 We demonstrate the above algorithm by deriving $MSC(t_{10}, t_{11})$ using the type lattice of Figure 5.1. For completeness, assume that types t_1, t_2, t_3 and t_7 are all immediate subtypes of *Root*. For simplicity, we will denote the *Root* type as t_0 and replace all iterations from $1..n$ in the algorithm to $0..n$. Vector $\vec{d}$ is [000000000011]. Floyd's algorithm is used to derive the transitive closure of the type dependency matrix resulting in

$$T_c = \begin{bmatrix} & t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10} & t_{11} \\ t_0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_4 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_5 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_6 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_8 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_9 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_{10} & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t_{11} & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The common ancestors vector $\vec{c}$ is determined by

$$\begin{aligned} c_j &= \bigwedge_{i=1}^{n} (T_c(i,j) \vee \neg d_j) \\ &= [111000000000] \end{aligned}$$

The induced subgraph S is:

$$S(i,j) = \begin{bmatrix} & t_0 & t_1 & t_2 \\ t_0 & 0 & 0 & 0 \\ t_1 & 1 & 0 & 0 \\ t_2 & 1 & 0 & 0 \end{bmatrix}$$

Applying the final step of the algorithm to determine $\vec{m}$ gives:

$$\begin{aligned} m_j &= (\bigwedge_{i=1}^{n} \neg S_{ij}) \wedge c_j \\ &= [011000000000] \\ &\equiv \{t_1, t_2\} \quad \diamond \end{aligned}$$

5.3 Type Inference Rules

The previous sections developed algorithms for determining the most specific conformance (MSC) of a set of types. This section formalizes the notion of typing in object algebra expressions by providing a set of type inference rules which utilize the MSC and MSC^{-1} functions. Although called type inference rules, they are really conformance inference rules since they determine a conformance for the result of an object algebra expression. The rules determine the conformance of an expression from the conformance(s) of its subexpressions. The rules themselves do not imply a specific

type checking mechanism. Instead, a type checking algorithm is considered correct if it computes types that are derivable by these rules. An expression is considered type inconsistent if the rules can not be used to derive a type (conformance) for all variables in the expression.

A syntax for inference rules similar to that of [CW85] will be used:

Rule Name:

$$\frac{X}{Y} \tag{5.1}$$

where the horizontal line is a logic implication. If we can infer X, then we can infer Y. Variables are used in a consistent fashion to denote similar items in each of the rules. Upper case variables denote sets while lower case variables denote single entities. For example, O, P, Q, and R are object set variables while o, p, q, and r are object variables with the implication that $o \in O$, $p \in P$, etc. I is a set of conformance inclusion constraints and A is a set of conformance assumptions for free variables. C is a conformance variable denoting a set of types. $A.e{:}C$ is the set A extended with the assumption that expression e has conformance C. $A \vdash expr$ is an assertion meaning that from A we can infer $expr$. Table 5.1 summarizes the variable denotations.

Table 5.1: Definition of variables used in the inference rules.

e	object algebra expression
f	predicate subexpression (i.e., lhs or rhs of an atom of the form $(lhs\ \theta\ rhs)$)
i, j, k	index variables and subscripts
m	method name variable
o, p, q, r	object variables
$F/[o_1{:}C_1, \ldots]$	predicate F is a set of atoms connected by $\wedge$ and/or $\vee$ with all occurrences of o_i having conformance C_i, etc.
t	type variable
A	set of conformance assumptions for free variables
C	conformance variable
I	set of conformance inclusion constraints
O, P, Q, R	object set variables

The following predefined functions are used by the typing rules.

unique(m, t, C): This boolean function evaluates to *True* if $t \in C$ and method m is defined only on type t and not on any other type in C.

arg_type(m, t, i): This function returns the declared type of the i^{th} argument of method m on type t.

num_args(m, t): This function returns an integer representing the number of arguments required by method m defined on type t.

res_type(m, t): This function returns the declared result type of method m on type t.

We present some conformance inclusion rules in order to familiarize the reader with the rule format.

Top:

$$I \vdash C \sqsubseteq \{Root\} \tag{5.2}$$

This rule states that every conformance is related via inclusion $\sqsubseteq$ to the set containing just the *Root* type. This can be derived from the conformance inclusion definition and recognizing that all types are a subtype of the *Root* type.

Transitivity:

$$\frac{I \vdash C_1 \sqsubseteq C_2, C_2 \sqsubseteq C_3}{I \vdash C_1 \sqsubseteq C_3} \tag{5.3}$$

Provable by definition of the $\sqsubseteq$ relationship.

Reflexivity:

$$I \vdash C \sqsubseteq C \tag{5.4}$$

Provable by definition of the $\sqsubseteq$ relationship and reflexivity of $\preceq$.

We are now ready to develop a family of type inference rules for the object algebra. Section 5.3.1 first introduces rules which can be applied to the predicates of select and generate operations. This is followed by Section 5.3.2 which develops rules for complete algebra expressions.

5.3.1 Predicate Inference Rules

Predicates qualify algebra operators and are composed of legal atoms connected by $\wedge$, $\vee$ and $\neg$. Each atom is of the form $(lhs\ \theta\ rhs)$ where lhs and rhs are either an object variable, a literal value, or a method application on object variables and $\theta \in \{=, ==, \in, =_{\{\}}\}$. The following predicate typing rules can be used to determine the type consistency of object variables when used as either the lhs or rhs of an atom.

Defining Set:

$$\frac{I, A \vdash O{:}C}{I, A \vdash o{:}C} \tag{5.5}$$

This rule states that an object variable conforms to the same types as the set from which it is drawn. This is a restatement of the implication given earler: $O{:}C \wedge o \in O \Rightarrow o{:}C$.

This rule is fundamental to the notion of type consistency. While arguments to algebra operators are sets of objects, predicates on algebra operators reference individual object variables. Since at query execution time the predicate is evaluated once for each object in the argument sets[3], the entire algebra operator is type consistent only if the predicate is type consistent for each type which exists in the argument sets, i.e., the conformance.

The next two rules determine the result type of a sequence of method applications. The first rule, for single methods only, insures that the types of the arguments to the method match those specified in its signature. The second rule recursively determines the result type of a sequence of method applications of length n by defining itself in

[3]Actually, the predicate is evaluated once for each element in the cross product of the argument sets.

terms of a sequence of methods of length $n-1$. When $(n-1)=1$, the recursion terminates and the first rule is applied.

Single Method:

$$\frac{I, A \vdash \left[\begin{array}{l} unique(m,t,C_1), \\ num\_args(m,t) = k-1, \\ C_i \sqsubseteq \{arg\_type(m,t,i)\}, 2 \leq i \leq k \end{array}\right.}{I, A \vdash <o_1 : C_1 \ldots o_k : C_k>.m : \{res\_type(m,t)\}} \tag{5.6}$$

This rule determines the conformance of the result of a single method application. The method application is legal if three conditions are met:

1. $unique(m,t,C_1)$ insures that there exists a type t which is a member of conformance C_1 and that method m is defined only on t and not on any other members of C_1. This restriction insures that there is no ambiguity as to which type's method m is to be applied.

2. $num\_args(m,t) = k-1$ insures that the signature of method m on type t requires the same number of parameters $(o_2 \ldots o_k)$ as are provided in the operand list of the method application.

3. $C_i \sqsubseteq \{arg\_type(m,t,i)\}, 2 \leq i \leq k$ insures that the types of the i^{th} argument, represented by the conformance C_i, are subtypes (and therefore substitutable) of the type stipulated by the signature of method m on type t.

If all of the above conditions are met, then the conformance of the result of the method application is the singleton set containing the result type as specified by the signature of m on t.

Multiple Methods:

$$\frac{I, A \vdash \left[\begin{array}{l} <o_1 : C_1 \ldots o_{j-1} : C_{j-1}>.m_1 \cdots m_{n-1} : \{t\}, \\ unique(m_n, t, \{t\}), \\ num\_args(m_n, t) = k-j+2, \\ C_i \sqsubseteq \{arg\_type(m_n,t,i)\}, j \leq i \leq k \end{array}\right.}{I, A \vdash <o_1 : C_1 \ldots o_j : C_j \ldots o_k : C_k>.m_1 \cdots m_n : \{res\_type(m_n,t)\}} \tag{5.7}$$

This recursive rule determines the conformance of a sequence of method applications based on the conformance of the sequence which applies methods m_1 through m_{n-1}. The multi-operation

$$<o_1 \ldots o_k>.m_1 \cdots m_n$$

is considered to be logically equivalent to the multi-operation

$$<o_1 \ldots o_j>.m_1 \cdots m_{n-1}$$

followed by the single operation

$$<res, o_{j+1} \ldots o_k>.m_n$$

where res is the result of the first multi-operation. The first condition of rule 5.7 stipulates that the conformance of the multi-operation which applies methods $m_1 \cdots m_{n-1}$ is known. This conformance is denoted as the singleton set $\{t\}$ since the signature of method m_{n-1} defines a single result type. The second condition insures that method m_n is actually defined on type t, an ancillary result of the *unique* function. The third condition guarantees that the proper number of arguments are present for method m_n while the last condition insures that each argument conforms to the types declared by m_n's signature.

If all of the above conditions are met, then the conformance of the result of the sequence of method applications $m_1 \cdots m_n$ is the set containing the result type of method m_n on type t.

Object Identity:

$$\frac{I, A \vdash f : C}{I, A \vdash o : C == f : C} \tag{5.8}$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o == f$ implies that o has conformance C as well. This should make sense intuitively based upon the meaning of the $==$ relationship. If two expressions are identity equal, i.e., denote the same object, then they conform to the same types.

Set Inclusion:

$$\frac{I, A \vdash f : C}{I, A \vdash o : \{Root\} \in f : C} \tag{5.9}$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o \in f$ implies that o conforms to the *Root* type. This apparent loss of type information is due to the fact that the data model does not include a parametrically polymorphic set type definition operator such as Set[t] [SZ90]. The data model supports only 'generic' set valued objects which make no restrictions on the type of the objects in the set.

Set Equivalence:

$$\frac{I, A \vdash f : C}{I, A \vdash o : \{Root\} =_{\{\}} f : C} \tag{5.10}$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o =_{\{\}} f$ implies that o conforms to the *Root* class. The reasoning is the same as in rule 5.9.

Atom Disjunction:

$$\frac{I, A \vdash F_1/[o : C_1], F_2/[o : C_2]}{I, A \vdash (F_1 \vee F_2)/[o : MSC(C_1 \cup C_2)]} \tag{5.11}$$

This rule states that if o has conformance C_1 for all occurrences in predicate F_1 and conformance C_2 for all occurrences in predicate F_2, then the conformance of o in the disjunction $(F_1 \vee F_2)$ is $MSC(C_1 \cup C_2)$. Consider that F_1 and F_2 both independently define a set of types for o. Their disjunction then implies that o represents objects which conform to types in C_1 **or** types in C_2. This is similar to the case of Example 5.1. The only statement one can make about all instances of o in the disjunction is that

they conform to the most specific common ancestors of types in C_1 and C_2 which is given by $MSC(C_1 \cup C_2)$.

Atom Conjunction:

$$\frac{I, A \vdash F_1/[o:C_1], F_2/[o:C_2]}{I, A \vdash (F_1 \wedge F_2)/[o:MSC^{-1}(C_1 \cup C_2)]} \tag{5.12}$$

This rule states that if o has conformance C_1 for all occurrences in predicate F_1 and conformance C_2 in all occurrences of predicate F_2, then the conformance of o in the conjunction $(F_1 \wedge F_2)$ is $MSC^{-1}(C_1 \cup C_2)$. The MSC^{-1} can be rationalized as follows. Consider that F_1 and F_2 both independently define a set of types for o. Their conjunction then implies that o represents objects which conform to types in C_1 **and** types in C_2. Clearly, only subtypes which inherit from all types in C_1 and C_2 can conform in this manner. $MSC^{-1}(C_1 \cup C_2)$ determines that set of types.

The previous two rules, atom disjunction and atom conjunction, are important results. They show that the manner in which atoms are combined in a predicate affects whether type information is lost (disjunction) or gained (conjunction). Type information can be lost in the case of disjunction since the inference rule derives a conformance for the variable in question which contains types which are more general, i.e., higher in the type lattice. Type information can be gained in the case of conjunction since the inference rule derives a conformance for the variable in question which contains types which are more specific, i.e., lower in the type lattice. An example illustrating this situation will be given in the following section.

5.3.2 Algebra Expression Inference Rules

This section presents the algebra operators and their associated typing rules. First some general inference rules for algebra expressions are needed.

Top:

$$I, A \vdash e:\{Root\} \tag{5.13}$$

This rule states that all object algebra expressions minimally conform to the type $\{Root\}$. This is clear since algebra expressions denote sets of objects and all objects conform to the *Root* type.

Transitivity:

$$\frac{I, A.e:C_1 \vdash C_1 \sqsubseteq C_2}{I, A \vdash e:C_2} \tag{5.14}$$

This rule is the same as the conformance inclusion transitivity rule applied to algebra expressions.

Basis:

$$I, A \vdash ext(t):\{t\} \tag{5.15}$$

Leaves of object algebra expression trees denote all instances of some type t in the database. This rule states that since all objects in $ext(t)$ conform to t, the conformance of a leaf node is $\{t\}$. This rule is called **Basis** since the only type information initially available in a query is the types it references explicitly. Just as query processing

proceeds from the leaves of the query tree to the root, one can think of type inference as proceeding from the leaves to the root as well.

Union:

$$\frac{I, A \vdash P : C_1, Q : C_2}{I, A \vdash (P \cup Q) : MSC(C_1 \cup C_2)} \tag{5.16}$$

This rule states that the conformance of a union operation is the MSC of types contained in the conformances of its operands. The reasoning is that $MSC(C_1 \cup C_2)$ denotes the most specific types to which **all** members of $(P \cup Q)$ conform.

Difference:

$$\frac{I, A \vdash P : C_1, Q : C_2}{I, A \vdash (P - Q) : C_1} \tag{5.17}$$

This rule states that the conformance of a difference operation is the conformance of the first operand. This should be clear as the result of a difference is a subset of the first operand.

Select:

$$\frac{I, A \vdash F/[p : C'_p, q_1 : C'_1 \ldots q_k : C'_k]}{I, A \vdash (P : C_p \ \sigma_F \ \langle Q_1 : C_1 \ldots Q_k : C_k \rangle) : C'_p} \tag{5.18}$$

Here F denotes the predicate of the select operation. The rule states that if the input sets $P, Q_1 \ldots Q_k$ have conformances $C_p, C_1, \ldots, C_k$ respectively, then the result of the select operation has conformance C'_p as derived for occurrences of p in predicate F. Since F may have multiple atoms connected by $\wedge$ and/or $\vee$, the atom conjunction and atom disjunction rules may determine conformances for variables in F (C'_p, C'_1, etc.) which are different from the conformances of the input argument sets (C_p, C_1, etc.). This allows for predicates which restrict or enhance the types of p.

Example 5.5 Consider the following query against the sample database of Figure 2.4.

> "Find all *TextObject*s whose keywords include "cycling" and which can be displayed in color."

An object algebra expression equivalent to this query is

$$TextObject^* \ \sigma \left[\begin{array}{c} \text{"}cycling\text{"} \in <t>.keywords \ \wedge \\ \text{"}True\text{"} = <d>.isColor \ \wedge \\ d == t \end{array} \right] \ \langle DisplayObject^* \rangle$$

where t ranges over *TextObject** and d ranges over *DisplayObject**. The type lattice of Figure 2.4 is simple enough that one can tell by inspection that the only displayable *TextObject*s are *Node*s. However, in a more complex schema, there could be many subtypes at various levels in the lattice which conform to both *TextObject* and *DisplayObject*. Thus it is appropriate to query over *TextObject** and *DisplayObject** and force membership in both sets via the atom $(d == t)$ rather than having the query range directly over the *Node* type.

We next show how the type inference rules could be used to derive a conformance of $\{Node\}$ for the query result. We intentionally say "could be used" since a specific

inference engine is not specified and the order in which rules will be applied is unknown. The only thing we can state (or require) of the type inference process is that it terminate when each occurrence of a variable has the same conformance associated with it.

We assume the predicate is initially annotated with type information reflecting the arguments to the query. *TO* and *DO* are used as a shorthand for *TextObject* and *DisplayObject* respectively.

$$\underbrace{\text{“}cycling\text{”} \in <t:\{TO\}>.keywords}_{a_1} \wedge \underbrace{\text{“}True\text{”} = <d:\{DO\}>.isColor}_{a_2} \wedge \underbrace{d:\{DO\} == t:\{TO\}}_{a_3}$$

Type inference proceeds by applying rules to derive new types for variables. We first apply object identity rule 5.8 to atom a_3 as follows

$$\frac{I, A \vdash t:\{TO\}}{I, A \vdash d:\{TO\} == t:\{TO\}}$$

resulting in the following annotated predicate.

$$\text{“}cycling\text{”} \in <t:\{TO\}>.keywords \wedge \text{“}True\text{”} = <d:\{DO\}>.isColor \wedge d:\{TO\} == t:\{TO\}$$

Next, atom conjunction rule 5.12 is applied as follows

$$\frac{I, A \vdash (a_1 \wedge a_2)/[d:\{DO\}], a_3/[d:\{TO\}]}{I, A \vdash ((a_1 \wedge a_2) \wedge a_3)/[d:MSC^{-1}(DO, TO)]}$$

resulting in the following annotated predicate.

$$\text{“}cycling\text{”} \in <t:\{TO\}>.keywords \quad \wedge \quad \text{“}True\text{”} = <d:\{Node\}>.isColor \wedge d:\{Node\} == t:\{TO\}$$

Note that $MSC^{-1}(DO, TO)$ would not just be the singleton set $\{Node\}$ if there were several types which inherited from both *TextObject* and *DisplayObject*. Object identity rule 5.8 is applied to atom a_3 again resulting in the following annotated predicate.

$$\text{“}cycling\text{”} \in <t:\{TO\}>.keywords \quad \wedge \quad \text{“}True\text{”} = <d:\{Node\}>.isColor \wedge d:\{Node\} == t:\{Node\}$$

Applying atom conjunction rule 5.12 again as follows

$$\frac{I, A \vdash (a_2 \wedge a_3)/[t:\{Node\}], a_1/[t:\{TO\}]}{I, A \vdash ((a_2 \wedge a_3) \wedge a_1)/[t:MSC^{-1}(Node, TO)]}$$

results in the annotated predicate

$$\text{“}cycling\text{”} \in <t:\{Node\}>.keywords \quad \wedge \quad \text{“}True\text{”} = <d:\{Node\}>.isColor \wedge d:\{Node\} == t:\{Node\}$$

in which all occurrences of a variable have the same conformance. The last step applies the rule for select expressions to determine the result of the overall query. Substituting into rule 5.18 gives

$$\frac{I, A \vdash (a_1 \wedge a_2 \wedge a_3)/[t:\{Node\}, d:\{Node\}]}{I, A \vdash (TextObject^*:\{TO\}\ \sigma_{(a_1 \wedge a_2 \wedge a_3)}\ \langle DisplayObject^*:\{DO\}\rangle) : \{Node\}}$$

Note that the nature of atom a_3 causes d and t to have the same conformance. Variables not related by the == operator will usually not have the same conformance. $\diamond$

The remaining two rules for the generate and map operators are similar to that for select.

Generate:

$$\frac{I, A \vdash F/[r:C'_r, q_1:C'_1 \ldots q_k:C'_k]}{I, A \vdash (Q_1:C_1\ \gamma^r_F\ \langle Q_2:C_2 \ldots Q_k:C_k\rangle) : C'_r} \tag{5.19}$$

Similar to the rule for select expressions, this rule states that the result of a generate operation has the same conformance as that derived for the result variable r in the predicate F, C'_r.

Map:

$$\frac{I, A \vdash <q_1:C_1 \ldots q_k:C_k>.mlist:\{t\}}{I, A \vdash (Q_1:C_1\ \mapsto_{mlist}\ \langle Q_2:C_2 \ldots Q_k:C_k\rangle) : \{t\}} \tag{5.20}$$

This rule states that the result of a map operation has the same conformance as that derived for the multi-operation $<q_1 \ldots q_k>.mlist$.

To summarize, the lack of homogeneity in the results of object algebra expressions may cause later expressions which use such a result to be type inconsistent. The conformance was defined as the set of types all members of a set of objects are guaranteed to conform to. An algorithm was developed to determine a most specific conformance. This provided the basis for a suite of type inference rules which determined the conformance of object algebra expression results based on the conformances of the argument sets.

Chapter 6

Rewriting Object Algebra Expressions

6.1 Introduction

This chapter introduces equivalence preserving transformations rules for object algebra expressions. The full suite of rules consists of *algebraic* and *semantic* rules. Algebraic rules create equivalent expressions based upon pattern matching and textual substitution. Semantic rules are similar, but they are additionally dependent on the semantics of the database schema as defined by the class definitions and inheritance lattice.

The overall goal of expression transformation is to reduce the cost of query evaluation. Haas et. al. [HFLP89] make the distinction between two rule based query transformation techniques: 'query rewrite' and 'plan optimization'. Query rewrite is a high level process where general purpose heuristics drive the application of transformation rules. Plan optimization is a lower level process which transforms a query into the most cost effective access plan based on a specific cost model and knowledge of access paths and database statistics. The rules presented here are intended for use during query rewrite. Plan optimization is discussed in Chapter 7.

The focus of the chapter is on rule specification as opposed to rule application. Recent work on rule based transformation systems [GD87, HFLP89, HP88, RH86] has shown their viability as a method for improving queries. However, the work has shown that although the search and transformation engine can be generalized, the rules themselves are specific to the data model. Thus, defining appropriate rules is a major challenge.

6.2 Transformation Rule Notation

Transformation rules will be written as $E_1 \Leftrightarrow E_2$ which specifies that expression E_1 is equivalent to expression E_2. Restricted rules, which are only applicable when condition c is true, are written as $E_1 \overset{c}{\Leftrightarrow} E_2$ [Fre87]. Conditions are a conjunction of functions which determine properties of argument sets, predicates and variables

used in a rule. Function $\mathrm{ref}(F,(v_1,\ldots,v_n))$ is true when $v_1,\ldots,v_n$ are the *only* variables referenced in the predicate F. Function $\mathrm{gen}(F,v)$ tests whether the predicate F contains a generating atom for the variable v. Similarly, $\mathrm{res}(F,v)$ is true when predicate F restricts values of v. For example, $\mathrm{gen}(F,t)$ is true in the case of $F \equiv (t \in <q_1,q_2>.mlist)$ and false in the case of $F \equiv (q_2 =_{\{\}} <t,q_1>.mlist)$. Furthermore, $\mathrm{res}(F,v) \Rightarrow \neg\ \mathrm{gen}(F,v)$.

Chapter 4 defined the object algebra select and generate operations as

$$P\ \sigma_{F_1}\ \langle Q_1 \ldots Q_k \rangle \equiv \{\ p \mid P(p) \wedge Q_1(q_1) \wedge \cdots \wedge Q_k(q_k) \wedge F_1(p,q_1,\ldots,q_k)\ \} \quad (6.1)$$

$$Q_1\ \gamma^p_{F_2}\ \langle Q_2 \ldots Q_k \rangle \equiv \{\ p \mid\ Q_1(q_1) \wedge \cdots \wedge Q_k(q_k) \wedge F_2(p,q_1,\ldots,q_k)\ \} \quad (6.2)$$

These two expressions are identical except that (6.2) does not define a universe of discourse for p thereby implying that F_2 in some way generates values for p from $q_1,\ldots,q_k$. To simplify proofs in later sections, the following definition will be used.

Definition 6.1 *Select/Generate Set Notation:*

$$\{\ p \mid \exists q_1 \cdots \exists q_k\ F(p,q_1,\ldots,q_k)\ \} \equiv \begin{cases} P\ \sigma_F\ \langle Q_1 \ldots Q_k \rangle & \text{iff } \mathrm{res}(F,p) \\ Q_1\ \gamma^p_F\ \langle Q_2 \ldots Q_k \rangle & \text{iff } \mathrm{gen}(F,p) \end{cases}$$

In other words, the set definition for select and generate operations can only be distinguished by the properties of predicate F. If F is defined as restricting values of p, then the operation is a select. If F is defined as generating values for p, then the operation is a generate. □

An arbitrary expression in a list of expressions is referenced using the notation $(E_1 \ldots E_i \ldots E_n)$ where '...' denotes zero or more occurrences of some E_i. For example, the rule

$$P\ \sigma_F\ \langle \ldots Q_x \ldots Q_y \ldots \rangle \quad \Leftrightarrow \quad P\ \sigma_F\ \langle \ldots Q_y \ldots Q_x \ldots \rangle \quad (6.3)$$

indicates that the result of a select operation is independent of the ordering of the arguments between '⟨' and '⟩'. The set being restricted must appear before the select operator σ_F, thus there is no rule to change the position of P. This is not the case for generate operations as the target variable does not correspond to one of the input sets. The next two rules state that the outcome of a generate is independent of the operand ordering.

$$Q_x\ \gamma^t_F\ \langle \ldots Q_y \ldots \rangle \quad \Leftrightarrow \quad Q_y\ \gamma^t_F\ \langle \ldots Q_x \ldots \rangle \quad (6.4)$$

$$Q_x\ \gamma^t_F\ \langle \ldots Q_y \ldots Q_z \ldots \rangle \quad \Leftrightarrow \quad Q_x\ \gamma^t_F\ \langle \ldots Q_z \ldots Q_y \ldots \rangle \quad (6.5)$$

Section 3.1 introduced the map operator as a special case of generate. This can be captured by the conditional rule:

$$Q_1\ \gamma^t_F\ \langle Q_2 \ldots Q_k \rangle \quad \overset{c}{\Leftrightarrow} \quad Q_1\ \mapsto_{mlist}\ \langle Q_2 \ldots Q_k \rangle \quad (6.6)$$

where condition c insures that $F \equiv (t == <q_1,\ldots,q_k>.mlist)$.

We introduce the abbreviations $Qset$, $Rset$ and $Sset$ to replace $Q_1 \ldots Q_k$, $R_1 \ldots R_l$ and $S_1 \ldots S_m$ respectively. For example:

$$P\ \sigma_F\ \langle Qset, Rset, Sset\rangle \ \stackrel{c}{\Leftrightarrow}\ P\ \sigma_F\ \langle Q_1 \ldots Q_k, R_1 \ldots R_l, S_1 \ldots S_m\rangle \qquad (6.7)$$

where condition c is $\mathrm{ref}(F, (p, q_1, \ldots, q_k, r_1, \ldots, r_l, s_1, \ldots, s_m))$. As before, a lower case letter represents an object variable which ranges over the set denoted by the corresponding upper case letter, (i.e., $q_i \in Q_i$, $r_i \in R_i$ and $s_i \in S_i$).

6.3 Identities

This section presents rules which are identities in the object algebra, i.e., there are no conditions associated with them.

Theorem 6.1 *Set Identities:*
The following equations summarize the rewrite rules for binary set operators. These are conventional set operators that can be found in any set theory textbook (see, for example, [SM77]).

$$\begin{aligned}
P \cup (Q \cup R) &\Leftrightarrow (P \cup Q) \cup R && (6.8)\\
P \cup (Q \cap R) &\Leftrightarrow (P \cup Q) \cap (P \cup R) && (6.9)\\
P - (Q \cup R) &\Leftrightarrow (P - Q) \cap (P - R) && (6.10)\\
P - (Q \cap R) &\Leftrightarrow (P - Q) \cup (P - R) && (6.11)\\
P \cap (Q \cup R) &\Leftrightarrow (P \cap Q) \cup (P \cap R) && (6.12)\\
P \cap (Q \cap R) &\Leftrightarrow (P \cap Q) \cap R && (6.13)\\
P - (Q - R) &\Leftrightarrow (P - Q) \cup (P \cap Q \cap R) && (6.14)\\
(P - Q) - R &\Leftrightarrow (P - Q) \cap (P - R) && (6.15)\\
(P \cup Q) - R &\Leftrightarrow (P - R) \cup (Q - R) && (6.16)\\
(P \cap Q) - R &\Leftrightarrow (P - R) \cap (Q - R) && (6.17)
\end{aligned}$$

Proof: Equations (6.8) through (6.17) are valid as P, Q and R denote sets of objects where set membership is determined by object identity and the algebra operations $-$, $\cup$ and $\cap$ operate on object identities. □

Theorem 6.2 *Commutativity of Select:*

$$(P\ \sigma_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \ \Leftrightarrow\ (P\ \sigma_{F_2}\ \langle Rset\rangle)\ \sigma_{F_1}\ \langle Qset\rangle \qquad (6.18)$$

Proof: The result of $P\ \sigma_{F_1}\ \langle Qset\rangle$ is the subset of P for which $F_1(p, q_1, \ldots, q_k)$ is true. Call this result P'. Then $P'\ \sigma_{F_2}\ \langle Rset\rangle$ represents the LHS of (6.18) and is the subset of P' which satisfies $F_2(p', r_1, \ldots, r_l)$. Since $P' \subseteq P$ the final result contains those elements of P which satisfy both $F_1(p, q_1, \ldots, q_k)$ and $F_2(p, r_1, \ldots, r_l)$ independent of the order in which they are obtained. □

Theorem 6.3 *Commutativity of Difference with respect to Select:*

$$(P - Q)\ \sigma_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset\rangle) - Q \tag{6.19}$$

Proof: The result of $(P - Q)\ \sigma_F\ \langle Rset\rangle$ does not contain any members of Q since the selection is applied to $(P - Q)$ which is disjoint with Q. Thus the selection criterion can be applied to P alone and members of Q removed afterwards. □

Theorem 6.4 *Distributivity of Union with respect to Select – A:*

$$(P \cup Q)\ \sigma_F\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset\rangle) \cup (Q\ \sigma_F\ \langle Rset\rangle) \tag{6.20}$$

Proof: Let $(P \cup Q)\ \sigma_F\ \langle Rset\rangle$ be written as

$$(P \cup Q) - X \tag{6.21}$$

where X represents those elements of $(P \cup Q)$ which do not satisfy F and each $x \in X$ is a member of P or Q or both. Replace X by $X_P \cup X_Q$ where $\forall x \in X_P, x \in P$ and $\forall x \in X_Q, x \in Q$. Substituting into 6.21 and rewriting according to 6.16 gives

$$(P - (X_P \cup X_Q)) \cup (Q - (X_P \cup X_Q)) \tag{6.22}$$

Each element in X_Q is either in Q alone or in $(Q \cap P)$ in which case it is also in X_P. Thus $(P - (X_P \cup X_Q)) = (P - X_P)$ since those elements which are in Q alone do not contribute to the difference. By similar argument, $(Q - (X_P \cup X_Q)) = (Q - X_Q)$. Applying these identities to (6.22) gives

$$(P - X_P) \cup (Q - X_Q) \tag{6.23}$$

Since X_P (X_Q) represents those members of P (Q) which do not satisfy predicate F, 6.23 is equivalent to the right hand side of 6.20. □

Theorem 6.5 *Distributivity of Union with respect to Select – B:*

$$\begin{aligned} &P\ \sigma_F\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k\rangle \\ &\quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Q_1 \ldots Q_x \ldots Q_k\rangle)\ \cup\ (P\ \sigma_F\ \langle Q_1 \ldots Q_y \ldots Q_k\rangle) \end{aligned} \tag{6.24}$$

Proof: Let $q_{xy} \in (Q_x \cup Q_y)$. Then the LHS of (6.24) returns the p components of each vector $<p, q_1, \ldots, q_{xy}, \ldots, q_k>$ in $P \times Q_1 \times \cdots \times (Q_x \cup Q_y) \times \cdots \times Q_k$ which satisfies $F(p, q_1, \ldots, q_{xy}, \ldots, q_k)$. Distributing union across cartesian product gives

$$\begin{aligned} &P \times Q_1 \times \cdots \times (Q_x \cup Q_y) \times \cdots \times Q_k \ \Leftrightarrow \\ &\quad (P \times Q_1 \times \cdots \times Q_x \times \cdots \times Q_k) \cup (P \times Q_1 \times \cdots \times Q_y \times \cdots \times Q_k) \end{aligned}$$

Thus the LHS of (6.24) is the union of p components of each vector $<p, q_1, \ldots, q_x, \ldots, q_k>$ in $P \times Q_1 \times \cdots \times Q_x \times \cdots \times Q_k$ which satisfies $F(p, q_1, \ldots, q_x, \ldots, q_k)$ and p components of each vector $p, q_1, \ldots, q_y, \ldots, q_k$ in $P \times Q_1 \times \cdots \times Q_y \times \cdots \times Q_k$ which satisfies $F(p, q_1, \ldots, q_y, \ldots, q_k)$ which is equivalent in meaning to the RHS of (6.24). □

Theorem 6.6 *Distributivity of Intersect with respect to Select:*

$$(P \cap Q)\ \sigma_F\ \langle Rset \rangle \quad \Leftrightarrow \quad (P\ \sigma_F\ \langle Rset \rangle) \cap (Q\ \sigma_F\ \langle Rset \rangle) \tag{6.25}$$

Proof: Let $(P \cap Q)\ \sigma_F\ \langle Rset \rangle$ be written as

$$(P \cap Q) - X \tag{6.26}$$

where X represents those elements of $(P \cap Q)$ which do not satisfy predicate F and each $x \in X$ is a member of both P and Q. By the definition of intersection, $X = X_P = X_Q$ where $\forall x \in X_P, x \in P$ and $\forall x \in X_Q, x \in Q$. Rewriting (6.26) using (6.17) gives

$$(P - (X_P \cap X_Q)) \cup (Q - (X_P \cap X_Q)) \tag{6.27}$$

Substituting X_P for the first occurrence of X and X_Q for the second occurrence of X gives

$$(P - X_P) \cap (Q - X_Q) \tag{6.28}$$

Since X_P (X_Q) represents those members of P (Q) which do not satisfy the qualifying formula F, 6.28 is equivalent to the right hand side of 6.25. □

Theorem 6.7 *Distributivity of Union with respect to Generate – A:*

$$(P \cup Q)\ \gamma_F^t\ \langle Rset \rangle \quad \Leftrightarrow \quad (P\ \gamma_F^t\ \langle Rset \rangle) \cup (Q\ \gamma_F^t\ \langle Rset \rangle) \tag{6.29}$$

Proof: Predicate F in the LHS of (6.29) generates an object denoted by t for each vector $<(pq), r_1, \ldots, r_l>$ in $(P \cup Q) \times R_1 \times \cdots \times R_l$. Distributing union across cartesian product gives

$$(P \cup Q) \times R_1 \times \cdots \times R_l \quad \Leftrightarrow \quad (P \times R_1 \times \cdots \times R_l) \cup (Q \times R_1 \times \cdots \times R_l) \tag{6.30}$$

Thus the LHS of (6.29) is the union of objects denoted by t when predicate F is evaluated for each vector $<p, r_1, \ldots, r_l>$ in $P \times R_1 \times \cdots \times R_l$ and each vector $<q, r_1, \ldots, r_l>$ in $Q \times R_1 \times \cdots \times R_l$ which is equivalent in meaning to the RHS of (6.29). □

Theorem 6.8 *Distributivity of Union with respect to Generate – B:*

$$\begin{aligned} & P\ \gamma_F^t\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \\ & \quad \Leftrightarrow \quad (P\ \gamma_F^t\ \langle Q_1 \ldots Q_x \ldots Q_k \rangle)\ \cup\ (P\ \gamma_F^t\ \langle Q_1 \ldots Q_y \ldots Q_k \rangle) \end{aligned} \tag{6.31}$$

Proof: Applying identity (6.5) to the LHS of (6.31) gives

$$P\ \gamma_F^t\ \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \quad \Leftrightarrow \quad (Q_x \cup Q_y)\ \gamma_F^t\ \langle Q_1 \ldots P \ldots Q_k \rangle \tag{6.32}$$

The remainder of the proof follows that of Theorem 6.7. □

Theorem 6.9 *Distributivity of Union with respect to Map:*

$$(P \cup Q) \;\mapsto_{mlist}\; \langle Rset \rangle \;\Leftrightarrow\; (P \;\mapsto_{mlist}\; \langle Rset \rangle) \cup (Q \;\mapsto_{mlist}\; \langle Rset \rangle) \tag{6.33}$$

$$P \;\mapsto_{mlist}\; \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \;\Leftrightarrow\; (P \;\mapsto_{mlist}\; \langle Q_1 \ldots Q_x \ldots Q_k \rangle) \cup (P \;\mapsto_{mlist}\; \langle Q_1 \ldots Q_y \ldots Q_k \rangle) \tag{6.34}$$

Proof: Identity (6.6) can be used to rewrite the left hand side of (6.33) and (6.34) as the following equivalent generate operations.

$$(P \cup Q) \;\mapsto_{mlist}\; \langle Rset \rangle \;\Rightarrow\; (P \cup Q)\; \gamma_F^t\; \langle Rset \rangle$$

$$P \;\mapsto_{mlist}\; \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle \;\Leftrightarrow\; P\; \gamma_F^t\; \langle Q_1 \ldots (Q_x \cup Q_y) \ldots Q_k \rangle$$

The remainder of the proof follows that of Theorems 6.7 and 6.8. □

Example 6.1 Consider the query on the sample database of Chapter 2 "Return the root nodes of all documents which are either about cats or about dogs". Let

d	range over the class *Document*
F_1	be the atom ("*cats*" ∈<*d*>.*keyWords*)
F_2	be the atom ("*dogs*" ∈<*d*>.*keyWords*)
n	range over *Node* objects

Then we can use the following object algebra expression to implement the query

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle)) \;\mapsto_{rootNode}\; \langle\ \rangle$$

and apply rule 6.33 to get

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle) \;\mapsto_{rootNode}\; \langle\ \rangle) \cup ((Doc\ \sigma_{F_2}\ \langle\ \rangle) \;\mapsto_{rootNode}\; \langle\ \rangle)$$

The transformation is shown graphically on the right hand side of Figure 6.1. ◇

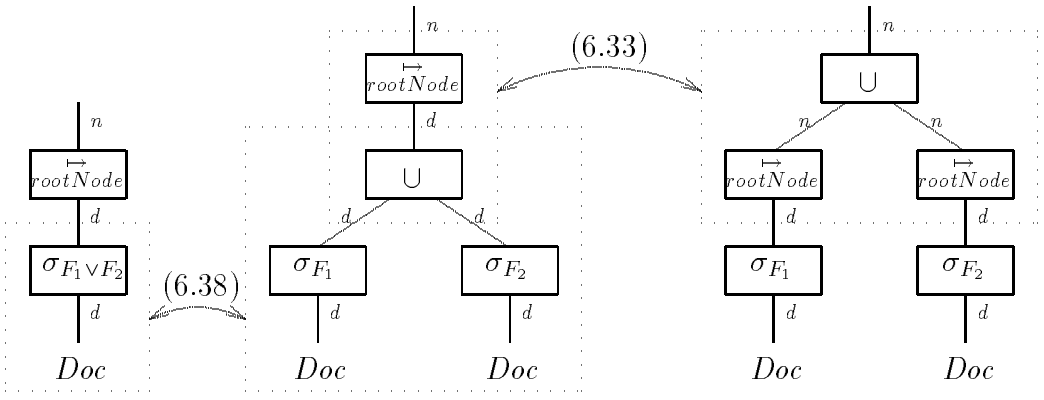


Figure 6.1: Transformations of examples 6.1 and 6.2.

6.4 Select Rewrite Rules

Theorem 6.10 *Factorization of Cascaded Selects:*

$$(P\ \sigma_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \quad \Leftrightarrow \quad (P\ \sigma_{F_1}\ \langle Qset\rangle)\ \cap\ (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{6.35}$$

Proof: Let P' represent $P\ \sigma_{F_1}\ \langle Qset\rangle$, i.e., the subset of P which satisfies $F_1(p, q_1, \ldots, q_k)$. Then $P\ \sigma_{F_2}\ \langle Rset\rangle$ is the subset of P' which satisfies $F_2(p', r_1, \ldots, r_l)$. Since $P' \subseteq P$, the final result contains those elements of P which satisfy both $F_1(p, q_1, \ldots, q_k)$ and $F_2(p, r_1, \ldots, r_l)$. Thus the LHS of (6.35) can be rewritten as

$$\{\ p \mid F_1(p, q_1, \ldots, q_k)\ \}\quad \cap \quad \{\ p \mid F_2(p, q_1, \ldots, q_k)\ \}$$

which is equivalent in meaning to the RHS of (6.35). □

Theorem 6.11 *Conjunctive Select Predicate – A:*

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, Rset\rangle \quad \overset{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset\rangle) \cap (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{6.36}$$

where:

$$c:\ \mathrm{ref}(F_1, (p, q_1 \ldots q_k))\ \wedge\ \mathrm{res}(F_1, p)\ \wedge\ \mathrm{ref}(F_2, (p, r_1 \ldots r_l))\ \wedge\ \mathrm{res}(F_2, p)$$

Proof: Taking into account condition c, the LHS of (6.36) can be written as

$$\{\ p \mid \exists q_1 \cdots \exists q_k \exists r_1 \cdots \exists r_l\ (F_1(p, q_1, \ldots, q_k)\ \wedge\ F_2(p, r_1, \ldots, r_l))\ \}$$

Since F_1 does not reference $r_1, \ldots, r_l$ and F_2 does not reference $q_1, \ldots, q_k$, the existential quantifiers can be distributed across the conjunction as follows

$$\{\ p \mid \exists q_1 \cdots \exists q_k\ F_1(p, q_1, \ldots, q_k)\ \wedge\ \exists r_1 \cdots \exists r_l\ F_2(p, r_1, \ldots, r_l)\ \}$$

This can be simplified to

$$\{\ p \mid F'_1(p)\ \wedge\ F'_2(p)\ \} \tag{6.37}$$

which is also the definition for set intersection. Thus it can be written as

$$\{\ p \mid F'_1(p)\ \} \cap \{\ p \mid F'_2(p)\ \}$$

Expanding F'_1 and F'_2 again gives

$$\{\ p \mid \exists q_1 \cdots \exists q_k\ F_1(p, q_1, \ldots, q_k)\ \} \cap \{\ p \mid \exists r_1 \cdots \exists r_l\ F_2(p, r_1, \ldots, r_l)\ \}$$

which is equivalent in meaning to the RHS of (6.36). □

Theorem 6.12 *Disjunctive Select Predicates:*

$$P\ \sigma_{(F_1 \vee F_2)}\ \langle Qset, Rset\rangle \quad \overset{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset\rangle) \cup (P\ \sigma_{F_2}\ \langle Rset\rangle) \tag{6.38}$$

where:

$$c:\ \mathrm{ref}(F_1,(p,q_1\ldots q_k))\ \wedge\ \mathrm{res}(F_1,p)\ \wedge\ \mathrm{ref}(F_2,(p,r_1\ldots r_l))\ \wedge\ \mathrm{res}(F_2,p)$$

Proof: The proof mimics that of Theorem 6.11 except that equation (6.37) now becomes

$$\{\ p \mid F'_1(p)\ \vee\ F'_2(p)\ \} \tag{6.39}$$

which is the definition for set union. □

Example 6.2 The union subquery $((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle))$ of Example 6.1 matches the right hand side of rule 6.38. Substituting Doc for P and $\langle\ \rangle$ for $Qset$ and $Rset$ we can apply rule 6.38 right to left resulting in

$$((Doc\ \sigma_{F_1}\ \langle\ \rangle) \cup (Doc\ \sigma_{F_2}\ \langle\ \rangle)) \quad \Leftrightarrow \quad Doc\ \sigma_{(F_1 \vee F_2)}\ \langle\ \rangle$$

This transformation is shown graphically on the left hand side of Figure 6.1. ◇

Theorem 6.13 *Conjunctive Select Predicate – B:*

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset\rangle \overset{c_1}{\Leftrightarrow} P\ \sigma_{F_1}\ \langle Qset, (R\ \sigma_{F_2}\ \langle Sset\rangle)\rangle \tag{6.40}$$
$$\overset{c_2}{\Leftrightarrow} P\ \sigma_{F_1}\ \langle Qset, (R\ \gamma^t_{F_2}\ \langle Sset\rangle)\rangle \tag{6.41}$$

where:

$$c_1:\ \mathrm{ref}(F_1,(p,q_1\ldots q_k,r))\ \wedge\ \mathrm{ref}(F_2,(r,s_1\ldots s_m))\ \wedge\ \mathrm{res}(F_2,r)$$
$$c_2:\ \mathrm{ref}(F_1,(p,q_1\ldots q_k,t))\ \wedge\ \mathrm{ref}(F_2,(t,r,s_1\ldots s_m))\ \wedge\ \mathrm{gen}(F_2,t)$$

Proof: Taking into account condition c_1, the LHS of (6.40) can be written as

$$\{\ p \mid \exists q_1 \cdots \exists q_k, \exists r \exists s_1 \cdots \exists s_m\ (F_1(p,q_1,\ldots,q_k,r)\ \wedge\ F_2(r,s_1,\ldots,s_m,))\ \}$$

Since F_1 does not reference $s_1,\ldots,s_m$ and F_2 does not reference $q_1,\ldots,q_k$, the existential quantifiers can be distributed across the conjunction as follows

$$\{\ p \mid \exists r\ (\ \exists q_1 \cdots \exists q_k\ F_1(p,q_1,\ldots,q_k,r)\ \wedge\ \exists s_1 \cdots \exists s_m\ F_2(r,s_1,\ldots,s_m)\ )\ \}$$

This expression can be rewritten as

$$\{\ p \mid \exists r' \exists q_1 \cdots \exists q_k\ F_1(p,q_1,\ldots,q_k,r')\ \} \tag{6.42}$$

where the universe of discourse for r' is defined by

$$R' \ \equiv\ \{\ r \mid \exists s_1 \cdots \exists s_m\ F_2(r,s_1,\ldots,s_m)\ \} \tag{6.43}$$

Equation (6.42) is equivalent in meaning to $P\ \sigma_{F_1}\ \langle Qset, R'\rangle$ while R' of equation (6.43) is equivalent in meaning to $R\ \sigma_{F_2}\ \langle Sset\rangle$. Together these two expressions are equivalent to the RHS of (6.40).

The proof for (6.41) mimics the previous one with the exception that (6.42) becomes

$$\{\ p \mid \exists t' \exists q_1 \cdots \exists q_k\ F_1(p,q_1,\ldots,q_k,t')\ \} \tag{6.44}$$

where the universe of discourse for t' is defined by

$$T' \ \equiv\ \{\ t \mid \exists r \exists s_1 \cdots \exists s_m\ F_2(r,s_1,\ldots,s_m,t)\ \}\ \ \square \tag{6.45}$$

Example 6.3 Consider the query "Find all documents written by the child of a computer scientist and a doctor". Let

d	range over the class *Document*
p_1	range over the class *Person*
p_2	range over the class *Person*
c	range over *Person* objects which are children
a_1	be the atom $(c == <d>.author)$
a_2	be the atom $(\text{"}computers\text{"} \in <p_1>.expertise)$
a_3	be the atom $(\text{"}medicine\text{"} \in <p_2>.expertise)$
a_4	be the atom $(c \in <p_1, p_2>.children)$

The following object algebra expression can be used to represent the query

$$Doc\ \sigma_{(a_1 \wedge a_2 \wedge a_3 \wedge a_4)}\ \langle Person, Person \rangle$$

This expression satisfies the conditions of rule 6.41 when we substitute a_1 for F_1, $(a_2 \wedge a_3 \wedge a_4)$ for F_2, *Doc* for P, { } for $Qset$, *Person* for R and *Person* for $Sset$. Applying rule 6.41 with these substitutions gives

$$Doc\ \sigma_{(a_1 \wedge a_2 \wedge a_3 \wedge a_4)}\ \langle Person, Person \rangle \quad \Leftrightarrow \quad Doc\ \sigma_{a_1}\ \langle (Person\ \gamma^{c}_{(a_1 \wedge a_2 \wedge a_3)}\ \langle Person \rangle) \rangle$$

This transformation is shown graphically on the left hand side of Figure 6.2. $\diamond$

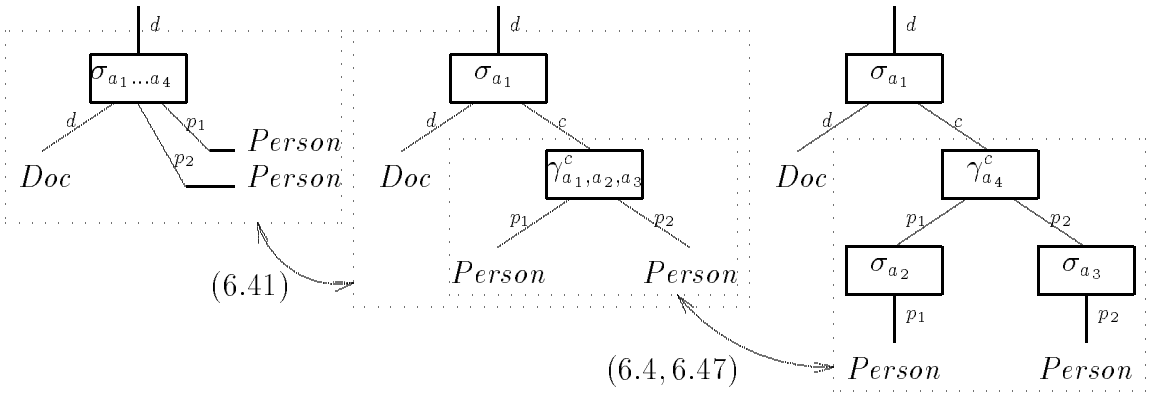


Figure 6.2: Transformations of examples 6.3 and 6.4.

Theorem 6.14 *Factoring Generate from a Conjunctive Select:*

A special case of rule 6.41 occurs when the select predicate contains a generating atom for the set being restricted (Figure 6.3).

$$P\ \sigma_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset \rangle \quad \overset{c}{\Leftrightarrow} \quad (P\ \sigma_{F_1}\ \langle Qset \rangle) \cap (R\ \gamma^{p}_{F_2}\ \langle Sset \rangle) \qquad (6.46)$$

where:

$$c: \ \mathrm{ref}(F_1,(p,q_1 \ldots q_k)) \ \wedge \ \mathrm{res}(F_1,p) \ \wedge \ \mathrm{ref}(F_2,(p,r,s_1 \ldots s_m)) \ \wedge \ \mathrm{gen}(F_2,p)$$

Using the same techniques as in the previous proofs, (6.46) can written as

$$\{\ p \mid \exists q_1 \cdots \exists q_k \ F_1(p,q_1,\ldots,q_k)\ \} \cap \{\ p \mid \exists r \exists s_1 \cdots \exists s_m \ F_2(p,r,s_1,\ldots,s_m)\ \}$$

Applying Definition 6.1 and condition c, this expression is equivalent in meaning to the RHS of (6.46). □

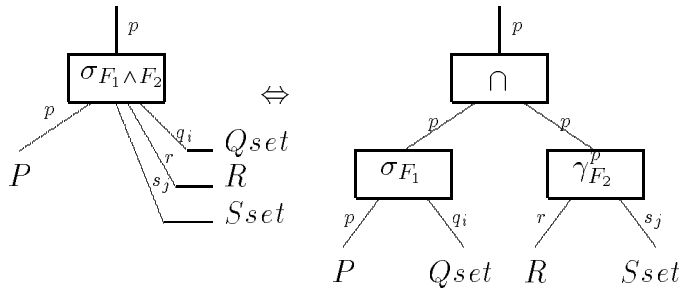


Figure 6.3: Graphical representation of rewrite rule 6.46.

6.5 Generate Rewrite Rules

Theorem 6.15 *Conjunctive Generate Predicates:*

$$P\ \gamma^t_{(F_1 \wedge F_2)}\ \langle Qset, Rset\rangle \overset{c_1}{\Leftrightarrow} (P\ \sigma_{F_1}\ \langle Qset\rangle)\ \gamma^t_{F_2}\ \langle Rset\rangle \tag{6.47}$$

$$\overset{c_2}{\Leftrightarrow} (P\ \gamma^u_{F_1}\ \langle Qset\rangle)\ \gamma^t_{F_2}\ \langle Rset\rangle \tag{6.48}$$

$$\overset{c_3}{\Leftrightarrow} (P\ \gamma^t_{F_1}\ \langle Qset\rangle)\ \sigma_{F_2}\ \langle Rset\rangle \tag{6.49}$$

where:

$$c_1: \ \mathrm{ref}(F_1,(p,q_1 \ldots q_k)) \ \wedge \ \mathrm{res}(F_1,p) \ \wedge \ \mathrm{ref}(F_2,(p,r_1 \ldots r_l,t)) \ \wedge \ \mathrm{gen}(F_2,t)$$
$$c_2: \ \mathrm{ref}(F_1,(p,q_1 \ldots q_k,u)) \ \wedge \ \mathrm{gen}(F_1,u) \ \wedge \ \mathrm{ref}(F_2,(r_1 \ldots r_l,t,u)) \ \wedge \ \mathrm{gen}(F_2,t)$$
$$c_3: \ \mathrm{ref}(F_1,(p,q_1 \ldots q_k,t)) \ \wedge \ \mathrm{gen}(F_1,t) \ \wedge \ \mathrm{ref}(F_2,(r_1 \ldots r_l,t) \ \wedge \ \mathrm{res}(F_2,t)$$

Proof of (6.47):

$$\{\ t \mid \exists p(\exists q_1 \cdots \exists q_k \ F_1(p,q_1,\ldots,q_k) \ \wedge \ \exists r_1 \cdots \exists r_l \ F_2(p,r_1,\ldots,r_l,t))\ \}$$

$$\{\ t \mid P(p) \ \wedge \ \exists q_1 \cdots \exists q_k \ F_1(p,q_1,\ldots,q_k) \ \wedge \ \exists r_1 \cdots \exists r_l \ F_2(p,r_1,\ldots,r_l,t)\ \}$$

Grouping the first two terms and changing $P(p)$ to a quantifier yields

$$\{\ t \mid \exists p \exists q_1 \cdots \exists q_k \ F_1(p,q_1,\ldots,q_k) \ \wedge \ \exists r_1 \cdots \exists r_l \ F_2(p,r_1,\ldots,r_l,t)\ \}$$

Substituting $P'(p)$ for the first term gives

$$\{\ t \mid P'(p') \ \wedge \ \exists r_1 \cdots \exists r_l \ F_2(p',r_1,\ldots,r_l,t)\ \} \tag{6.50}$$

where

$$P' \equiv \{ p \mid \exists p \exists q_1 \cdots \exists q_k \; F_1(p, q_1, \ldots, q_k) \} \tag{6.51}$$

Equation (6.50) is equivalent to $P' \; \gamma^t_{F_2} \; \langle Rset \rangle$ while equation (6.51) is equivalent to $P \; \sigma_{F_1} \; \langle Qset \rangle$. These two expressions can be combined to yield the RHS of (6.47).
Proof of (6.48):

$$\{ t \mid \exists p \exists q_1 \cdots \exists q_k \; F_1(p, q_1, \ldots, q_k, u) \; \wedge \; \exists r_1 \cdots \exists r_l \; F_2(r_1, \ldots, r_l, t, u) \}$$

Substituting $U'(u')$ for the first term yields

$$\{ t \mid U'(u') \; \wedge \; \exists r_1 \cdots \exists r_l \; F_2(r_1, \ldots, r_l, t, u') \} \tag{6.52}$$

where

$$U' \equiv \{ u \mid \exists p \exists q_1 \cdots \exists q_k \; F_1(p, q_1, \ldots, q_k, u) \} \tag{6.53}$$

Equation (6.52) is equivalent to $U' \; \gamma^t_{F_2} \; \langle Rset \rangle$ while equation (6.53) is equivalent tp $P \; \gamma^u_{F_1} \; \langle Qset \rangle$. Together these are equivalent to the RHS of (6.48).
Proof of (6.56):

$$\{ t \mid \exists p \exists q_1 \cdots \exists q_k \; F_1(p, q_1, \ldots, q_k, t) \; \wedge \; \exists r_1 \cdots \exists r_l \; F_2(r_1, \ldots, r_l, t) \}$$

Substituting $T'(t')$ for the first term gives

$$\{ t \mid T'(t') \; \wedge \; \exists r_1 \cdots \exists r_l \; F_2(r_1, \ldots, r_l, t') \} \tag{6.54}$$

where

$$T' \equiv \{ t \mid \exists p \exists q_1 \cdots \exists q_k \; F_1(p, q_1, \ldots, q_k, t) \} \tag{6.55}$$

Equation (6.54) is equivalent to $T' \; \sigma_{F_2} \; \langle Rset \rangle$ equation (6.55) is equivalent to $P \; \gamma^t_{F_1} \; \langle Qset \rangle$. These two expressions can be combined to yield the RHS of (6.56). □

Example 6.4 Consider the subquery $Person \; \gamma^c_{a_2 \wedge a_3 \wedge a_4} \; \langle Person \rangle$ of Example 6.3 where

$$\begin{aligned} a_2 &\equiv \text{“}computers\text{”} \in <p_1>.expertise \\ a_3 &\equiv \text{“}medicine\text{”} \in <p_2>.expertise \\ a_4 &\equiv c \in <p_1, p_2>.children \end{aligned}$$

which returns the children which have a doctor and computer scientist as parents. The subquery satisfies the conditions of rule 6.47 when we substitute a_2 for F_1, $a_3 \wedge a_4$ for F_2, $Person$ for P, c for t, $\{\}$ for $Qset$ and $Person$ for $Sset$ resulting in the transformation

$$Person \; \gamma^c_{a_2 \wedge a_3 \wedge a_4} \; \langle Person \rangle \quad \Leftrightarrow \quad (Person \; \sigma_{a_2} \; \langle \rangle) \; \gamma^c_{a_3 \wedge a_4} \; \langle Person \rangle$$

Ideally we would like to apply this rule again to break out atom a_3 which restricts the variable p_2 ranging over the class $Person$. Noting that the ordering of argument sets does not affect the result of a generate operation, we can apply rule 6.4 to give

$$(Person \; \sigma_{a_2} \; \langle \rangle) \; \gamma^c_{a_3 \wedge a_4} \; \langle Person \rangle \quad \Leftrightarrow \quad Person \; \gamma^c_{a_3 \wedge a_4} \; \langle (Person \; \sigma_{a_2} \; \langle \rangle) \rangle$$

Now we can apply rule 6.47 again to break out atom a_3 as follows.

$$Person \; \gamma^c_{a_3 \wedge a_4} \; \langle (Person \; \sigma_{a_2} \; \langle \rangle) \rangle \quad \Leftrightarrow \quad (Person \; \sigma_{a_3} \; \langle \rangle) \; \gamma^c_{a_4} \; \langle (Person \; \sigma_{a_2} \; \langle \rangle) \rangle$$

The result of these steps is shown on the right hand side of Figure 6.2. ◇

A special case of the generate operation occurs when the predicate generates values for the target variable (which does not range over an argument set) and also for a variable which does range over an argument set.

Theorem 6.16 *Factoring Generate from a Conjunctive Generate:*

$$P\ \gamma^{t}_{(F_1 \wedge F_2)}\ \langle Qset, R, Sset\rangle \ \overset{c}{\Leftrightarrow}\ ((P\ \gamma^{r}_{F_1}\ \langle Qset\rangle) \cap R)\ \gamma^{t}_{F_2}\ \langle Sset\rangle \qquad (6.56)$$

where:

$$c:\ \mathrm{ref}(F_1, (p, r, q_1 \ldots q_k))\ \wedge\ \mathrm{gen}(F_1, r)\ \wedge\ \mathrm{ref}(F_2, (r, t, s_1 \ldots s_m))\ \wedge\ \mathrm{gen}(F_2, t)$$

Proof: The condition states that F_1 generates values for r while F_2 generates values for the target variable t. Similar to rule 6.46, we now have two sources of values for r; the argument set R and the generating atom in F_1. Since the final values of r must exist in R and be generated by F_1, we can break F_1 out into its own generate operation and intersect the result with R prior to generating values for t. □

6.6 Semantic Rewrite Rules

This section presents transformations for object algebra expressions which utilize semantic information embodied in the data model, its restrictions, and the class inheritance graph. As each database has its own class lattice with a unique distribution of objects across those classes, the applicability of these transformations is highly dependent on the database state. The intent is that a query optimizer[1] with access to simple database statistics can easily adapt its optimization choices to the current database state. Four primary themes will be developed. First, a simple rule regarding class extents is presented. This is followed by the identification of binary operators which default to known solutions in special cases. Lastly, a transformation which depends on type consistency considerations is presented. In the following, c_i will denote a class while C_i and C_i^* denote the extent and deep extent of c_i respectively.

6.6.1 Deep Extent Expansion

Theorem 6.17 *Deep Extent Expansion:*

Each occurrence of C^* in an object algebra expression can be replaced by $C \cup C_1 \cup \cdots \cup C_n$ where $\{c, c_1, \ldots, c_n\}$ is the set containing c and all its n subclasses.

Proof: Operands to object algebra operators are sets of objects. At the leaves of a query tree these sets of objects represent either a class extent or a class deep extent. The data model states that the deep extent of class c, $ext^*(c)$, is the union of the extents of all classes in the subtree of the class lattice rooted at c. Thus

$$ext^*(c) \equiv ext(c) \cup ext(c_1) \cup \cdots \cup ext(c_n)$$

[1]Even though the terms 'optimum' and 'optimization' are used frequently in relation to query processing, optimality in this context needs to be interpreted somewhat loosely. There is a significant amount of parameter estimation incorporated into cost function parameters. Thus, the decisions are optimal to the extent that these estimates are accurate.

which is equivalent in meaning to the theorem statement. □

Example 6.5 Given the class lattice in Figure 6.4 the following expression holds:

$$C_1^* \ \sigma_F \langle\ \rangle \quad \Leftrightarrow \quad (C_1 \cup C_2 \cup C_3 \cup C_4)\ \sigma_F \langle\ \rangle$$

This example implements a full expansion of the classes in a deep extent. Of course, any subset of a union which identifies a deep extent can be replaced by the deep extent as well. Thus

$$(C_1 \cup C_2 \cup C_3^*)\ \sigma_F \langle\ \rangle$$

is also an equivalent expression. ◇

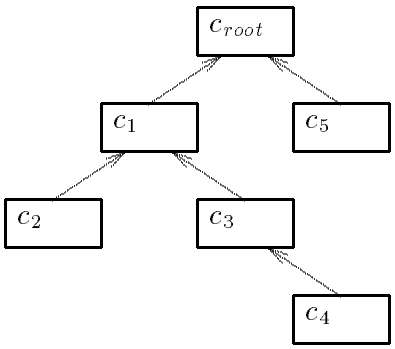


Figure 6.4: A simple class lattice.

6.6.2 Semantic Transformations for Binary Operators

Knowledge about the class lattice can be used to identify and reduce special cases of binary operations. In order to discuss these special cases in a uniform manner, a binary relation which captures information about the class inheritance graph is defined.

Definition 6.2 *Deep Extent Overlap:* The notation $c_1 \bigtriangledown c_2$ denotes that $\exists c_i \mid c_i \preceq c_1 \ \wedge \ c_i \preceq c_2$, i.e., C_1^* and C_2^* have elements in common. The relation $\bigtriangledown$ is reflexive, symmetric but not transitive. □

Theorem 6.18 The following special cases of binary operations are identified:

1. $C_1 \cap C_2 = \phi$ when $c_1 \neq c_2$.
   Proof: The data model states that an object may be a member of only one class, i.e. the extent of any two distinct classes is disjoint.
2. $C_1 - C_2 = C_1$ when $c_1 \neq c_2$.
   Proof: The proof follows from the previous case.

3. $C_1 \cap C_2^* = C_1$ when $c_1 \preceq c_2$.
   Proof: By definition of the $\preceq$ and $\bigtriangledown$ operators it is clear that if $c_1 \preceq c_2$ then all elements in C_1 are also in C_2^*. Since no member of c_1 can be a member of any other class, the assertion is true.

4. $C_1 \cap C_2^* = \phi$ when $c_1 \not\bigtriangledown c_2$.
   Proof: By Definition 6.2, C_1^* (which includes C_1) and C_2^* have no elements in common when $c_1 \not\bigtriangledown c_2$.

5. $C_1 \cup C_2^* = C_2^*$ when $c_1 \preceq c_2$.
   Proof: By definition, C_2^* includes all elements in the extent of each class which is a subclass of c_2.

6. $C_1 - C_2^* = C_1$ when $c_1 \neq c_2 \wedge c_1 \not\preceq c_2$.
   Proof: The condition $c_1 \neq c_2$ implies that c_1 and c_2 have no members in common, i.e. $C_1 \cap C_2 = \phi$. $c_1 \not\preceq c_2$ implies that no subclass of c_1 or c_2 has members in common. Thus C_1 and C_2^* have no elements in common.

7. $C_1^* - C_2 = C_1^*$ when $c_1 \not\bigtriangledown c_2$.
   Proof: By definition, $c_1 \not\bigtriangledown c_2$ means that $C_1^* \cap C_2^* = \phi$. Since $C_2 \subseteq C_2^*$, $C_1^* \cap C_2 = \phi$ as well.

8. $C_1^* \cap C_2^* = \phi$ when $c_1 \not\bigtriangledown c_2$.
   Proof: By Definition 6.2.

9. $C_1^* \cap C_2^* = \cup C_i, c_i \preceq c_1 \wedge c_i \preceq c_2$ when $c_1 \bigtriangledown c_2$
   Proof: Let $c_{11}, \ldots, c_{1n}$ be all the classes in the subtree of the class inheritance graph rooted at c_1. Applying Theorem 6.17, $C_1^* \cap C_2^*$ can be rewritten as:

   $$(C_{11} \cup \cdots \cup C_{1n}) \cap C_2^*$$

   Applying equation 6.12, this is equivalent to

   $$(C_{11} \cap C_2^*) \cup \cdots \cup (C_{1n} \cap C_2^*)$$

   As shown in 3 above, each term $(C_{1i} \cap C_2^*) = C_{1i}$ when $c_{1i} \preceq c_2$.

10. $C_1^* \cup C_2^* = C_2^*$ when $c_1 \preceq c_2$.
    Proof: By Definition 6.2, $c_1 \preceq c_2 \Rightarrow C_1^* \subseteq C_2^*$.

11. $C_1^* - C_2^* = C_1^*$ when $c_1 \not\bigtriangledown c_2$.
    Proof: By Definition 6.2 $c_1 \not\bigtriangledown c_2 \Rightarrow C_1^* \cap C_2^* = \phi$. □

These transformations are summarized in Table 6.1

6.6.3 Other Semantic Transformations

Transformation rules may rely on type consistency to determine their applicability. Consider the following rules.

Table 6.1: Special cases of the binary operators.

Rule	Condition
$C_1 \cap C_2 = \phi$	$c_1 \neq c_2$
$C_1 - C_2 = C_1$	$c_1 \neq c_2$
$C_1 \cap C_2^* = C_1$	$c_1 \preceq c_2$
$C_1 \cap C_2^* = \phi$	$c_1 \not\bigtriangledown c_2$
$C_1 \cup C_2^* = C_2^*$	$c_1 \preceq c_2$
$C_1 - C_2^* = C_1$	$c_1 \neq c_2 \wedge c_1 \not\preceq c_2$
$C_1^* - C_2 = C_1^*$	$c_1 \not\bigtriangledown c_2$
$C_1^* \cap C_2^* = \phi$	$c_1 \not\bigtriangledown c_2$
$C_1^* \cap C_2^* = \cup C_i$	$c_i \preceq c_1 \wedge c_i \preceq c_2 \wedge c_1 \bigtriangledown c_2$
$C_1^* \cup C_2^* = C_2^*$	$c_1 \preceq c_2$
$C_1^* - C_2^* = C_1^*$	$c_1 \not\bigtriangledown c_2$

Theorem 6.19 *Associativity of Select with respect to Intersect* (Figure 6.5)*:*

$$(P\ \sigma_F\ \langle Qset\rangle) \cap R \overset{c}{\Leftrightarrow} (P\ \sigma_F\ \langle Qset\rangle) \cap (R\ \sigma_{F'}\ \langle Qset\rangle) \quad (6.57)$$
$$\overset{c}{\Leftrightarrow} P \cap (R\ \sigma_{F'}\ \langle Qset\rangle) \quad (6.58)$$

where:

c: F' is identical to F except each occurrence of p is replaced by r.

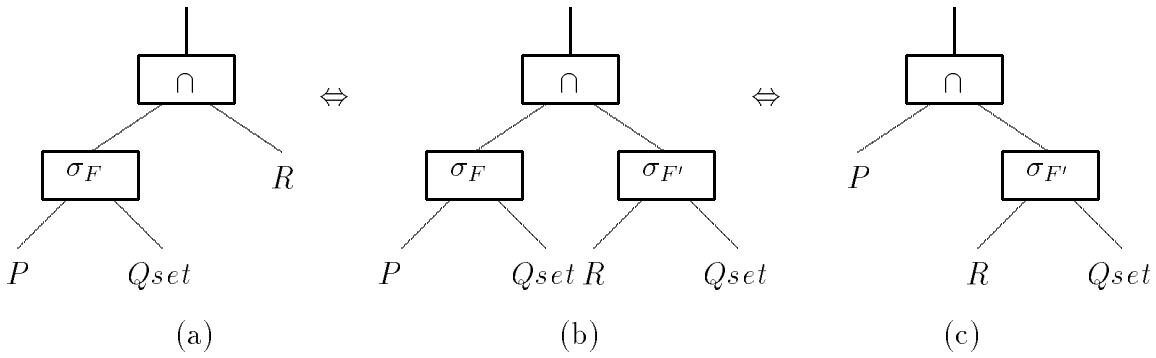


Figure 6.5: Graphical representation of Theorem 6.5.

Proof: By definition, intersection returns only those objects which are present in both input sets. From a pure set theory perspective, any restriction which removes objects from one input set only, automatically excludes those objects from the result of the intersection. In addition, the restriction could be applied equally well to the other input set instead and generate the same result. This is because intersection is not dependent on operand ordering and "doesn't care" which input set is missing the excluded objects. By a similar argument, the restriction could be applied to both input sets without affecting the result.

Referring to Figure 6.5, the restriction in (a) is given by the select operation $P\ \sigma_F\ \langle Qset\rangle$; i.e., the input set P is being restricted by predicate F. Let us first examine the transformation from (a) to (b).

The expression depicted in (a) is considered type consistent if the methods in predicate F are defined on all types represented by the objects in P. Note that P (R) may be a heterogeneous set of objects if it represents the result of a subquery as opposed to the extent of some class. The transformation from (a) to (b) is valid only if the expression in (b) is a legal. This means the subquery $R\ \sigma_{F'}\ \langle Qset \rangle$ in (b) must be type consistent, i.e., the methods in predicate F' are defined on all types represented by the objects in R. According to condition c on the transformation rule though, the methods in F' are identical to those in F. Thus, the validity of the transformation from (a) to (b), and by similar argument from (c) to (b), is dependent on both the database and the nature of the subqueries which produce P and R.

Once these transformations are shown to be valid, we can conclude that the equivalence of expressions (a) and (c) is valid by the characteristics of intersection discussed previously. If the restriction can legally be applied to both input sets, then intersection only requires that it be applied to one of them. □

6.7 Rule Application

The focus of this chapter is on the specification of transformation rules as opposed to their application. However, some comments on the application of the rules are in order. Several rule based optimizers have been reported [GD87, HP88] which could be adapted to apply the rules developed here. The main component of these systems is a pattern matching engine which matches subexpressions of a query against rules. In addition to the matching of subexpressions, the firing of rules is dependent on the satisfaction of user defined functions such as $\text{gen}(F, v)$ and $\text{res}(F, v)$.

There is, however, one major difference between our rules and those of the EXODUS [GD87] and Starburst [HP88] optimizers. Their rules are based on expressions of fixed arity, e.g., two operand joins, while our rules can have varying numbers of arguments, e.g., $Qset$, $Rset$, $Sset$. An inefficient solution to this incompatibility would be to define rules for all possible set sizes and combinations. A better approach would be to modify the pattern matching engine to accommodate arguments to expressions which are sequences (sets).

Several heuristics can be used to drive the rewriting process. A common heuristic is to push operations which reduce intermediate result sizes as far down the tree as possible. The result is that operand sets are are reduced as early as possible thereby minimizing the input to operations higher up in the query tree.

Another heuristic is to eliminate redundant or useless cross product generation. The semantics of select, generate and map operations require that the predicate (or method applications in the case of a map) be evaluated for each element in the cross product of the argument sets. For example, the predicate F in $P\ \sigma_F\ \langle Qset, Rset \rangle$ is evaluated once for each vector $<p, q_1 \ldots q_k, r_1 \ldots r_l>$ $\in\ P \times Q_1 \times \ldots \times Q_k \times R_1 \times \ldots \times R_l$. When F consists of the subformulas $F_1(p, q_1 \ldots q_k)$ and $F_2(p, r_1 \ldots r_l)$ then the r elements of the vector are not used in evaluating F_1 and the q elements are not used in evaluating F_2. Assuming that the cost of generating the cross product grows non-linearly with respect to the number of sets

(here $k+l+1$), it is cheaper to generate smaller cross products, $P \times Q_1 \times \ldots \times Q_k$ and $P \times R_1 \times \ldots \times R_l$ respectively.

One effect of applying these rules is that an operation with a complex predicate is split into multiple operations whose predicates are subformulas of the original one. This can be beneficial in several ways. Assume that access plans contain calls to an object manager interface which has the ability to apply single atom predicates to streams of objects. Simplifying predicates during query rewrite will reduce the complexity of access plan generation as there is a closer mapping between predicates and individual object manager calls. Another benefit is that applying a rule may expose operations which can be performed in parallel. This becomes more important as distribution is considered.

Chapter 7

Generating Query Execution Plans

7.1 Introduction

This chapter addresses the last step in the query processing methodology of Figure 1.3, namely access plan generation. Access plan generation is the process of mapping high level representations of queries (e.g., relational or object algebra expressions) to sequences of data manipulation operators which are present in the physical system. In the case of the relational data model [Cod70], there is a close correspondence between algebra operations and the low level primitives of the physical system [SAC+79]. The mapping between relations and files, and tuples and records may have contributed to this strong correspondence. However, there is no analogous, intuitive correspondence between object algebra operators and physical system primitives. Thus any discussion of access plan generation must first define the low level object manipulation primitives which will be the building blocks of access plans.

We call this low level object manipulation interface the Object Manager (OM) interface. Object managers have received attention lately in the context of distributed systems [BHJL86, DLA88, MG89, VKC86], programming environments [Dec86, Kae86, VBD89] and databases [CDRS86, CM84, EE87, HZ87, Kim88]. These object managers differ in terms of their support for data abstraction, concurrency and object distribution. In addition, they are typically oriented towards "one-at-a-time" object access which is an inefficient paradigm for query processing. We define a new OM interface which maintains many features of previous object managers but operates on streams of objects. We then develop algorithms for generating access plans whose processing steps are calls to the stream oriented object manager interface.

The rest of the chapter is organized as follows. Section 7.2 presents the object manager interface. Two algorithms for developing query execution plans are developed. The algorithm of Section 7.3 is simple but may not find best plans. Section 7.4 presents a more complex algorithm which finds all feasible plans and shows how OM cost functions can be used to select a best plan.

7.2 The Object Manager

There are two primary concerns in generating access plans. The first is to decompose the object algebra operators union ($\cup$), difference ($-$), select (σ_F), map ($\mapsto_{mlist}$) and generate (γ_F^t) (especially those with complex predicates) into a sequence of simpler operations which more accurately reflect the interface provided by a real object-based system. In other words, we are defining a lower level of abstraction than that provided by the data model and object algebra thus far, and treat access plan generation as the mapping of object algebra expressions to the new abstraction interface. The second concern is that in order to maintain the data abstraction provided by behaviorally defined objects we can not make assumptions about how objects are stored and implemented.

Object algebra expressions which are the input to the access plan generation process have several important characteristics:

1. They can be represented as graphs whose nodes are object algebra operators and whose edges represent streams (sets) of objects. Thus intermediate results do not have any structure. In fact, the intermediate results can be thought of as streams of individual object identifiers.

2. Some algebra operators (σ_F, γ_F^t) are qualified by a predicate. Predicates are formed as a conjunction of atoms, each of which may reference several variables. The variable corresponding to the result of the algebra operation is called the *target variable*.

3. Variables used in multiple atoms of a predicate imply a 'join' of some kind; i.e., objects denoted by the variable must satisfy several conditions concurrently. When object variables overlap in multiple atoms, the 'join' relationship becomes complex and may involve several variables at once.

The last point, namely implied 'joins' between object variables within a predicate, is the driving factor behind the query execution and access plan strategy. Consider the predicate F for the select operation $P\ \sigma_F\ \langle O, R, S, T\rangle$

$$F \quad \equiv \quad o == (<p, q, r>.m_1) \ \wedge \ (q \in t) \ \wedge \ (q == <s>.m_2) \tag{7.1}$$

where p is the target variable and O, P, R, S, T are inputs to the operation. All values for q are generated by the atoms in the predicate. The result of this select operation can be defined as

$$\{\ o \quad | \quad F(o, p, q, r, s, t) \text{ is true for } <o, p, r, s, t> \in \ O \times P \times R \times S \times T\ \} \tag{7.2}$$

Table 7.1 identifies which variables are referenced in each atom (numbered left to right) and reflects the dependencies between the variables. It should be clear from the table that an object denoted by q must satisfy all atoms concurrently. However, if we are to respect the data abstraction afforded by objects, then it is not possible for the query processor to directly evaluate all three atoms concurrently as required.

	o	p	q	r	s	t	
$a1$	x	x	x	x			$(o ==<p,q,r>.m_1)$
$a2$			x			x	$(q \in t)$
$a3$			x		x		$(q ==<s>.m_2)$

Table 7.1: Dependencies between variables in a predicate.

Instead, it is more likely that we call upon another agent which can perform individual operations on objects that correspond to the individual atoms. This would then require the ability to keep track of the combinations of variables in $O \times P \times R \times S \times T$ which satisfy F. This intuition leads to the following design decisions.

1. The low level operators used to generate an access plan for an algebra level operator will consume and generate streams (sets) of tuples of object identifiers. The notation $[a, b, c, \cdots]$ is used to denote a stream of tuples of object identifiers of the form $\{<a, b, c, \cdots>\}$. For convenience this is called an *oid-stream* in the remainder of the chapter. This way, relationships among variables and the atoms they satisfy can be maintained over a sequence of operations.

2. There exists an 'object manager' interface which performs low level operations comparable to individual atoms in a predicate.

The following section defines the object manager interface.

7.2.1 Object Manager Interface Specification

The object manager interface specifies a calling sequence and semantics for performing operations on oid-streams. Four operation types are defined:

1	$\mathbf{OM}_{\cup}([i_1], [i_2], [o])$	– stream union
2	$\mathbf{OM}_{diff}([i_1], [i_2], [o])$	– stream difference
3	$\mathbf{OM}_{eval}([i_1], \ldots, [i_n], [o], meth, pred)$	– atom evaluation
4	$\mathbf{OM}_{\bowtie}([i_1], \ldots, [i_n], [o])$	– stream reduction

where $[i_n]$ and $[o]$ denote input and output oid-streams respectively. The semantics of the OM calls are described next.

(1) Stream Union: This operator generates the union of the two input oid-streams. Streams $[i_1]$ and $[i_2]$ must reference the same variable names though not necessarily in the same order. The operation is analogous to the relational union operator. The output oid-stream contains those tuples which are present in $[i_1]$ or $[i_2]$ projected onto the variables identified by the output specifier $[o]$.

(2) Stream Difference: This operator generates the difference of the two input oid-streams. Streams $[i_1]$ and $[i_2]$ must reference the same variable names though not necessarily in the same order. The operation is analogous to the relational difference operator. The output oid-stream contains those tuples which are in $[i_1]$ but not in $[i_2]$ projected onto the variables identified by the output specifier $[o]$.

(3) Atom Evaluation: This operator applies the (optional) method given by *meth* to each member of $[i_1] \times \ldots \times [i_n]$ creating the intermediate oid-stream $[i_1] \times \ldots \times [i_n] \times [res]$ where res is the result of the method application for each $i_1, \ldots, i_n$ combination. Next, the predicate *pred* is applied to the intermediate oid-stream and the result is projected onto those variables given in the output stream identifier $[o]$. More specifically:

- $[i_1], \ldots, [i_n]$ denote a set of oid-streams which represent the input to the object manager call. A variable name may appear in only one input stream.
- $[o]$ denotes the oid-stream which will be returned as output of the object manager call. A variable name may appear only once in the output stream. Variables referenced in the oid-stream $[o]$ are a subset of those in the input streams or the special identifier res.
- *meth* is an optional method application specifier of the form $<a, b, c, \cdots>.mname$, where $a, b, c, \cdots$ correspond either to variables in the input streams or are the textual representation of an atomic value. The special identifier res denotes the result of the method application and can be referenced in the output stream and predicate.
- *pred* is an optional predicate on objects in the input streams and/or result of the *meth* field. The full set of permissible predicates is given in Table 7.2. Variables in the predicate correspond either to variables in the input streams, the special identifier res or are the textual representation of an atomic value (denoted by *const* in the table).

Table 7.2: Predicates allowed in $\mathbf{OM}_{eval}$ calls.

$o_i == o_j$
$o_i \in o_j$
$o_i =_{\{\}} o_j$
$const = o$
$const \in o$
$o \in const$
$const =_{\{\}} o$

An $\mathbf{OM}_{eval}$ call must have either a method or a predicate specified, and can have both if required. If specified, the method is always applied before the predicate is evaluated. The special identifier res denotes the result of the method application and can be referenced in the output stream or predicate only if a method is specified.

The input streams may contain variables which are not referenced in the output stream, the method or the predicate. In this case the respective oids in the input streams are ignored. Variables referenced in the input streams and output stream but not in the method or predicate are carried through without modification. In this case, the unreferenced oid in each input tuple which satisfies

the predicate after the optional method has been applied is copied unchanged to the corresponding output tuple. There is no relationship or restrictions on the ordering of variables in the input streams and output stream.

Example 7.1 Consider the atom evaluation operation

$$\mathbf{OM}_{eval}([a,b],[c],[res,c],<c,a>.m,b \in res)$$

The semantics of this operation are given by the following algorithm.

for (each tuple $t : <a,b,c> \in [a,b] \times [c]$) **begin** – iterate over $\times$-product
  let res be the object returned by $<t.c, t.a>.m^1$– method application
  if $(t.b \in res)$ **then** – set value inclusion
    add tuple $<res, t.c>$ to the output stream
end $\diamond$

(4) Stream Reduction: This operator combines and reduces the number of input streams by performing an equi join on those variables which are common to all input streams. This requires that all input streams have at least one variable name in common. The semantics of the operation is best described using an example.

Example 7.2 Consider the stream reduction

$$\mathbf{OM}_{\bowtie}([a,b,c],[b,d,c],[e,c,b],[a,b,e])$$

The variables common to all input streams are b and c. We can rewrite the operation as

$$\mathbf{OM}_{\bowtie}([a,b_1,c_1],[b_2,d,c_2],[e,c_3,b_3],[a,b,e])$$

in order to differentiate the different sources for variables b and c. The input streams are first combined by taking their cross product which results in the oid-stream $[a,b_1,c_1,b_2,d,c_2,e,c_3,b_3]$. The final result stream is of the form $[a,b,e]$ and contains only those tuples from the previous intermediate result where $(b_1 = b_2 = b_3) \wedge (c1 = c2 = c3)$. $\diamond$

7.3 Access Plan Generation

Access plan generation can be thought of as creating a mapping from object algebra expression trees to trees of object manager operations. A query is initially represented as a tree of object algebra operators as shown in Figure 7.1(a). Edges in the figure have been annotated with oid-stream labels to indicate that a set of objects, e.g., P, can be considered a stream of individual objects, e.g., $[p]$, as well. One unique feature of object algebra expression trees is that all edges represent streams of single objects,

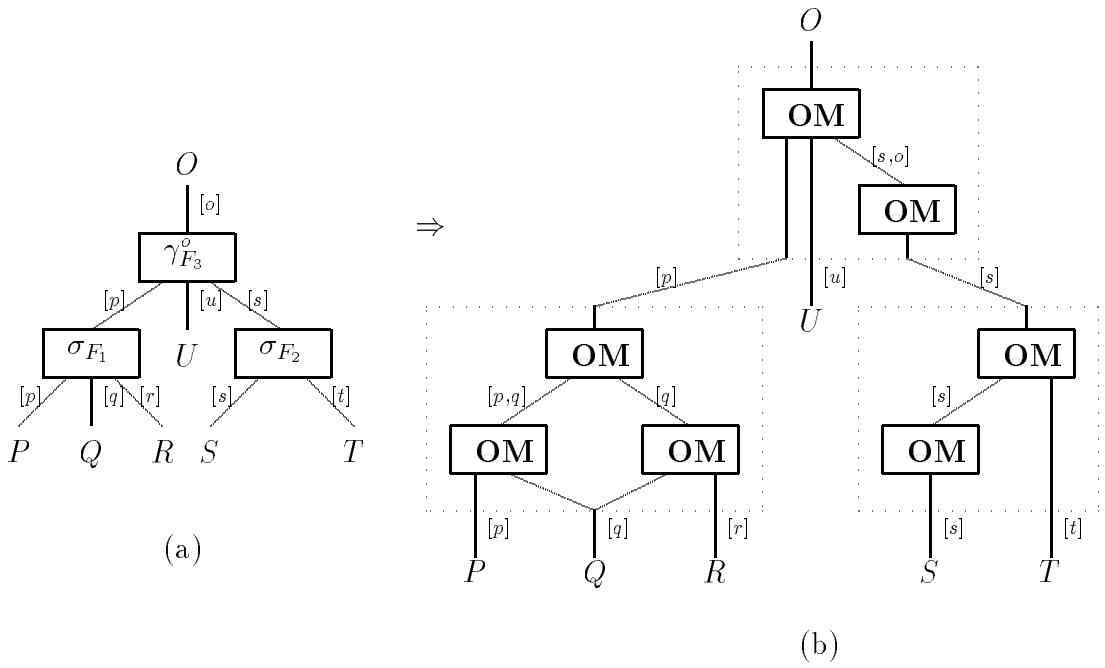


Figure 7.1: Mapping object algebra expression trees to object manager operation trees.

never streams of multiple objects. This is due to the closed nature of the algebra which insures that the output of any operation can be used as input to another.

The graph in Figure 7.1(b) represents an access plan corresponding to the algebra tree on the left. An *access plan graph* is a graph whose nodes are OM operators and whose edges are oid-streams. It is evaluated from the leaves to the root. The subtrees within dotted boxes are sequences of object manager operations corresponding to individual algebra operators of the original query. Edges which do not cross subtree boundaries may represent streams of tuples of objects (e.g., $[p, q]$ and $[s, o]$). In addition, streams may be used as input to multiple object manager operations within a subtree, e.g., $[q]$.

The following sections shows how the mapping to object manager operators is performed for each of the object algebra operators ($\cup$, $-$, σ_F, $\mapsto_{mlist}$ and γ^t_F). For ease of presentation, how these operations can be mapped to access plan graphs is first discussed intuitively and then formal algorithms are developed.

7.3.1 Union and Difference Operations

The union and difference operators map directly to their object manager counterparts. Inputs and output of these two algebra operations are always unary streams of objects even though $\mathbf{OM}_{\cup}$ and $\mathbf{OM}_{diff}$ accept streams of tuples of object identifiers.

[1] We use the notation $t.c$ to denote component c of tuple t.

7.3.2 Map Operation

Reviewing briefly, the map operator $Q_1 \mapsto_{m_1 \ldots m_n} \langle Q_2, \ldots, Q_k \rangle$ denotes the multi-operation $<q_1, \ldots, q_k>.m_1 \ldots m_n$ where $<q_1, \ldots, q_k>$ are drawn from $Q_1 \times \ldots \times Q_k$. Since the object manager interface can only apply one method per call, the multi-operation must be decomposed into individual method applications. Determining which q_i are a parameter for a given m_j has been treated previously in Chapter 5 and is not repeated here. Figure 7.2 depicts how the map operation $Q_1 \mapsto_{m_1.m_2.m_3.m_4} \langle Q_2, Q_3, Q_4, Q_5, Q_6, Q_7 \rangle$ is represented as a sequence of OM operations. The formal steps to perform this transformation is given in Algorithm 7.1.

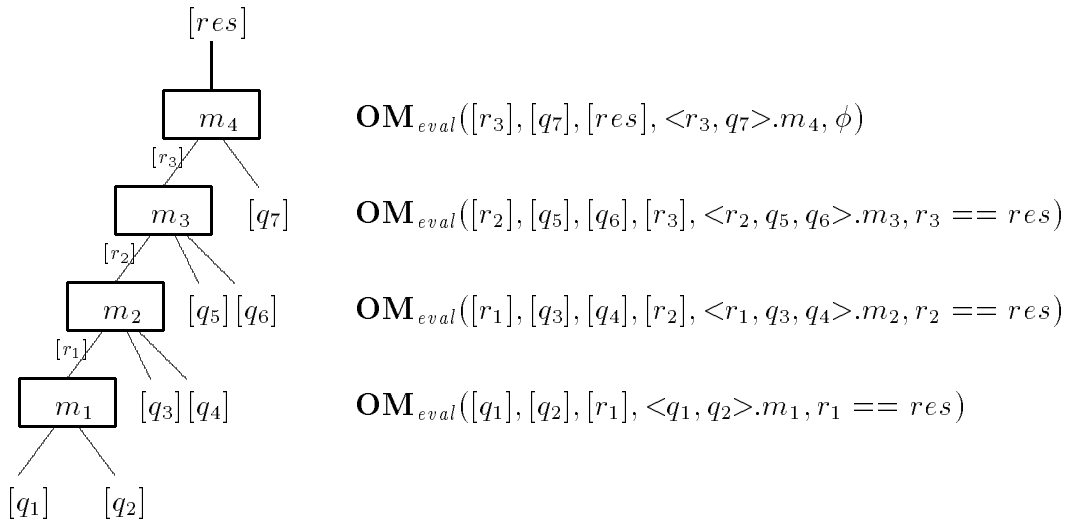


Figure 7.2: Access plan generation for a sample map operation.

Algorithm 7.1 *Create access plan graph for the map operator*
Input: Object algebra map operation of the form $Q_1 \mapsto_{m_1 \cdots m_n} \langle Q_2, \ldots, Q_k \rangle$
Output: Access plan graph G

begin
let $G := \phi$ – initialization (1)
use the type inference rules of Chapter 6 to create mappings $m_i \rightarrow q_x, \ldots, q_y$ where:(2)
– i denotes the i^{th} method in $m_1 \cdots m_n$
– x denotes the subscript in $Q_1, \ldots, Q_k$ of the second argument to m_i
– y denotes the subscript in $Q_1, \ldots, Q_k$ of the last argument to m_i
let $r_0 := q_1$ (3)
for $(i := 1$ **to** $n)$ **begin** (4)
add node $\mathbf{OM}_{eval}([r_{i-1}], [q_x], \ldots, [q_y], [r_i], <r_{i-1}, q_x, \ldots, q_y>.m_i, r_i == res)$ to G(5)
if (this is not the first node to be placed) **then** (6)
add an edge to G connecting the output of the node placed on the previous (7)
pass to the first input ($[r_{i-1}]$) of the node placed on this pass (8)
endif (9)
endfor (10)

end Algorithm 7.1

7.3.3 Select and Generate Operations

The select and generate operators introduce complexity into access plan generation due to their use of predicates. At first it may appear that the two should be treated separately as select returns a subset of an input set while the generate generates objects from those in the input sets. But from the perspective of low level access plan creation, they are quite similar. Consider again the selection predicate of Equation 7.1. Even though the operation is a selection, the predicate generates values for q. There is no inherent difference in complexity between predicates for selections and those for generate operations. The only real distinction between the two is that the target variable of a generate operation does not correspond to one of the input sets.

Figures 7.3 and 7.4 illustrate several possible access plans, specified as access plan graphs, for the select example whose predicate is given in Equation 7.1. Nodes in these access plan graphs are either $\mathbf{OM}_{eval}$ operations corresponding to individual atoms of the predicate or $\mathbf{OM}_{\bowtie}$ operations for reducing intermediate oid-streams. $\mathbf{OM}_{eval}$ nodes are labeled with the atom they represent ($a1$, $a2$ or $a3$) while $\mathbf{OM}_{\bowtie}$ nodes are labeled with the $\bowtie$ symbol.

The first requirement in creating these access plans is to rewrite the predicate such that each atom does indeed correspond to just a single object manager call. This is done by applying the following rewrite rules.

1. Given an atom of the form $const = var$ where $const$ is the textual representation of an atomic value and var is some object variable, remove the atom from the predicate and replace each occurrence of var with the text of $const$.

2. Replace all occurrences of atoms of the form

   $$(\text{multi-op}_1\ \theta\ \text{multi-op}_2)$$

   with

   $$(\text{multi-op}_1\ \theta\ new\_var)\ \wedge\ (new\_var\ \theta\ \text{multi-op}_2)$$

   where new_var is a newly introduced object variable currently unused in the predicate and $\theta \in \{=, ==, \in, =_{\{\}}\}$. This insures there is only one multi-operation per atom.

3. Expand each multi-operation into a sequence of single method applications. This is identical to the steps performed for the map operator in Section 7.3.2. Using the map operation of Figure 7.2 as an example, the following equivalence would be used:

   $$\begin{aligned} <q_1, q_2, q_3, q_4, q_5, q_6, q_7>.m_1.m_2.m_3.m_4 \quad \equiv \quad & r_1 == <q_1, q_2>.m_1\ \wedge \\ & r_2 == <r_1, q_3, q_4>.m_2\ \wedge \\ & r_3 == <r_2, q_5, q_6>.m_3\ \wedge \\ & res == <r_3, q_7>.m_4 \end{aligned}$$

where r_i and res are previously unused object variables.

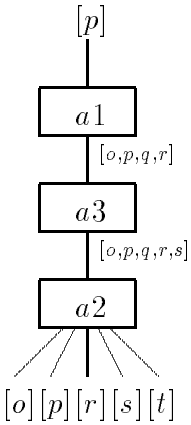
Table 7.3 illustrates how each of the simplified atoms is mapped to an object manager call. The table shows all variables from the input streams being carried through to the output stream merely to illustrate the full extent of possibilities. Typically the output stream would be a subset of the input stream variables. The "$*$" character denotes the sequence of all variable names $o, a, b, \cdots$.

Table 7.3: Mapping simplified atoms to $\mathbf{OM}_{eval}$ calls.

	Object Manager Call	Comment
$a == b$	$\mathbf{OM}_{eval}([a,b],[*],\phi, a == b)$	$\text{res}(a,b)$
$o ==<a,b,\cdots>.m$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m,\phi)$	$\text{gen}(o)$
	$\mathbf{OM}_{eval}([o,a,b,\cdots],[*],<a,b,\cdots>.m, o == res)$	$\text{res}(*)$
$a \in b$	$\mathbf{OM}_{eval}([b],[*],\phi, a \in b)$	$\text{gen}(a)$
	$\mathbf{OM}_{eval}([a,b],[*],\phi, a \in b)$	$\text{res}(a,b)$
$o \in <a,b,\cdots>.m$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m, o \in res)$	$\text{gen}(o)$
	$\mathbf{OM}_{eval}([o,a,b,\cdots],[*],<a,b,\cdots>.m, o \in res)$	$\text{res}(*)$
$<a,b,\cdots>.m \in o$	$\mathbf{OM}_{eval}([o,a,b,\cdots],[*],<a,b,\cdots>.m, res \in o)$	$\text{res}(*)$
$a =_{\{\}} b$	$\mathbf{OM}_{eval}([a,b],[*],\phi, a =_{\{\}} b)$	$\text{res}(a,b)$
$o =_{\{\}} <a,b,\cdots>.m$	$\mathbf{OM}_{eval}([o,a,b,\cdots],[*],<a,b,\cdots>.m, o =_{\{\}} res)$	$\text{res}(*)$
$const =<a,b,\cdots>.m$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m, const = res)$	$\text{res}(a,b,\cdots)$
$const \in o$	$\mathbf{OM}_{eval}([o],[o],\phi, const \in o)$	$\text{res}(o)$
$const \in <a,b,\cdots>.m$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m, const \in res)$	$\text{res}(a,b,\cdots)$
$<a,b,\cdots>.m \in const$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m, res \in const)$	$\text{res}(a,b,\cdots)$
$const =_{\{\}} o$	$\mathbf{OM}_{eval}([o],[o],\phi, const =_{\{\}} o)$	$\text{res}(o)$
$const =_{\{\}} <a,b,\cdots>.m$	$\mathbf{OM}_{eval}([a,b,\cdots],[*],<a,b,\cdots>.m, const =_{\{\}} res)$	$\text{res}(a,b,\cdots)$

We are now ready to consider, in some detail, the alternative access plans in Figures 7.3 and 7.4 for the sample select operation. Each access plan assumes that atoms in the predicate are sorted into a partial dependency order where values for non-input variables are generated before they are used; i.e., an atom which generates q comes before an atom which uses q. In the predicate of Equation 7.1 variable q is 'generated' twice, once by $a2 : q \in t$ and again by $a3 : q ==<s>.m_2$, and only 'referenced' in $a2 : o ==<p,q,r>.m_1$. Thus one of either $a2$ or $a3$ must be performed before $a1$. However, once values for q have been generated by one of these atoms, the semantics of conjunction allow us to think of the remaining atom as restricting q as opposed to generating new values for q. As long as oid-streams for all variables referenced by the remaining two atoms are present, their ordering is irrelevant. The access plans of Figures 7.3 and 7.4 express some of the variations which are possible within this framework.

Plan 1

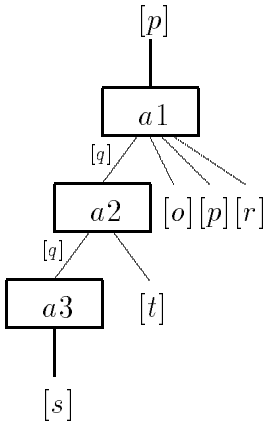


$\mathbf{OM}_{eval}([o,p,q,r],[p],<p,q,r>.m_1, o == res)$

$\mathbf{OM}_{eval}([o,p,q,r,s],[o,p,q,r],<s>.m_2, q == res)$

$\mathbf{OM}_{eval}([o],[p],[r],[s],[t],[o,p,q,r,s],\phi, q \in t)$

Plan 2

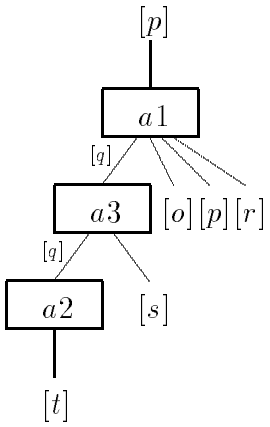


$\mathbf{OM}_{eval}([q],[o],[p],[r],[p],<p,q,r>.m_1, o == res)$

$\mathbf{OM}_{eval}([q],[t],[q],\phi, q \in t)$

$\mathbf{OM}_{eval}([s],[q],<s>.m_2, q == res)$

Plan 3



$\mathbf{OM}_{eval}([q],[o],[p],[r],[p],<p,q,r>.m_1, o == res)$

$\mathbf{OM}_{eval}([q],[s],[q],<s>.m_2, q == res)$

$\mathbf{OM}_{eval}([t],[q],\phi, q \in t)$

Figure 7.3: Access plans for the sample select operation.

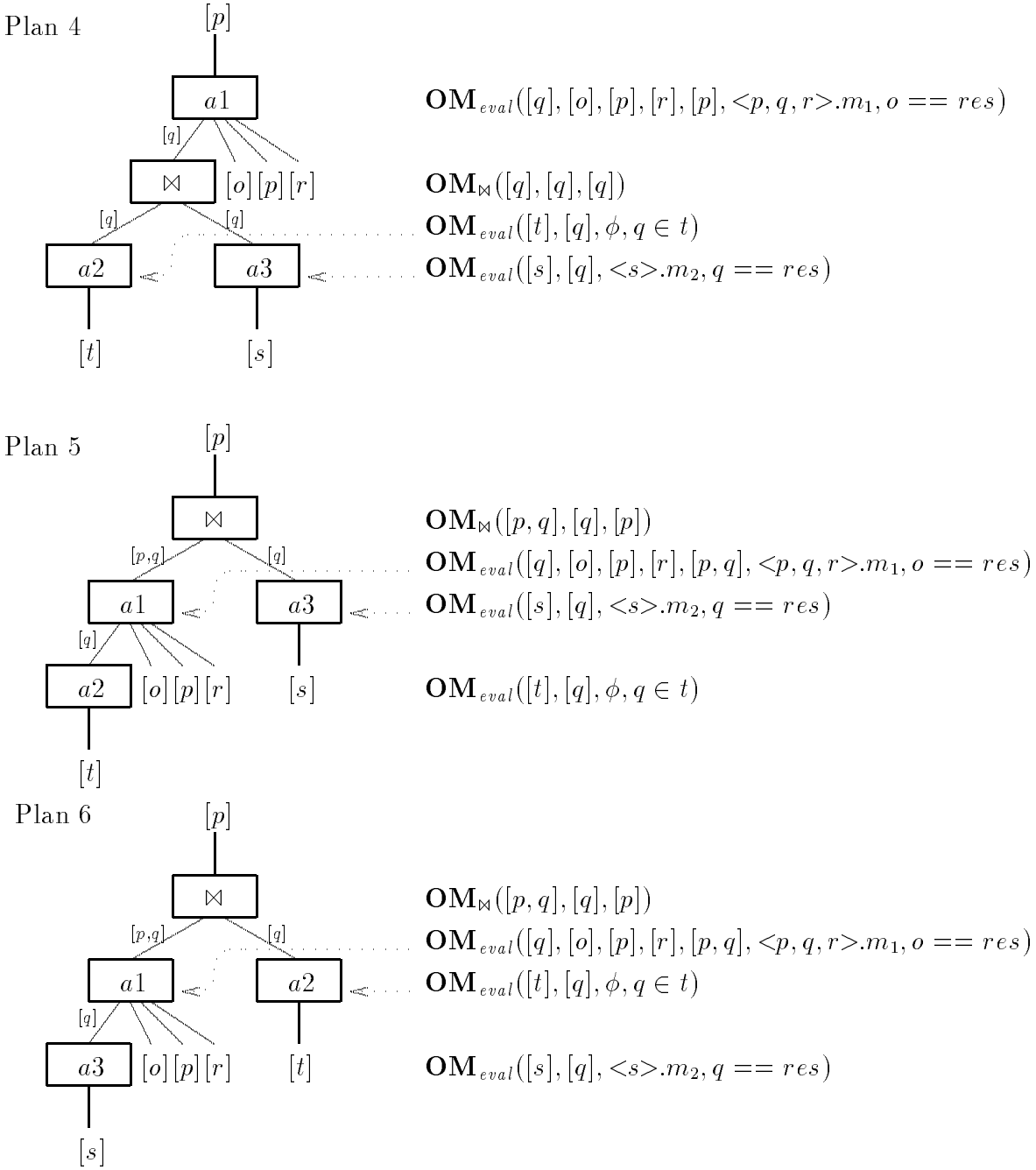


Figure 7.4: More access plans for the sample select operation.

Plan 1: (Figure 7.3) In this plan, the first OM operation[2] consumes atom $a2$ and creates an output oid-stream which contains those variables required by the remaining unconsumed atoms $(a1, a3)$. Since t is not referenced by either $a1$ or $a3$ it can be omitted from the output oid-stream. It is important to realize, though, that the OM operation to do this (designated by $a2$ in the diagram) requires taking the cross product $O \times P \times R \times S \times T$. In other words, an OM operation always takes the cross product of all its input oid-streams prior to applying a method and evaluating a predicate. The next object manager call consumes atom $a3$ and drops variable s from its output since it is not required by the remaining unconsumed atom $(a1)$. The last object manager call consumes the remaining atom and includes only the target variable in its output stream.

In summary, variables which represent generated objects are added to the intermediate oid-stream upon generation, e.g., q. Variables in the intermediate oid-stream are dropped by the last object manager call which references them, e.g., t. The final object manager call includes only the target variable in its output.

Plans 2,3: (Figure 7.3) These two plans are similar to the first except that cross products of some variables are delayed until just before the object manager call which requires them. Since $a1$ is the only atom which references variables o, p and r, prior OM calls for atoms $a2$ and $a3$ need not carry through these variables without using them. Plans 2 and 3 differ only in which atom is chosen to generate values for q first.

Plan 4: (Figure 7.4) This plan exercises another interpretation of the semantics of conjunction: if two atoms generate values for the same variable, then the final set of values is the equi join of the separately generated values. This parallelization technique has been combined with delayed cross products to minimize the size of parameters to object manager calls. For example, if we had performed an early cross product as in plan 1, then the input to $q \in t$ $(a2)$ would be $[o, p, r, s, t]$ which has many more tuples than there are unique values of t.

Plans 5,6: (Figure 7.4) These two plans utilize the same semantic principles as in plan 4, only in a different fashion. As before, cross products are delayed as long as possible. The two atoms which generate values for q are performed in parallel. The output of only one of these atoms is combined with o, p and r to restrict values of p and to generate the oid-stream $[p, q]$. However, unlike earlier restrictions, both p and q are retained in the output stream of this object manager call. There are now two oid-streams containing values for q, $[q]$ and $[p, q]$. The final oid-stream representing the target variable p is created by performing an $\mathbf{OM}_{\bowtie}$ operation on the two streams and projecting over the p component.

[2]Recall that OM operator trees are evaluated bottom up.

Some generalizations about the relative merits and drawbacks of these access plans can be made. It would seem that 'just in time' cross product generation is beneficial as it reduces intermediate oid-stream sizes, both in cardinality and in arity. Cardinality is reduced since restrictions are performed on the oid-stream prior to subsequent cross products. Arity is reduced as new variables are introduced only when needed rather than at the start of the subquery. The argument for delayed cross product generation would be even stronger if the complexity of the cross product operator grew non-linearly with respect to either the number of fields or the stream length. The parallelization performed in plans 5 and 6 looks beneficial initially, but has several drawbacks. One is that values for q are not restricted as much as they are in all the other plans prior to being used in later operations, e.g., to restrict p. Second, the intermediate oid-streams are larger since q must be carried through until the final $\mathbf{OM}_{\bowtie}$ operation. Plans 2 and 3 are similar to 4 in that they perform all restrictions on q as early as possible. Their relative merits would depend on the efficiency of parallel operations followed by an $\mathbf{OM}_{\bowtie}$ operation versus the sequential implementation.

Creating Select/Generate Access Plans

This section develops an algorithm for generating type 2 and 3 plans; i.e., delayed cross products combined with early restriction. The algorithm takes three inputs: (1) a set of atoms corresponding to a predicate which has been rewritten using the simplification rules presented earlier, (2) a set of variable names identifying inputs to the object algebra level operation, and (3) the name of the target variable. Output is an access plan graph. The algorithm uses a hypergraph [Ber73] representation of the predicate. The hypergraph contains one node for each unique variable name referenced in the atoms of the predicate and is initialized with an edge for each atom which covers all nodes corresponding to variables referenced in the atom. (Note that edges in a hypergraph define subsets of its nodes.) The nodes are marked as either red or green. A green node indicates that values for this variable exist, either because the variable is a member of the algorithm's second input parameter or because an object manager call has generated the values. A red marking indicates that values do not exist, i.e., the variable may not be used yet. The node markings are initialized to reflect the variables which represent inputs to the object algebra operation. The initial hypergraph corresponding to Equation 7.1 is shown in Figure 7.5-A. The algorithm proceeds by successively placing into the access plan graph $\mathbf{OM}_{eval}$ operations for atoms (hypergraph edges) until all atoms have been placed. An atom is eligible for placement in the access plan graph if all the nodes in its corresponding edge are green, or only one node is red but it represents a variable whose values are generated by the atom. The complete algorithm is given below.

Algorithm 7.2 *Create access plan for Select/Generate operators*
Inputs:

1. set of simplified atoms $a_1, \ldots, a_n$
2. set of input variable names V
3. target variable name t

Output: Graph of object manager operations G

begin
let H be the initialized hypergraph for $a_1, \ldots, a_n$ (1)
let G be the output access plan graph – initially empty (2)
let *stream_vars* represent variables in the intermediate oid-stream – initialized to ϕ (3)

while (there are edges in H) **begin** (4)
  let $a \in H$ be an edge (atom) eligible for placement (5)
    – all variables are green or one is red and generated by the atom, and
    – if *stream_vars* is non-empty a references all variables in *stream_vars*
  if ($stream\_vars = \phi$) **then** (6)
    let $i := 0$ (7)
  else (8)
    let $i := 1$ (9)
    let $in\_stream_i := stream\_vars$ (10)
  endif (11)
  for (all nodes $n_j \in a \mid n_j \in V \ \wedge \ n_j \notin stream\_vars$) **begin** (12)
    let $in\_stream_{(++i)} := n_j$ (13)
  endfor (14)
  let *out_stream* contain any variables which edge a shares with other (15)
    edges in H and t iff t is a member of any *in_stream* (16)
  add $\mathbf{OM}_{eval}([in\_stream_1], \ldots, [in\_stream_i], [out\_stream], meth, pred)$ to G where: (17)
    method *meth* and predicate *pred* reflect atom a as per Table 7.3 (18)
  if (this is not the first atom to be placed) **then** (19)
    connect in_stream_1 to *out_stream* of node placed on the previous pass (20)
  endif (21)
  let $stream\_vars := out\_stream$ (22)
  color all nodes $n_j \in a$ green (23)
  remove edge a from H (24)
endwhile (25)
end Algorithm 7.2

Example 7.3 We apply Algorithm 7.2 to the predicate of Equation 7.1. The result of each pass through the **while** loop is shown in Figure 7.5. Figure 7.5-A shows the initialized hypergraph prior to running the algorithm. Note that the node for q is red while all others are green. During the first pass through the **while** loop both atoms $a2$ and $a3$ are eligible for placement because all but one node in their respective hypergraph edges are green and each atom generates values for the single red node. Atom $a1$ is ineligible at this point as it does not generate values for the red node. Assuming atom $a3$ is chosen at random, Figure 7.5-B shows the nodes and edges added to the output graph. At the end of the first pass, q is colored green since values now exist for it and the edge for atom $a3$ is removed from the hypergraph. The result of pass two is shown in Figure 7.5-C. Both remaining atoms are eligible for placement and we assume atom $a1$ was randomly chosen. The out-stream of the

corresponding **OM** call is $[q, p]$ because (1) atom $a1$ overlaps with $a2$ on q, and (2) p is the target variable. Pass three places the remaining atom, $a3$. $\diamond$.

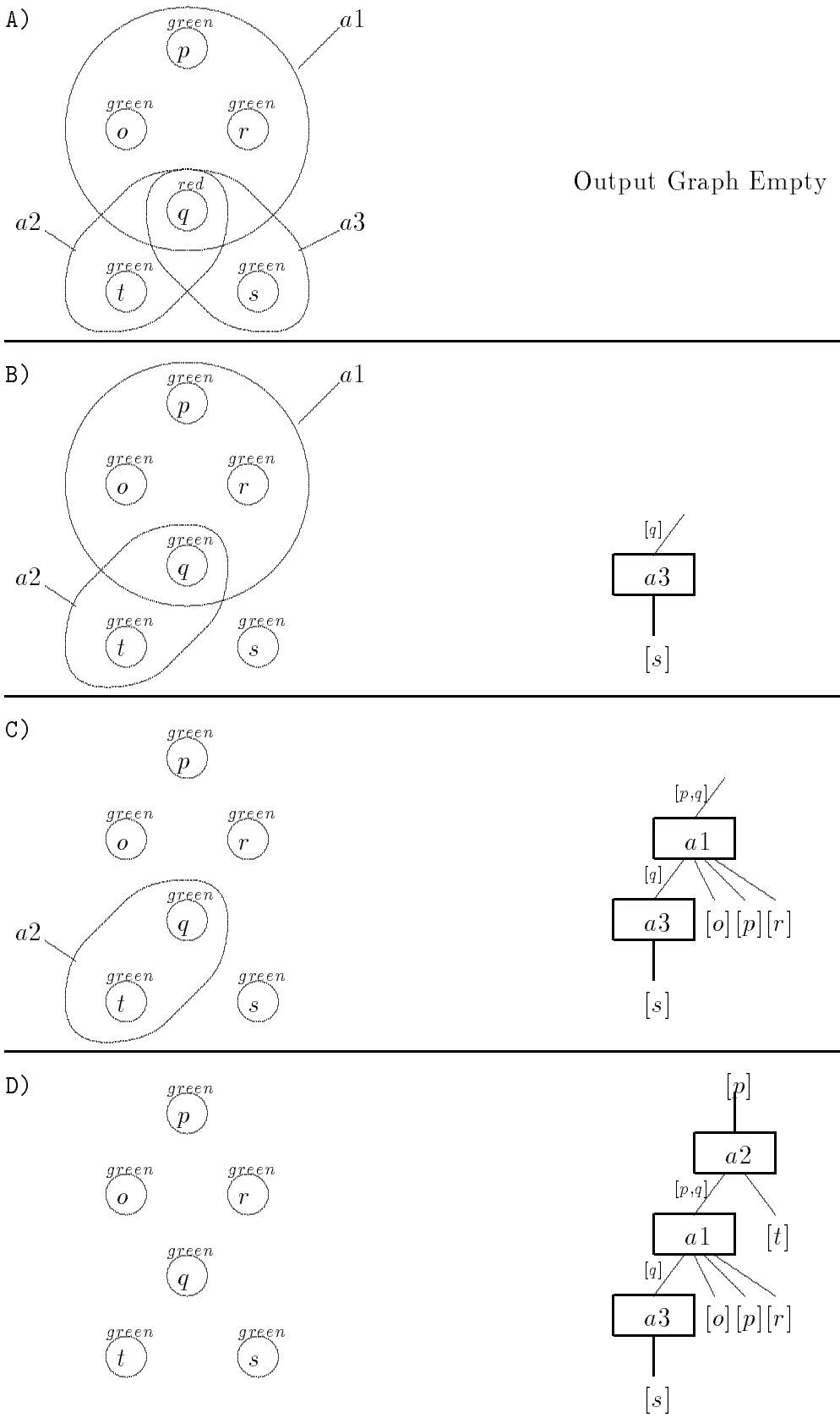


Figure 7.5: Building an access plan using Algorithm 7.2.

7.4 Select/Generate Access Plans Revisited

The previous section introduced the notion of an access plan as a tree of object manager operations. Queries expressed as trees of object algebra operators are converted to access plans by mapping each operator in the algebra tree to a corresponding subgraph of object manager operations. Algorithms were developed to perform this mapping for union and difference (Section 7.3.1), map (Section 7.3.2), and the select and generate operators (Section 7.3.3).

This section examines the mapping process for select and generate operators in more detail. Algorithm 7.2 is quite limited in that it can only generate access plans which are a linear sequence of $\mathbf{OM}_{eval}$ operations. Specifically:

- only one access plan is generated,
- the ordering of multiple eligible OM operations is determined by random choice and does not allow a cost based analysis of different orderings,
- object manager operations are never performed in parallel, and
- $\mathbf{OM}_{\bowtie}$ is not used to reduce intermediate oid-streams.

Ideally one would like to generate a family of access plans from which a best plan can be chosen based on some cost criteria. With respect to the select predicate of Equation 7.1, this would imply that the algorithm generate all feasible plans including those of Figures 7.3, 7.4 and Example 7.3.

To assist in an exhaustive generation of access plans, the notion of a *join template* [RR82] is extended to define a *processing template*. A processing template represents a family of logically equivalent access plans and is used as an intermediate formalism in mapping object algebra query trees to access plan graphs. A processing template for the predicate of Equation 7.1 is given in Figure 7.6.

A processing template consists of two types of nodes: *stream nodes* and *operator nodes*. Stream nodes (drawn as rectangles in Figure 7.6) represent intermediate results in a tree of object manager operations, i.e., access plan. In other words, stream nodes reflect the variables present in an intermediate oid-stream and the atoms which were evaluated to produce them. Since there are conceivably many ways to produce equivalent oid-streams, each stream node in the processing template represents an *equivalence class* of oid-streams.

Each stream node has two fields. The top field denotes the object variables present in the oid-stream. The bottom field denotes which atoms have been evaluated in order to create the oid-stream, but does not indicate the order in which the atoms were evaluated. We will refer to these atoms as being *consumed* by the stream node.

Operator nodes (drawn as circles in Figure 7.6) denote the $\mathbf{OM}_{eval}$ or $\mathbf{OM}_{\bowtie}$ operations in a select or generate access plan. As in the access plans of Section 7.3, an operator node is labeled with an atom number ($a1$, $a2$, etc.) if it corresponds to a $\mathbf{OM}_{eval}$ operation and with $\bowtie$ if it is a stream reduction operation.

Stream nodes with no consumed atoms, i.e., the leaf nodes, represent the original input streams of an object algebra select or generate operator. We define the *final*

node as the stream node in the processing template whose variables field contains just the target variable of the object algebra operator and whose atoms consumed field contains all the atoms in the object algebra operator's simplified predicate. The final node is always node 0.

Edges represent the flow of tuples from one operator node to the next. Individual access plans are represented by connected subgraphs of the processing template which cover all leaf nodes and the final node.

Referring to Figure 7.6, nodes 1 through 5 represent the original input streams to the algebra operation of Equation 7.1 and node 0 represents the final result. Node 6 is the result of the object manager operation $\mathbf{OM}_{eval}([s],[q],<s>.m_2, q == res)$ and node 7 is the result of $\mathbf{OM}_{eval}([t],[q],\phi, q \in t)$. Each of these nodes represents an equivalence class of size one as they each only have one input. Node 8 represents an equivalence class with three members. Using oid-streams subscripted with their processing template node numbers to indicate their source, the following OM calls all create the equivalent output denoted by stream node 8.

$$\mathbf{OM}_{eval}([q]_6,[t]_5,[q]_8,\phi, q \in t)$$
$$\mathbf{OM}_{eval}([q]_7,[s]_4,[q]_8,<s>.m_2, q == res)$$
$$\mathbf{OM}_{\bowtie}([q]_6,[q]_7,[q]_8)$$

Careful examination of the diagram will reveal that it includes all of the access plans of Figure 7.3 and 7.4. As an example, solid edges in the processing template of Figure 7.7 correspond to the access plan shown in the same diagram.

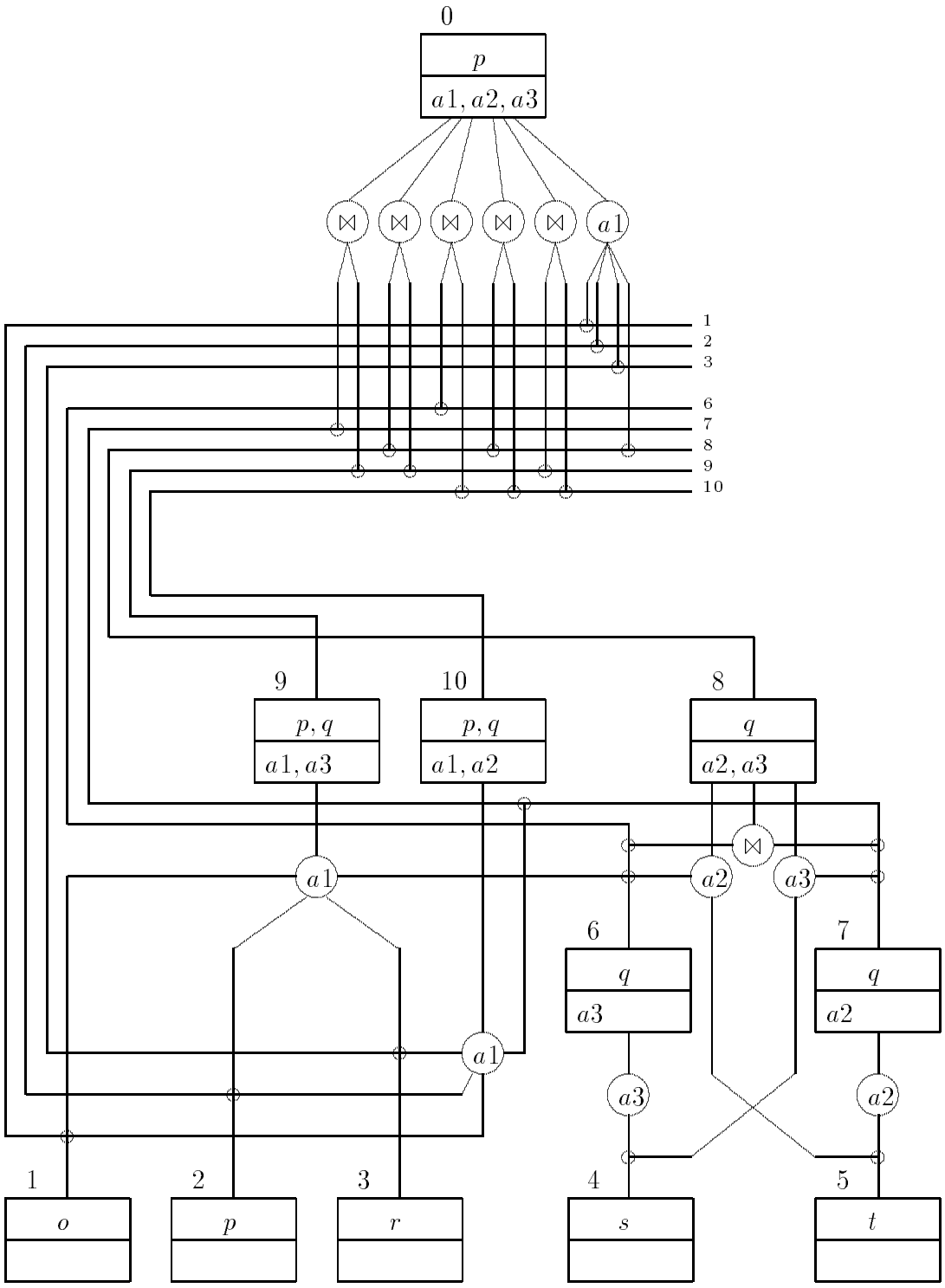


Figure 7.6: Processing template for the predicate of Equation 7.1.

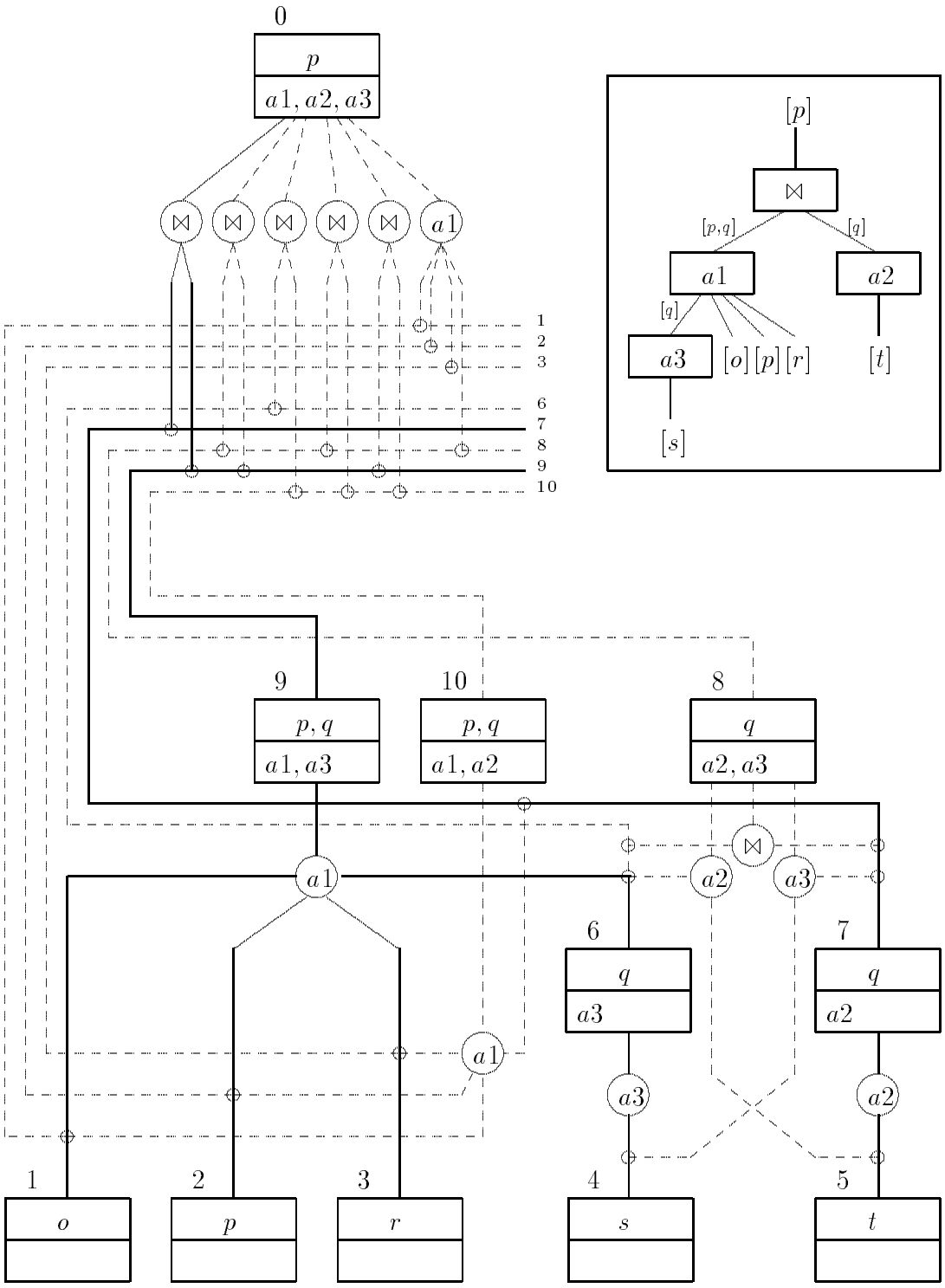


Figure 7.7: Relationship between a processing template and an access plan.

The goal then is to develop an algorithm which, given a select or generate operation in the object algebra, returns a processing template which enumerates all possible access plan graphs for that operation. The algorithm is first described using an extended example and then a more formal definition is provided.

7.4.1 Extended Processing Template Example

This example will show how the processing template is developed for the object algebra operation of Equation 7.1. An initial processing template is created by identifying the input streams of the algebra operation and placing nodes for each of them. In the example the input streams are $[o]$, $[p]$, $[r]$, $[s]$ and $[t]$ corresponding to nodes 1, 2, 3, 4 and 5 respectively. The final node, node 0, is also placed in the processing template. Its variables field contains either the variable being restricted in the case of a select operation or the target variable in the case of a generate operation. The example operation is a selection on the input set P, thus p is placed in the final node. Similarly, all atoms of the reduced predicate (a1,a2,a3) are placed in the final node's consumed atoms field.

Once the initial processing template is created, the following steps are repeated until it is no longer possible to create any new stream nodes. Each iteration of the following steps is referred to as a *pass* through the algorithm.

Pass 1

Recall that processing template stream nodes represent oid-streams which can be combined to evaluate atoms or to remove duplicates. The first step of each pass then, is to enumerate all possible ways of combining stream nodes. The algorithm given in [OL88] for join enumeration is modified slightly such that it does not produce combinations where a stream node is combined with itself (self-join). The final node is not included in the enumeration. Enumeration of the initial processing template results in the following permutations of stream nodes. Each permutation is shown as a set of node numbers and the sets are organized by size.

```
1: {1} {2} {3} {4} {5}
2: {1,2} {1,3} {2,3} {1,4} {2,4} {3,4} {1,5} {2,5} {3,5} {4,5}
3: {1,2,3} {1,2,4} {1,3,4} {2,3,4} {1,2,5} {1,3,5} {2,3,5} {1,4,5}
   {2,4,5} {3,4,5}
4: {1,2,3,4} {1,2,3,5} {1,2,4,5} {1,3,4,5} {2,3,4,5}
5: {1,2,3,4,5}
```

Similar to the filtering process described in [OL88], each permutation is tested to determine whether it is a *useful* combination of stream nodes. The filtering process is described next.

Each permutation of stream nodes defines mappings to sets of variables and sets of consumed atoms. For example the permutation {1,2,5} defines the mapping shown in Figure 7.8.

We define two interesting types of mappings:

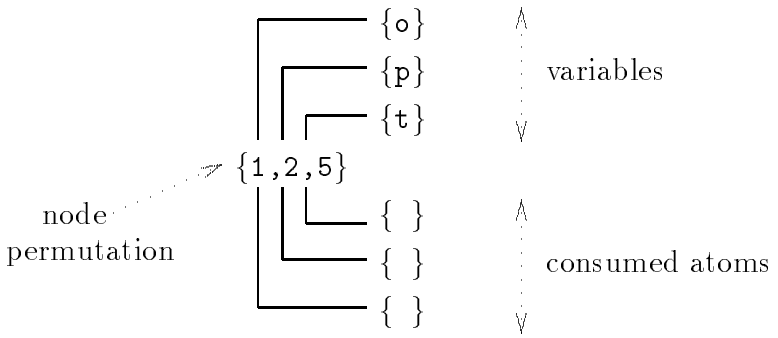


Figure 7.8: Mappings from stream node numbers to variables and atoms.

1. The variable sets are disjoint and, together, all the variables exactly match those required by an atom which has not been consumed by any of the nodes in the permutation. In other words, an unused atom can be consumed using exactly those streams represented by the nodes in the permutation. We can now create a third mapping from the permutation under consideration to a set of atoms which can be consumed by the combination of streams in the permutation. Figure 7.9 shows this extended mapping for three permutations of the first pass.

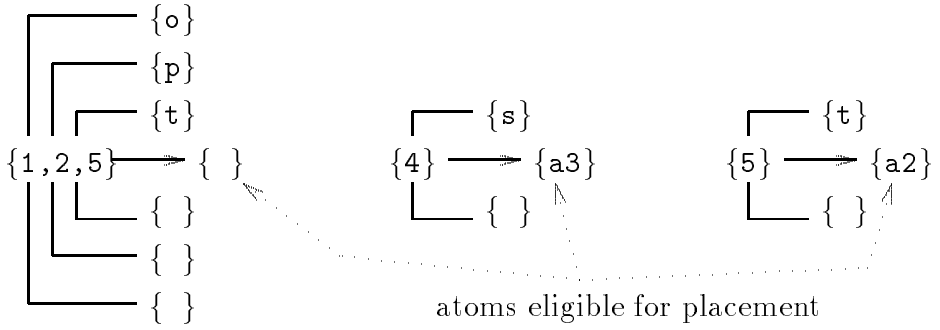


Figure 7.9: Extended mappings from stream node numbers to consumable atoms.

The first permutation, {1,2,5}, does not meet our criteria while the other permutations do. Although the variables which {1,2,5} maps to are disjoint, they do not exactly match the variables required by an unconsumed atom. The third mapping is to a set of atoms ({a3}) as opposed to a single atom (a3) since it is possible that several atoms can be consumed using the combination of streams in the permutation under consideration.

For each permutation with a non-empty set of eligible atoms, we consume each atom in the set by adding a stream node and operator node with appropriate connections to the processing template. In Pass 1, permutation {4} leads to the placement of stream node 6 and the operator node labeled a3 while permutation {5} leads to placement of stream node 7 and the operator node labeled a2. No other placements are possible. Since each stream node in the processing

template represents an equivalence class of oid-streams, we do not always place a new stream node. If a stream node already exists with the appropriate set of variables and consumed atoms, only the operator node is added and the appropriate connections made.

2. One or more variables are replicated in each of the variable sets. Discussion of this case is deferred to Pass 2 since the condition does not occur during Pass 1 in this example.

At the end of the first pass the processing template consists of stream nodes 0–7 and the OM operations which connect them. Appendix B summarizes the additions made during each pass.

Pass 2

Pass 2 begins by again enumerating all possible combinations of stream nodes. However, since the contents of a stream node are not modified after it is initially added to the processing template (only new connections are made), we only need to enumerate all *new* permutations of stream nodes which were not considered in any of the previous passes. This results in the following new permutations.

```
1: {6} {7}
2: {1,6} {2,6} {3,6} {4,6} {5,6} {1,7} {2,7} {3,7} {4,7} {5,7} {6,7}
3: {1,2,6} {1,3,6} {2,3,6} {1,4,6} {2,4,6} {3,4,6} {1,5,6} {2,5,6} {3,5,6}
   {4,5,6} {1,2,7} {1,3,7} {2,3,7} {1,4,7} {2,4,7} {3,4,7} {1,5,7} {2,5,7}
   {3,5,7} {4,5,7} {1,6,7} {2,6,7} {3,6,7} {4,6,7} {5,6,7}
4: {1,2,3,6} {1,2,4,6} {1,3,4,6} {2,3,4,6} {1,2,5,6} {1,3,5,6} {2,3,5,6}
   {1,4,5,6} {2,4,5,6} {3,4,5,6} {1,2,3,7} {1,2,4,7} {1,3,4,7} {2,3,4,7}
   {1,2,5,7} {1,3,5,7} {2,3,5,7} {1,4,5,7} {2,4,5,7} {3,4,5,7} {1,2,6,7}
   {1,3,6,7} {2,3,6,7} {1,4,6,7} {2,4,6,7} {3,4,6,7} {1,5,6,7} {2,5,6,7}
   {3,5,6,7} {4,5,6,7}
5: {1,2,3,4,6} {1,2,3,5,6} {1,2,4,5,6} {1,3,4,5,6} {2,3,4,5,6} {1,2,3,4,7}
   {1,2,3,5,7} {1,2,4,5,7} {1,3,4,5,7} {2,3,4,5,7} {1,2,3,6,7} {1,2,4,6,7}
   {1,3,4,6,7} {2,3,4,6,7} {1,2,5,6,7} {1,3,5,6,7} {2,3,5,6,7} {1,4,5,6,7}
   {2,4,5,6,7} {3,4,5,6,7}
6: {1,2,3,4,5,6} {1,2,3,4,5,7} {1,2,3,4,6,7} {1,2,3,5,6,7} {1,2,4,5,6,7}
   {1,3,4,5,6,7} {2,3,4,5,6,7}
7: {1,2,3,4,5,6,7}
```

As before, we build the mappings of stream node permutation to variables and consumed atoms and apply the filtering criteria. In this pass, both types of mappings which we consider interesting occur.

1. Mapping type 1 – the variable sets are disjoint and together, all variables exactly match those required by an atom which has not been consumed by any of the stream nodes in the permutation. These criteria are met by permutations {4,7}, {5,6}, {1,2,3,6} and {1,2,3,7}. The mapping to variables and consumed

atoms for two[3] of these permutations are shown in Figure 7.10. Referring to

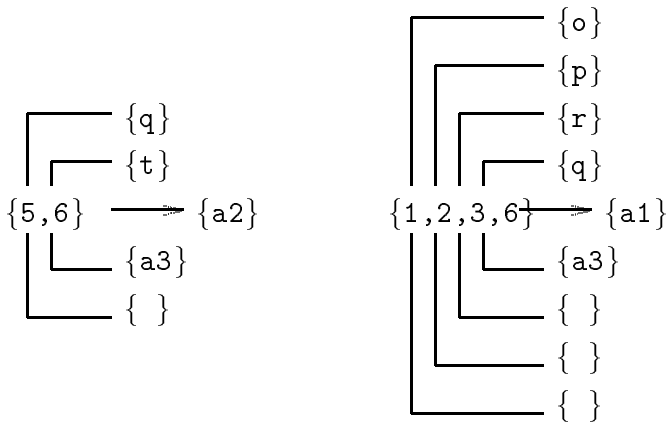


Figure 7.10: Node permutations to eligible atom mappings for pass 2.

Appendix B we see that in order to consume atom $a2$ using the oid-streams from nodes 5 and 6, node 8 must be created. However, permutation {4,7} would result in an identical stream node, thus we just make the connections to node 8 rather than create a new stream node when processing permutation {4,7}. This maintains the notion of a stream node representing an equivalence class of oid-streams.

Permutations {1,2,3,6} and {1,2,3,7} also meet our criteria and result in the creation of nodes 9 and 10 respectively.

2. Mapping type 2 – one or more variables are replicated in each of the variable sets. This condition means that several stream nodes exist with values for the same variable(s) and that an $\mathbf{OM}_{\bowtie}$ operation can be used to combine and reduce the oid-streams. Permutation {6,7} meets this criteria for variable q as shown in Figure 7.11.

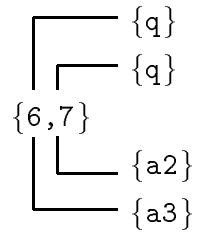


Figure 7.11: Node permutation with replicated variables.

Each stream node in the permutation represents values for variable q generated by a different set of atoms. In other words, node 6 represents values for q generated by atom $a3$ while node 7 represents values for q generated by atom

[3]Picked for illustrative purposes only.

$a2$. The nodes are joined by an $\mathbf{OM}_{\bowtie}$ operation and all variables required by unconsumed atoms are carried through to the output oid-stream. In this case, the output oid-stream would contain only the variable q and would have consumed atoms $a2$ and $a3$. Since this is equivalent to node 8, we only add the $\mathbf{OM}_{\bowtie}$ operator node and make connections to node 8 rather than create an entirely new stream node.

Pass 3

Enumeration of all previously unevaluated stream nodes results in the following interesting permutations: {1,2,3,8}, {7,9}, {8,9}, {6,10}, {8,10} and {9,10}. All of these permutations cause insertion of operator nodes only and do not cause any new stream nodes to be added to the processing template. The first permutation consumes an atom resulting in the placement of a $\mathbf{OM}_{eval}$ operation while all others result in $\mathbf{OM}_{\bowtie}$ operations. A further criteria is applied to the $\mathbf{OM}_{\bowtie}$ creating permutations which was not mentioned earlier.

Each of the stream nodes in the permutation must add to the consumed atoms field of the result. For example, permutation {6,9} is not acceptable as all of node 6's consumed atoms ({a3}), are already represented in those of node 9 ({a1,a3}). Appendix B summarizes the connections added in this pass.

The algorithm terminates after Pass 3 because no new stream nodes were created in this pass. In other words, enumerating all stream node combinations again will not result in any permutations which were not evaluated previously.

The full sequence of steps for exhaustive access plan enumeration is given in Algorithm 7.3 below.

Algorithm 7.3 *Build Select/Generate Processing Template*
Inputs:

1. set of simplified atoms $a_1, \ldots, a_n$
2. set of input variable names V
3. target variable name t

Output: Processing template, $Pt(V, E)$, enumerating multiple access plans

- Names denoting a set of items begin with an upper case letter, e.g., Pt is a set of processing template nodes.
- Names denoting a set of sets are completely upper case.
- All other names are lower case.

define $ENUM(i)$
to return the set of all permutations of numbers in $[1, \ldots, i]$. For example, $ENUM(3)$ returns the set {{1},{2},{3},{1,2},{1,3},{2,3},{1,2,3}}.

define $max\_node()$ to return the highest numbered node in Pt.

define $exists\_node(Var\_set, Atom\_set)$ to return the number of the node in Pt with Var_set and $Atom\_set$ in its 'variables' and 'atoms consumed' fields respectively, or -1 if such a node is not found.

define $add\_node(Var\_set, Atom\_set)$ to add a node with the given parameters to Pt. Nodes will be numbered consecutively starting at 0.

define $connect(From\_nodes, to\_node, using)$ to connect the nodes in $From\_nodes$ to to_node in Pt via the OM operation defined by $using$.

define $Retained\_vars(Var\_set, Atom\_set)$ to return the set of variable names which should be retained in the output stream after the variables in Var_set are used to consume or join the atoms in $Atom\_set$. This is identical to the technique used to determine which variables to retain in the intermediate stream used in Algorithm 7.2.

define $Find\_consumable\_atoms(Var\_set, Atom\_set)$ to return a set of atoms which can be consumed using the variables in Var_set and assuming the atoms in $Atom\_set$ have already been consumed. This is similar to the technique used in Algorithm 7.2 to pick an eligible atom only here all eligible atoms are returned rather than selecting just one atom at random.

begin

– initialization

let $Pt := \phi$ (1)
$add\_node(\{t\}, \{a_1, \ldots, a_n\})$ – final node of Pt is node 0 (2)
for (each $v \in V$) **begin** (3)
    $add\_node(\{v\}, \{\ \})$ – one node for each input stream (4)
endfor (5)
let $USED := \{\ \}$ – set of node permutation sets which have been evaluated (6)
let $continue := true$ – insure one pass through while loop (7)

– main algorithm

while ($continue$) **begin** (8)
    $continue := false$ (9)
    – $Node\_set$ denotes a set of node numbers such as $\{1, 3, 5\}$, i.e., a permutation
    for (each $Node\_set \in (ENUM(max\_node()) - USED)$) **begin** (10)
        let $USED := USED \cup Node\_set$ (11)
        – n denotes a node number in $Node\_set$
        – $n.Vars$ denotes the variables field of Pt node n
        – $n.Atoms$ denotes the atoms field of Pt node n
        let $U\_atoms := \{\ a \mid \exists n(n \in Node\_set \wedge a \in n.Atoms)\ \}$– atom union (12)
        let $U\_vars := \{\ v \mid \exists n(n \in Node\_set \wedge v \in n.Vars)\ \}$– variable union (13)
        let $I\_vars := \{\ v \mid \forall n(n \in Node\_set \Rightarrow v \in n.Vars)\ \}$– var intersection (14)

        if $(\forall n_1, n_2 \in Node\_set, (n_1.Vars \cap n_2.Vars) = \phi)$ **then** (15)
            – no two nodes in $Node\_set$ have any variables in common
            let $Atoms\_to\_consume := Find\_consumable\_atoms(U\_vars, U\_atoms)$(16)

```
            for (each a ∈ Atoms_to_consume) begin                          (17)
                let R_vars := Retained_vars(U_vars, (U_atoms ∪ a))         (18)
                if ((to_node := exists_node(R_vars, (U_atoms ∪ a))) = −1) then(19)
                    let to_node := add_node(R_vars, (U_atoms ∪ a))         (20)
                    let continue := true                                   (21)
                endif                                                      (22)
                connect(Node_set, to_node, OM_eval : a)                    (23)
            endfor                                                         (24)
        endif
        if (I_vars ≠ φ) then                                               (25)
            – there are variables common to all input streams
            if (∀a ∈ U_atoms(∃!n ∈ Node_set | a ∈ n)) then
                – each node contributes at least one new element to U_atoms(26)
                let R_vars := Retained_vars(U_vars, U_atoms)               (27)
                if ((to_node := exists_node(R_vars, U_atoms)) = −1) then(28)
                    let to_node := add_node(R_vars, U_atoms)               (29)
                    let continue := true                                   (30)
                endif                                                      (31)
                connect(Node_set, to_node, OM⋈)                            (32)
            endif                                                          (33)
        endif                                                              (34)
    endfor
endwhile
end Algorithm 7.3
```

7.4.2 Choosing the Cheapest Plan

Output of the enumeration algorithm described above is a processing template which identifies a family of logically equivalent query execution plans. Each connected subtree of edges in the processing template which includes all initial nodes and the final node is a valid plan. But which is the best plan?

Section 7.2 defined an object manager interface but our research does not address its implementation. An implementation design would be highly dependent on the object representation, the technique used to bind method code to objects and other system parameters. Thus, although a specific cost function is not proposed, we assume that the object manager is capable of using oid-stream statistics to derive a cost for calls to its interface.

Appropriate oid-stream statistics might be stream cardinality and information about the classes represented in the stream. For a given call, the object manager could derive a processing cost and statistics for the resulting output oid-stream. A processing template could then be annotated with cost information as follows.

Initially only leaf nodes (which are stream nodes) of the processing template would have stream statistics associated with them. If the leaf nodes correspond to the leaf nodes of the original object algebra query, then they represent the extent or deep extent of classes in the database and their statistics are readily available. Otherwise

the leaf nodes represent the output of a previous subtree of object manager calls and the output oid-stream statistics of the appropriate subtree are attached.

Working from leaf to root in the processing template, the object manager cost function is used to assign a processing cost to each operator node as well as a set of stream statistics for the stream node the operator feeds into. All operator nodes and stream nodes in the processing template can be annotated with cost and statistical information in this fashion. The total cost of any specific access plan within the processing template is the sum of the operator costs which are included in the access plan's subgraph. If time information is included in the cost function, then when operator nodes execute in parallel, only the longest running operator should be included in the sum.

Note that cost information can not be used to prune the search space of the processing template generation algorithm. The search space of the algorithm is defined by the number of stream nodes present in the processing template at the start of each pass. This value can only be affected by the criteria used to define the "interesting permutations" which cause new operator and stream nodes to be created.

Chapter 8

Conclusion

This chapter summarizes the results presented in the thesis, identifies the contribution and novelty of the work, and provides direction for future research in the area.

8.1 Summary of Research

This thesis developed a query processing methodology for object-oriented databases which parallels that for relational databases. Chapter 1 motivated the work by demonstrating how ad hoc object-oriented query mechanisms do not make the distinction between declarative and procedural forms of a query and are not optimized for performance. Relevant topics in advanced database modeling, query languages and query processing techniques were reviewed and their effect on this thesis was discussed. A brief summary of several prototype implementations which influenced the work was provided in Appendix A.

A formal object-oriented data model was presented in Chapter 2 which served as a foundation for the rest of the thesis. Objects are viewed as instances of abstract data types called classes. Classes define methods which provide the only access to their instances. Classes can be organized into a lattice based on behavioral inheritance. Databases are defined in terms of constraints on objects, classes and methods. Legal database operations consist of applying (sequences of) methods to objects. A database schema representing a hypertext application was developed for use in subsequent examples.

Chapter 3 presented an object calculus for declarative specification of queries. The object calculus is similar in form to the tuple relational calculus but differs in several fundamental respects. First, object calculus expressions define sets of single objects, not sets of tuples of objects. Second, any method defined on a class can be used to form predicates. Lastly, expression results need not range over sets explicitly referenced in the expression. The predicate subcomponents responsible for this were termed *generating atoms*. A query safety criterion was developed based on the properties of generating atoms.

Chapter 4 presented an object algebra for procedural specification of queries. The algebra is object preserving meaning that queries must return objects which already

exist in the database. Input and output to algebra operators are sets of objects. A restricted form of object calculus expressions was defined and an algorithm presented for translating restricted expressions in the calculus to trees of algebra operators.

A type consistency theory for object algebra expressions was developed in Chapter 5 which allows algebra operations to consume and produce non-homogeneous sets of objects. Type consistency was defined in terms of a set of type inference rules. The rules determine the result type of an algebra operation based on the input arguments' types. An algorithm for and determining type inclusion relationships was developed to support the inference rule mechanism.

Chapter 6 presented a suite of equivalence preserving rewrite rules for logical optimization of object algebra expressions. Two kinds of rules were developed: algebraic and semantic. The applicability conditions for algebraic rules are strictly syntactic meaning that a straightforward pattern matching mechanism can be used to drive a rule based optimizer. Semantic rules also have a syntactic applicability condition, but additionally require that some schema and query dependent conditions be met. These conditions might include the type consistency of alternate forms of subexpressions or satisfaction of inclusion relationships in the database schema.

Techniques to generate access plans for object algebra expressions were developed in Chapter 7. The interface and semantics of an object manager subsystem which implements primitive operations of the data model on streams of objects was initially defined. Next, algorithms were developed to map individual algebra operators to graphs of object manager operations. This first set of algorithms demonstrated the intuition behind the mapping process but produced inefficient access plans. A second algorithm was developed which uses an extension to relational join templates, the processing template, to enumerate all possible access plans for object algebra select and generate operations. Techniques for annotating the processing template with cost information and reducing the search space of the algorithm were discussed.

8.2 Contribution and Novelty

The primary contribution of this thesis is its formal approach to query models and query processing in object-oriented databases. Most developers of post-relational data models (semantic, $\neg$1NF, functional) have felt obliged to cover the same ground that was done for the relational model, e.g., precise model definition, formal query languages, query safety criterion, proofs about properties of the model. However, object-oriented database proponents have tended to either map a loosely defined data model to the relational or implemented their model in an ad hoc manner. This thesis demonstrates that a formal approach in the context of the object-oriented model is possible and develops several novel results.

1. The safety criterion of Chapter 3 is the only one known to us for systems which view objects as instances of abstract data types.

2. The type consistency theory of Chapter 5 is more complex, but significantly less restrictive than other typing constraints proposed to date. The discov-

ery that some combinations of predicates can increase rather than reduce type information is a new result.

3. The algebraic rewrite rules of Chapter 6 show that a rule based optimizer can be used for improving queries in an object-oriented database. The semantic rewrite rules are, to our knowledge, the first to use class schema information to modify queries.

4. The object manager interface of Chapter 7 combines system support for primitive object operations with stream operations in a unique fashion. Use of the processing template to enumerate logically equivalent families of access plans shows how relational plan optimization techniques can be adapted to object-oriented databases.

8.3 Directions for Future Research

The work presented in this thesis suggests many interesting areas for future research. One important area is to investigate how extensions to the data model can be integrated throughout the entire query processing methodology. For instance, primitive operations such as shallow and deep equality [KC86], additional predefined value types other than atomic, set or structural (e.g., tuple values [AGOP88]) or parametric types such as *Set[t]* [SZ90] would significantly enhance the usefulness of the model. Each addition to the basic data model must be propagated through the methodology of Figure 1.3. This means it must be incorporated into the calculus and algebra, type inference rules need to be developed, logical equivalences must be proven and the object manager interface must be extended. Performing this exercise for several extensions would provide insight into the tradeoffs between maintaining the proposed query processing methodology and completeness of the data model.

Improving the query languages is another important topic. The object calculus, while expressive, is not user friendly. Design of a user query language, perhaps an object SQL [Lyn88, Ont89], would enhance usability and uncover many programming language integration issues. The object algebra can be extended in two respects. The first is to provide support for object creating operations. This raises many philosophical as well as technical issues. For example, what is the class of an object created by such an operation and what methods are defined on it? Should such objects, and their new class, persist after execution of the query? The second extension to the object algebra involves support for universal quantification. This could be achieved by allowing quantification in predicates or by defining the algebra to operate on tuples of objects and providing a division operator similar to that of the relational algebra. Both approaches may affect the scope of transformations possible during logical optimization and the generation of access plans.

Designing an object manager implementation is another important area of research. Such a design must address many related issues such as object representation, physical partitioning of logical entities such as classes and their extents, object

buffering, indexes, and how and when method code is bound to objects. The design also is affected by the underlying hardware architecture, e.g., uni-processor or multi-processor, and the available operating system services.

The access plan generation scheme proposed in this thesis assumes that the logical transformation rules of Chapter 6 have been used to ameliorate the original object algebra query prior to plan generation. Plan generation then only replaces each individual algebra operator with a "best" subtree of object manager calls. In other words, the overall shape of the query tree remains that which was arrived at during logical optimization. An area of future research indicated by this methodology is the development of equivalence preserving rewrite rules for trees of object manager operations. Such rules would allow global optimization of the entire access plan as opposed to merely picking "best" subtrees. Another interesting topic would be to develop an access plan generation strategy which cycles back and forth between the logical algebra optimization phase and the access plan generation phase. This would allow interleaving transformations which change the shape of the query with the introduction of access plan subtrees possibly resulting in more efficient plans.

Appendix A

Survey of Implementations

This appendix briefly summarizes several object-oriented programming environments and database systems which are relevant to the work presented in this thesis.

A.1 Smalltalk-80

Smalltalk [GR85] is an object-oriented programming language which includes a language kernel, a programming paradigm and user interface. The first version, Smalltalk-72, was developed by Alan Kay as part of the Dynabook project at Xerox. Several versions have followed with Smalltalk-80 being the most common version in use today. All entities in a Smalltalk system (variables, complex structures, classes, literals, processes) are considered objects which are instances of some class. The programming paradigm includes "message sending" as a means to initiate actions, specialization via subclassing and inheritance.

Smalltalk's significance lies in its uniform, understandable approach and early existence. Detracting from its acceptance are its performance and closed, single user environment. Various schemes such as type checking [Joh86] and improved object management [Kae86, Sta84] have been proposed to improve performance. A distributed version [Dec86] provides multi-user capability but does not improve the interface to other programming environments.

A.2 POSTGRES

POSTGRES [RS87, SR86, Sto86] is a successor to the INGRES [Sto76] relational database system which supports ADTs, relation attributes of type procedure and some forms of inheritance. Other aspects of its design address versioning [Sto87] and a rule subsystem [SHH87].

Users can extend POSTGRES by defining new atomic data types using an ADT definition facility. Defining an ADT requires specification of its name, internal representation, default value and operators. User defined associativity and precedence of ADT operators is used to correctly parse queries. The ADT implementor must also

supply *sort*, *restrict* and *join* procedures which the query optimizer uses to maintain indices and to compute restrict and join selectivities.

Relation attributes may be any atomic data type (including ADTs) or procedures which are sequences of POSTQUEL query language commands. Procedures may be unique to a tuple or common to a relation. When defined on a relation the procedure references numerical identifiers (e.g., , $1, $2, etc.) which select attribute fields of the current tuple at run time. A relation may be defined to inherit relation schemes from other relations when it is initially declared.

Despite supporting data abstraction and inheritance POSTGRES is not considered an object-oriented database. The database implementor does not "see" a new data model and has the responsibility to properly (efficiently) map the application semantics to ADTs, relations and procedures in the context of the relational model. Finally, aside from utilizing some user defined characteristics of ADTs, traditional relational query processing techniques are used.

A.3 EXODUS

The EXODUS project is a toolkit approach to building and extending database systems [Car86]. It is a modular system intended to provide a collection of kernel facilities enabling semi-automatic generation of application specific DBMSs. The storage manager [CDRS86] provides concurrency control, recovery and versioning for arbitrary sized storage objects. The optimizer generator [GD87] builds a rule based query optimizer from query algebra specifications. A type system and the E programming language [CDV88, RC89], an extension to C++, provide the ability to define ADTs and their associated operations. A database implementor can use these tools to quickly build an application specific DBMS.

The EXODUS type system supports complex objects with identity and inheritance. The query language is based on QUEL [Sto76] with extensions for nested sets, aggregate functions and a unique mix of object and value oriented semantics. However, since QUEL is tuple oriented and based on the relational calculus, the underlying query processing is primarily relational in nature. The optimizer generator has not been applied to an object algebra at this time.

A.4 Starburst

Starburst is an extensible relational database which supports externally defined data types called EDTs[1] [HFLP89, WSSH88]. The main goal of the project is to efficiently process queries which use EDTs as opposed to significantly enhancing the relational data model.

Query processing is considered to have two distinct phases: query rewrite [HP88, OL88] and plan optimization [LFL88, Loh88]. Query rewrite translates queries to

[1] EDTs are similar to POSTGRES ADTs.

equivalent ones for better performance. Examples of useful rewrites are semantic transformations, elimination of redundant joins in queries with views, and predicate migration. Plan optimization refers to the more traditional cost based optimization activities such as determining the methods and order of joins and the methods for accessing tables.

Starburst implements both phases of query processing using rule based systems instead of embedding optimization strategies in algorithmic code. This allows the "how" of individual query manipulation steps to be specified independent of "when" they will be applied. The EDT implementor can specify transformations on expressions containing EDTs without concerning himself with the details of optimization. The database insures that any available transformations will be applied if they improve the query. Starburst's success in isolating transformation semantics while providing a generic rule engine to apply them allows other query processing studies such as this thesis to take a rule based approach.

A.5 Iris

Iris is a prototype object-oriented database developed by Hewlett-Packard Laboratories [DKL85, Fis87, Fis88, LK86]. Its data model definition was heavily influenced by Daplex [Shi81] and is based on three concepts: objects, types and functions. Functions can be defined in various ways. Stored functions are implemented as tables which map input values to result values. Derived functions are specified in terms of other functions. Foreign functions are implemented in a general purpose programming language and allow access to underlying file systems, specialized storage managers and algorithmic generation of data. The storage subsystem is currently built on top of Allbase, Hewlett-Packard's relational DBMS. This subsystem is similar to RSS of System R and supports transactions, concurrency control, recovery and indexing.

Iris provides three interfaces: an object SQL [Lyn88], a graphical browser and a procedural interface. The object SQL adds to relational SQL (1) the ability to directly reference objects in addition to keys, and (2) the ability to use functions in `WHERE` and `SELECT` statements. `SELECT` statements return tuples of values and/or object identifiers. Predicates are a conjunction of terms consisting of function calls, object references, literal values and relational operators such as $=$, $<$, etc. The relational operators are defined only on the types `Integer`, `Real`, `Bitstring` and `Charstring` thereby insuring that user defined (abstract) types may interpret their value domains as required by the application. Despite the completeness of object SQL as an OODB query facility, the requirement to map queries to a relational subsystem has precluded investigating new methods for evaluating queries in an object-oriented database.

A.6 GemStone

GemStone is an object-oriented database system based on Smalltalk [CM84, MSOP86, PS87]. Several extensions have been made in order to create a multi-user, persistent

environment. GemStone is a two component architecture where Stone roughly corresponds to the object memory and Gem corresponds to the virtual machine of the standard Smalltalk implementation. Multiple Gem processes can exist concurrently, each managing a user interface and communicating with a shared Stone process. Concurrent users are supported by managing a shadow copy of the object space for each user session with conflicts reconciled by an optimistic concurrency control scheme.

One of GemStone's goals is to support efficient associative access via indices [MS86]. Indexing was chosen to be on object structure as opposed to how objects respond to messages since this could be supported at the Stone (object memory) level. In Smalltalk, object structure is defined by a set of named, untyped instance variables. However, indexing on instance variable names requires that every member of an indexed collection must have that index variable. Thus, instance variables in GemStone are typed and the inheritance model is extended to include inheritance of instance variable types as well as names.

The GemStone data manipulation language, OPAL, provides constructs for the general query mechanism described in Section 1.2 where a block of code is applied to each member of a collection. A rudimentary query sublanguage is also defined which allows specifying predicates as conjunctions of comparisons on index expressions [MS88]. This is very restrictive in that only indexable components of objects (i.e., , instance variables) can be referenced in a predicate. Arbitrary methods defined on a class may not be used.

A.7 O_2

The goal of the O_2 project is to combine database technology and programming language technology into a cohesive system using the object-oriented paradigm. In order to be language independent, O_2 provides only a type model [LRV88] and DDL while allowing methods associated with types to be written in a variety of languages. The type model includes set, list and tuple constructors for defining complex types. Inheritance is based on the set inclusion semantics developed in [Car84].

A first prototype has been demonstrated [Ban88b] and progress on components of a second version are reported in [VBD89]. A key feature of the second version is support for both a "development mode" and an "execution mode". Development mode is similar to the Smalltalk run time environment where method lookup is done dynamically and tuple attributes are accessed by name. In execution mode methods are loaded statically and tuple attributes (as well as other items) are accessed by physical offsets. These two modes are intended to maintain the desirable features of an interpretive development environment while providing the performance of a compiled environment when required.

An unpublished technical report is referenced in [VBD89] which proposes a query language for O_2. There is no indication whether this is a declarative or procedural language. However, it is likely that predicates are similar to those proposed in this thesis as the predefined operations on primitive types are similar. The O_2 type system supports value and identity equality operators on all types and set inclusion operators

on set types which are analogous to $=$, $==$, $\in$ and $=_{\{\}}$ developed in Chapter 3.

A.8 ORION

ORION is a full featured object-oriented database developed in Common Lisp at Microelectronics and Computer Technology Corporation (MCC) [Ban87, Kim87, Kim88]. It represents a bold attempt to integrate many object-oriented concepts into a single system: versions, composite objects [KBC87], schema evolution [BKKK87], transaction management [GK88] and predicate based queries [BKK88, KKD89, Kim89].

A query language was proposed in [BKK88] which supports arbitrary code blocks augmented with quantification as predicates. It was shown that a one-to-one mapping between relational queries and object-oriented queries is possible if only unary methods are allowed. However, the corresponding relational queries may be inefficient due to the joins required to reconstruct complex and set valued objects. A more detailed analysis of queries is given in [KKD89, Kim89] which classified queries by the manner in which evaluation traverses the class hierarchy. The concept of explicit and implicit joins between classes is also introduced.

Unlike the approach proposed in this thesis, ORION evaluates queries using the standard facilities for method evaluation built on top of the Common Lisp base. In this respect, although the system has contributed to understanding the nature of queries, it has not developed a new query processing paradigm.

Appendix B

QEP Algorithm Output

```
*** INITIAL processing_template ***
    1:  Variables: o
        Atoms:
    2:  Variables: p
        Atoms:
    3:  Variables: r
        Atoms:
    4:  Variables: s
        Atoms:
    5:  Variables: t
        Atoms:
Final:  Variables: p
        Atoms: 1,2,3
*** Pass # 1 ...
Adding node 6
        Variables: q
        Atoms: 3
Connecting node(s) 4 to node 6 using atom a3
Adding node 7
        Variables: q
        Atoms: 2
Connecting node(s) 5 to node 7 using atom a2
*** Pass # 2 ...
Adding node 8
        Variables: q
        Atoms: 2,3
Connecting node(s) 5,6 to node 8 using atom a2
Connecting node(s) 4,7 to node 8 using atom a3
Connecting node(s) 6,7 to node 8 via equi-join
Adding node 9
        Variables: q,p
```

```
        Atoms: 1,3
Connecting node(s) 1,2,3,6 to node 9 using atom a1
Adding node 10
        Variables: q,p
        Atoms: 1,2
Connecting node(s) 1,2,3,7 to node 10 using atom a1
*** Pass # 3 ...
Connecting node(s) 7,9 to node 0 via equi-join
Connecting node(s) 8,9 to node 0 via equi-join
Connecting node(s) 6,10 to node 0 via equi-join
Connecting node(s) 8,10 to node 0 via equi-join
Connecting node(s) 9,10 to node 0 via equi-join
```

Bibliography

[AB84] S. Abiteboul and N. Bidoit. An Algebra for Non Normalized Relations. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 191–200, March 1984.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[Agh87] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M.I.T. Press, 1987.

[AGOP88] A. Albano, F. Giannotti, R. Orsini, and D. Pedreschi. The Type System of Galileo. In M. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 101–119. Springer Verlag, 1988.

[AH84] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 119–132, 1984.

[AMY88] R. M. Akscyn, D. L. McCracken, and E. A. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge Organizations. *Communications of the ACM*, 31(7):820–835, July 1988.

[ASL89] A. Alashqur, S. Su, and H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. In *Proc. 15th International Conference on Very Large Databases*, pages 433–442, 1989.

[Ast76] M. Astrahan. System R: A Relational Approach to Data. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[Ban87] J. Banerjee, et. al. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Ban88a] F. Bancilhon. Object-Oriented Database Systems. In *Proc. of the 7th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 152–162, August 1988.

[Ban88b] F. Bancilhon, et. al. The Design and Implementation of O_2, an Object-Oriented Database System. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 1–22. Springer Verlag, 1988.

[Ber73] C. Berge. *Graphs and Hypergraphs.* North-Holland, 1973.

[BHJ+87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[BHJL86] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 78–86, September 1986.

[BK86] F. Bancilhon and S. Khoshafian. A Calculus for Complex Objects. In *Proc. of the 5th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 53–59, 1986.

[BKK88] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. In *Proc. 4th Int'l. Conf. on Data Engineering*, pages 31–38, February 1988.

[BKKK87] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 311–322, May 1987.

[Bor88] A. Borgida. Class Hierarchies in Information Systems: Sets, Types, or Prototypes. In M. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistance*, chapter 10, pages 137–154. Springer Verlag, 1988.

[Bry89] F. Bry. Logical Rewritings for Improving the Evaluation of Quantified Queries. In J. Demetrovics and B. Thalheim, editors, *2nd Symposium on Mathematical Fundamentals of Database Systems*, volume 364 of *Lecture Notes in Computer Science*, pages 100–116. Springer Verlag, 1989.

[Bus45] V. Bush. As We May Think. *Atlantic Monthly*, 176(1):101–108, July 1945.

[Car84] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer Verlag, 1984.

[Car86] M. Carey, et. al. The Architecture of the EXODUS Extensible DBMS: A Preliminary Report. Technical Report 644, University of Wisconsin, Computer Sciences Department, May 1986.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shetika. Object and File Management in the EXODUS Extensible Database System. In *Proc. 12th International Conference on Very Large Databases*, pages 91–100, August 1986.

[CDV88] M. Carey, D. DeWitt, and S. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 413–423, June 1988.

[CG88] B. Campbell and J. M. Goodman. HAM: A General Purpose Hypertext Abstract Machine. *Communications of the ACM*, 31(7):856–861, July 1988.

[Che76] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 316–325, August 1984.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod71] E. F. Codd. Relational Completeness of Data Base Sublanguages. In *Courant Computer Science Symposium on Data Base Systems*, volume 6, pages 65–98. Prentice-Hall, May 1971.

[Cod79] E. Codd. Extending The Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, December 1979.

[Con87] J. Conklin. Hypertext: An Introduction and Survey. *Computer*, 20(9):17–41, September 1987.

[CP89] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 433–443, 1989.

[CW85] L. Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dat87] C. J. Date. *A Guide to the SQL Standard.* Addison Wesley, June 1987.

[Dec86] D. Decouchant. Design of a Distributed Object Manager for the Smalltalk-80 System. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 444–452, September 1986.

[DKL85] N. Derrett, W. Kent, and P. Lyngbaek. Some Aspects of Operations in an Object-Oriented Database. *Quart. bull. of the IEEE TC on Data Engineering*, 8(4):66–75, December 1985.

[DLA88] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds Distributed Operating System. In *Proc. 8th Int'l. Conf. on Distributed Computing Systems*, pages 2–17, June 1988.

[DT88] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[EE87] A. Ege and C. A. Ellis. Design and Implementation of GORDION, an Object Base Management System. In *Proc. 3rd Int'l. Conf. on Data Engineering*, pages 226–234, May 1987.

[Fag82] R. Fagin. Horn Clauses and Database Dependencies. *Journal of the ACM*, 29(4):952–985, October 1982.

[Fis87] D. H. Fishman, et. al. Iris: An Object-Oriented Database Management Systems. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Fis88] D. H. Fishman. An Overview of the Iris Object-Oriented DBMS. In *Proc. of COMPCIN 33rd IEEE Computer Society Int'l Conference*, pages 177–180, March 1988.

[Fre87] J. Freytag. A Rule-Based View of Query Optimization. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 173–180, 1987.

[GD87] G. Graefe and D. DeWitt. The EXODUS Optimizer Generator. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 160–172, May 1987.

[GK88] J. F. Garza and W. Kim. Transaction Management in an Object-Oriented Database System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 37–45, September 1988.

[Gou84] M. Goudran. *Graphs and Algorithms.* John Wiley & Sons, 1984.

[GR85] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation.* Addison Wesley, 1985.

[GT87] A. Van Gelder and R. W. Topor. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. of the 6th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 313–327, 1987.

[Hal88] F. G. Halasz. Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 377–388, June 1989.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HO87] D. Halbert and P. O'Brien. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, pages 71–79, September 1987.

[HP88] W. Hasan and H. Pirahesh. Query Rewrite Optimization in Starburst. Technical Report TR RJ 6367, IBM Almaden Research Center, August 1988.

[Hul87] R. Hull. A Survey of Theoretical Research on Typed Complex Database Objects. In J. Paredaens, editor, *Databases*, chapter 5, pages 193–256. Academic Press, 1987.

[HY84] R. Hull and C. Yap. The Format Model: A Theory of Database Organization. *Journal of the ACM*, 31(3):518–537, July 1984.

[HZ87] M. F. Hornick and S. B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1):70–95, January 1987.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):112–152, June 1984.

[Joh86] R. Johnson. Type-Checking Smalltalk. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 315–321, 1986.

[JS82] G. Jaeschke and H. Schek. Remarks on the Algebra of Non First Normal Form Relations. In *Proc. of the 1st ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 124–137, 1982.

[Kae86] T. Kaehler. Virtual Memory on a Narrow Machine for an Object-Oriented Language. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 87–106, September 1986.

[KBC87] W. Kim, J. Banerjee, and H. Chou. Composite Object Support in an Object-Oriented Database System. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 118–125, 1987.

[KC86] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 406–416, September 1986.

[Ken79] W. Kent. Limitations of Record-Based Information Models. *ACM Transactions on Database Systems*, 4(1):107–131, March 1979.

[Kim87] W. Kim, et. al. Features of the Orion Object-Oriented Database System. Technical Report ACA-ST-308-87, Microelectronics and Computer Technology Corporation, September 1987.

[Kim88] W. Kim, et. al. Integrating an Object-Oriented Programming System with a Database System. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 142–152, September 1988.

[Kim89] W. Kim. A Model of Queries for Object-Oriented Databases. In *Proc. 15th International Conference on Very Large Databases*, pages 423–432, 1989.

[Kin89] R. King. My Cat is Object-Oriented. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 23–30. Addison Wesley, 1989.

[KKD89] K. Kim, W. Kim, and A. Dale. Cyclic Query Processing in Object-Oriented Databases. In *Proc. 5th Int'l. Conf. on Data Engineering*, pages 564–571, 1989.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[KV84] G. Kuper and M. Vardi. A New Approach to Database Logic. In *Proc. of the 3rd ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 86–96, 1984.

[LFL88] M. Lee, J. Freytag, and G. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. Technical Report RJ 6125 (60619) 3/15/88, IBM Research Division, Yorktown Heights, 1988.

[Lie86] H. Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 214–223, 1986.

[LK86] P. Lyngbaek and W. Kent. A Data Modeling Methodology for the Design and Implementation of Information Systems. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 6–17, 1986.

[Loh88] G. Lohman. Grammar-like Functional Rules for Representing Query Ooptimization Alternatives. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 18–27, 1988.

[LRV88] C. Lécluse, P. Richard, and F. Velez. O_2, An Object-Oriented Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 425–433, June 1988.

[LTP86] W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An Exemplar Based Smalltalk. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 322–330, 1986.

[Lyn88] P. Lyngbaek. Iris/OSQL 3.0 Reference Manual. Technical Report HPL-DTD-88-3, Hewlett-Packard Company, October 1988.

[MBW80] J. Mylopoulos, P. Bernstein, and H. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*, 5(2):185–207, June 1980.

[MG89] J. Marques and P. Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 113–122, October 1989.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 171–182, September 1986.

[MS88] D. Maier and J. Stein. Development and Implementation of an Object-Oriented dbms. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392. M.I.T. Press, 1988.

[MSOP86] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 472–482, July 1986.

[Nic82] J. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, December 1982.

[OL88] K. Ono and G. Lohman. Extensible Enumeration of Feasible Joins for Relational Query Optimization. Technical Report RJ 6625 (63936), IBM Almaden Research Center, December 1988.

[Ont89] Ontologic, Inc. *ONTOS SQL Command Reference*, November 1989. For the Ontologic ONTOS Object Database.

[OOD90] Reference Model for Object Data Management (draft), May 1990. Working draft document OODB 89-01R3 of the ASC X3/SPARC Object-Oriented Database Task Group (OODBTG).

[Osb88] S. L. Osborn. Identity, Equality and Query Optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 346–351. Springer Verlag, 1988.

[Osb89] S. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):310–317, September 1989.

[OT86] S. Osborn and T. Treleaven. The Design of a Relational Database System with Abstract Types as Domains. *ACM Transactions on Database Systems*, 11(3):357–373, September 1986.

[OW89] G. Ozsoyoglu and H. Wang. A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages. *IEEE Transactions on Software Engineering*, SE-15(9):1038–1052, September 1989.

[OY87] Z. Ozsoyoglu and L. Yuan. A New Normal Form for Nested Relations. *ACM Transactions on Database Systems*, 12(1):111–136, March 1987.

[PS87] D. J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 111–117, October 1987.

[RC89] J. Richardson and M. Carey. Persistence in the E Language: Issues and Implementation. *Software – Practice & Experience*, 19(12):1115–1150, December 1989.

[RH86] A. Rosentahl and P. Helman. Understanding and Extending Transformation-Based Optimizers. *Quart. bull. of the IEEE TC on Data Engineering*, 9(4):44–51, December 1986.

[RK87] M. Roth and H. Korth. The Design of ¬1NF Relational Databases into Nested Normal Form. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 143–159, 1987.

[Rot86] M. Roth. *Theory of Non-First Normal Form Relational Databases.* PhD thesis, The University of Texas at Austin, 1986.

[RR82] A. Rosenthal and D. Reiner. An Architecture for Query Optimization. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 246–255, 1982.

[RS87] L. A. Rowe and M. R. Stonebraker. The POSTGRES Data Model. In *Proc. 13th International Conference on Very Large Databases*, pages 83–96, 1987.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 23–34, May 1979.

[SB85] M. Stefik and D. Bobrow. Object-Oriented Programming: Themes and Variations. *The AI Magazine*, pages 40–62, 1985.

[SC88] B. R. Schatz and M. A. Caplinger. Searching in a Hyperlibrary. In *Proc. 4th Int'l. Conf. on Data Engineering*, pages 188–197, February 1988.

[Sch85] H. Schek. Toward a Basic Relational NF^2 Algebra Processor. *Proc. Int. Conf. on Foundations of Data Organization*, pages 173–182, May 1985.

[Sci89] E. Sciore. Object Secialization. *ACM Transactions on Information Systems*, 7(7):103–122, April 1989.

[SCW85] C. Schaffert, T. Cooper, and C. Wilport. Trellis Object-Based Environment Language Reference Manual. Technical Report DEC-TR-372, Digital Equipment Corporation, 1985.

[SHH87] M. Stonebraker, E. Hanson, and C. Hong. The Design of the POSTGRES Rules System. In *Proc. 3rd Int'l. Conf. on Data Engineering*, pages 365–374, 1987.

[Shi81] D. W. Shipman. The Functional Data Model and the Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SM77] D. F. Stanat and D. F. McAllister. *Discrete Mathematics in Computer Science*. Prentice-Hall, 1977.

[Sny87] A. Snyder. Inheritance and the Development of Encapsulated Software Components. In *Proc. of the 20th Annual Hawaii Int'l Conference on System Sciences*, volume 2, pages 227–237, 1987.

[SR86] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 340–355, May 1986.

[SS77] J. M. Smith and D. C. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.

[SS86] H. Schek and M. Scholl. The Relational Model with Relation-valued Attributes. *Information Systems*, 11(2):137–147, June 1986.

[SS90] M. Scholl and H. Schek. A Relational Object Model. Unpublished manuscript, 1990.

[Sta84] J. W. Stamos. Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. *ACM Transactions on Computer Systems*, 2(2):155–180, May 1984.

[Ste87] L. A. Stein. Delegation is Inheritance. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 138–146, 1987.

[Sto76] M. Stonebraker, et. al. The Design and Implementation of Ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[Sto86] M. Stonebraker. Object Management in POSTGRES Using Procedures. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 66–72, 1986.

[Sto87] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proc. 13th International Conference on Very Large Databases*, pages 289–300, 1987.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[SZ89] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. *Quart. bull. of the IEEE TC on Data Engineering*, 12(3):29–36, September 1989.

[SZ90] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

[Tal84] S. Talbot. An Investigation into Logical Optimization of Relational Query Languages. *The Computer Journal*, 27:301–309, 1984.

[TYI88] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema Virtualization in Object-Oriented Databases. In *Proc. 4th Int'l. Conf. on Data Engineering*, pages 23–20, February 1988.

[Ull82] J. D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: An Overview. In *Proc. 15th International Conference on Very Large Databases*, pages 357–366, 1989.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proc. 12th International Conference on Very Large Databases*, pages 101–110, August 1986.

[Weg87] P. Wegner. Dimensions of Object-Based Language Design. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 168–182, 1987.

[Wol87] M. Wolczko. Semantics of Smalltalk-80. In J. Bèzivvin, J. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 108–120. Springer Verlag, 1987.

[WSSH88] P. Wilms, P. Schwarz, H. Schek, and L. Haas. Incorporating Data Types in an Extensible Database Architecture. In *Proc. 3rd Int'l. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 180–192, 1988.

[Zdo86] S. Zdonik. Why Properties are Objects or Some Refinements of "is-a". In *ACM/IEEE Fall Joint Computer Conference*, pages 41–47, November 1986.

[Zdo88] S. B. Zdonik. Data Abstraction and Query Optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 368–373. Springer Verlag, 1988.