Object-Oriented Query Language Design and Processing

Daniel Kim Chung CHAN

Computing Science Department
University of Glasgow

A Thesis submitted for the degree of Doctor of Philosophy
to
The University of Glasgow
in the month of September of the year
One Thousand Nine Hundred and Ninety Four

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

This thesis proposes an object-oriented query language that is more powerful than many existing query languages. The language is formally specified and its expressive power is demonstrated by giving four translation schemes from other prominent object-oriented query languages. Further, this query language can be supported by a query algebra and both the query language and query algebra can be optimised using meaning preserving transformation rules.

Object-Oriented Query Languages. The functional requirements of high-level object-oriented query languages are identified and they combine as well as supplement features found in existing object-oriented query languages. Effectively they formulate a query model against which existing query languages can be evaluated and compared. An evaluation of four representative query languages chosen from research prototypes and commercial products shows that none satisfies all the requirements. On the basis of the requirements a new query language, *object comprehensions*, is developed to provide a concise, clear, powerful, and optimisable query language for object-oriented databases. Some optimisation opportunities for the novel features are identified. A set of translation schemes from the query languages studied to object comprehensions is presented. Such translations demonstrate that object comprehensions are at least as powerful as these query languages and a system supporting object comprehensions can potentially support multiple query languages by providing translations to object comprehensions.

Algebraic Support. The *canonical algebra* provides an abstract execution engine with which object comprehension queries can be expressed using algebraic operations. The translation scheme from object comprehensions to the canonical algebra is very simple and is no[illegible] for supporting queries involving mixed collection classes. The canonical algebra shares many operations with other query algebras and is formally specified. A set of transformation rules that can be used for optimisation is presented whose validity can be verified given the formal specification.

Formal Data Model. The data model which forms the basis of investigation is formally defined using the specification language Z. This *reference data model* captures all the essential features of existing object-oriented data models including multiple inheritance. However, unlike existing data models, it also supports a generalised form of method overloading. Static type checking of such overloaded methods is studied in this thesis.

Authorship

The material presented in the thesis is the product of the author's own independent research carried out at the Computing Science Department of Glasgow University under the supervision of Phil Trinder, David Harper, and Ray Welland.

A fair amount of the materials presented in Chapter 2, 3, 4, and 8 of this thesis has been published before in various technical reports, workshop proceedings, conference proceedings, and the *Computer Journal* [TCH90, CHT92a, CHT92b, CT93, CHT93a, CHT93b, CK94, CT94a, CT94b, CTW95]. All the papers involve more than one author and the co-authors include Phil Trinder, David Harper, Ray Welland, and David Kerr. However the author is responsible for the majority of the technical substance of these papers and of this thesis.

Acknowledgements

It is a great pleasure to acknowledge my debt to the many people involved, directly or indirectly, in my research which led to the production of this thesis.

I am indebted to my supervisor David Harper for arousing my interest in object-oriented database systems and for patiently encouraging me to pursue research on their query languages. I am grateful to my second supervisor Phil Trinder for his willingness to serve as the Polaris in my nomadic approach to conducting research, for going through half-baked ideas and giving useful comments, and for providing guidance and support throughout this work. I am deeply grateful to Ray Welland for being my teacher. As my third supervisor, he painstakingly supervised my write-up. The shape of this thesis owes a lot to the inspiring discussions I had with him. I would also like to thank my professorial supervisor Malcolm Atkinson, since I benefited a great deal from his comments at the early stage of my research.

Thanks also to the fellow members of the database and formal method research groups and particularly members of the COMANDOS, FIDE, and Bulk Types projects for providing a stimulating working environment. Special thanks go to David Kerr, Roberto Barros, and Alastair Reid.

It is also a real pleasure to express my appreciation of the friendliness, cooperation, and professionalism shown by members in the Glasgow University Computing Science Department throughout these memorable years. To all of them, who cannot all be named, my heartfelt appreciation and thanks.

To my family and friends, thank you for your help, for your support, for sharing my anxiety, for impatiently urging me to finish, and for never having doubted me - even when I did myself. To them I dedicate this thesis.

Finally I am grateful to Glasgow University, the Committee of Chancellors and Vice-Principals of the Universities of the United Kingdom, and the European Union for the scholarships as well as Professor Keith van Rijsbergen and Professor Arthur Allison for backing the applications that made this degree possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Novel applications, such as computer-aided design, require data models that support complex relationships and rich constructs. Object-oriented data models supporting complex objects and operations have been developed to cope with the requirements of these novel applications. To manipulate the rich structures found in these data models a query language will require more constructs. Often this is resolved by merely extending a relational query language with ad hoc constructs. More understanding of the requirements of object-oriented query languages is required before a general and consistent notation can be designed.

Equally important is the provision of algebraic support of such a language and its optimisation. Until now, the study of object-oriented query languages has been hampered by the lack of a formal data model. Attempts have been made to resolve this but the resultant data models are both simplistic and restrictive. What is required is a formal and realistic data model which includes the essential features of existing data models.

The aim of the research reported in this thesis is to address the above-mentioned shortcomings through the development of a new query language and a query algebra as well as the use of formal specification technique. This thesis proposes a set of functional requirements for object-oriented query languages. A new object-oriented query language which satisfies the requirements identified is presented. Further, this query language can be supported by a query algebra and both the query language and query algebra can be logically optimised. A reference data model which is formally defined using the specification language Z forms the basis of the investigation.

1.1 Scope of Thesis

Experience with relational database systems has demonstrated that efficient and effective query processing is a determining factor for system performance [JK84]. Relational query languages allow the user to access the data in a declarative manner. That is, the user only specifies what should be retrieved from the database. It is the responsibility of the

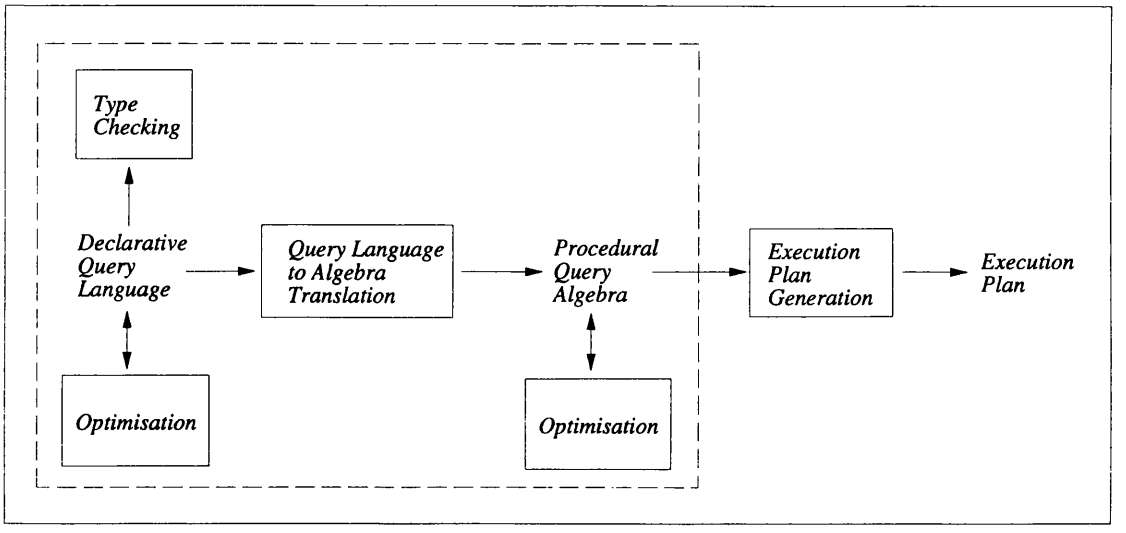

Figure 1.1: Query Language Processing Framework.

database system to determine how such a request should be evaluated against the database. Therefore the database system has to generate a procedural query evaluation program, also called an execution plan, from the declarative user query without changing the user's intention. This thesis examines how object-oriented query languages can be processed in a framework similar to that of relational systems (Figure 1.1). Here, a query language means a retrieval-based language not dealing with the manipulation of instances such as updates.

The parts of the query processing framework this thesis investigates include: (1) design of declarative user query language; (2) optimisation of user query language; (3) procedural query algebra to support declarative query language; (4) optimisation of query algebra; and (5) type checking of overloaded methods. They are contained in the rectangle with a dotted boundary in Figure 1.1. These parts will be further explained in a later section.

This thesis differs from previous works primarily in the scope that is covered: (1) a richer reference data model supporting multiple collection classes is introduced; (2) the problem of statically type checking overloaded methods in the presence of multiple inheritance is studied; (3) the interaction of different collection classes and other functional requirements of object-oriented query languages are examined; and (4) a comparison of the expressive power of object comprehensions with four well-known object-oriented query languages is made.

A number of issues related to query processing are not studied in this thesis. The study of optimisation generates a set of logical and semantic transformation rules. Search strategies used for transformation are not covered but preliminary results for non-recursive queries can be found in [Mit93]. Execution plan generation is studied in [SO90] but the choice between generation plans is only very briefly discussed. The research reported in this thesis is carried out in the context of a formal data model and has not been integrated

into a running system. However given a suitable implementation platform, it is believed that the query language and query algebra can be implemented since a similar language and its optimisation was prototyped before [TCH90].

1.2 Contributions

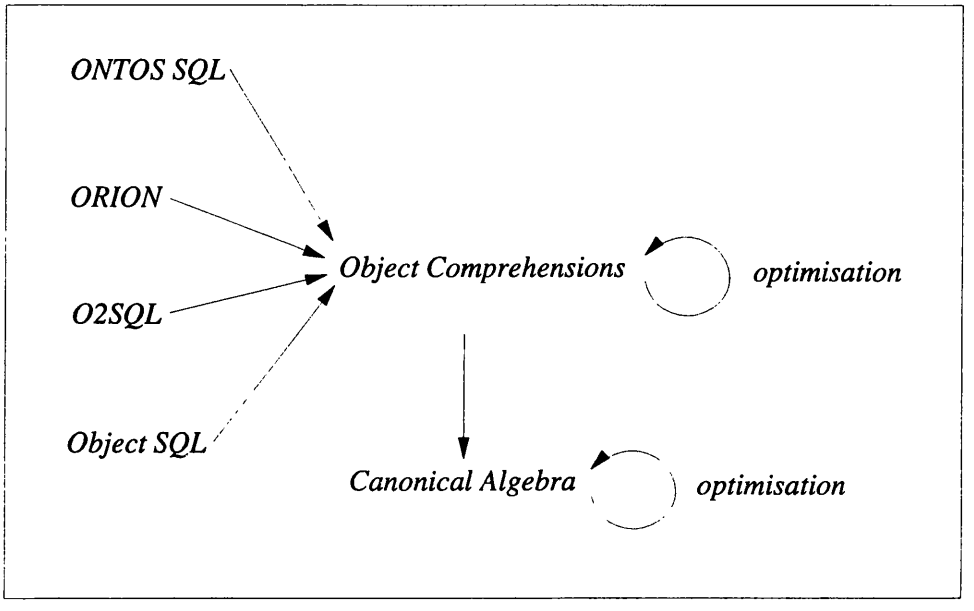

Figure 1.2: Query Language Processing.

The specific contributions documented in this thesis are listed below and the inter-relationship between the parts of the work is depicted in Figure 1.2.

- An object-oriented query model manifested as functional requirements that can be used to evaluate, compare, improve, and design query languages, is proposed.
- A new high-level object-oriented query language, *object comprehensions*, is shown to satisfy the object-oriented query model, and new optimisation rules are developed to improve queries involving class testing and quantifiers.
- The expressive power of object comprehensions is demonstrated by showing that any query expressed in four prominent query languages can be expressed in object comprehensions; hence the latter notation is at least as powerful as the four query languages.
- A new object-oriented query algebra, the canonical algebra, which supports three collection classes, set, bag, and list, is proposed together with a set of transformation rules that can be used for optimisation.

- An object-oriented data model called the *reference data model*, which features multiple inheritance, a generalised form of method overloading, dynamic binding and static type checking, is specified using the specification language Z and several useful properties of the data model have been proved.
- Challenges involved in proposing a satisfactory solution for view support using query language are identified.

1.3 Organisation

Chapter 2 covers the reference data model. A formal specification of the reference data model is given in the specification language Z. Some properties of the reference data model are proved using the specification. A running example database is described and defined in terms of the reference data model. A comparison of four prominent object-oriented data models and the reference data model is summarised.

Chapter 3 studies the requirements for object-oriented query languages. A list of functional requirements is presented. A summary evaluation of four prominent object-oriented query languages is reported. The query languages are chosen as representative languages mainly because they are well-reported and the most referenced in the literature. Non-functional requirements are briefly discussed.

Chapter 4 introduces object comprehensions beginning with a history of their development. It is followed by a set of example queries that illustrate object comprehensions and demonstrate that they satisfy the requirements identified in Chapter 3. Query optimisations are presented as meaning-preserving transformation rules together with conditions for their application.

Chapter 5 addresses the expressive power of object comprehensions. A suite of translation schemes from four prominent query languages to object comprehensions is developed. The four translation schemes demonstrate that object comprehensions are at least as powerful as the four query languages with respect to the reference data model and can provide support for these query languages.

Chapter 6 introduces the canonical algebra and its support for object comprehensions. The canonical algebra is introduced, illustrated, and formally defined using Z. A translation between object comprehensions and the canonical algebra is given. The translation scheme demonstrates that the algebra is canonical in the sense that it can express and hence support queries expressed in object comprehensions and hence in the other four query languages.

Chapter 7 summarises the results presented in the thesis and concludes that object comprehensions are a good query language for object-oriented databases and can be supported using the canonical algebra. Limitations of the approach taken in this thesis are discussed. Directions for future work are described.

Chapter 8 investigates one possible avenue of further research in detail. The difficulties of supporting views in object-oriented databases are revealed. Existing proposals are examined and their advantages and disadvantages are discussed. The problems to be overcome by a satisfactory solution are identified.

Chapter 2

The Reference Data Model

Many object-oriented data models have been introduced in various systems and proposals. Despite their apparent diversity, these models share many common features. These features have been acknowledged as the essence of any object-oriented data model [ABD+89, Ban89, Dit91]. The reference data model presented in this chapter includes the significant features found in most object-oriented data models, for example, IRIS [LK86], GemStone [Ser87], ORION [Kim90], ONTOS [Ont91a], and O_2 [BDK92], to name but a few. A number of object-oriented data models have been given a formal specification [MH87, Wol87, SO90, DD91, BDK92, Nor92]; however, the various features of object-orientation are studied separately. The reference data model provides a uniform framework for the study of their interaction. In particular, it supports a generalised form of method overloading and addresses the problem of statically type checking such overloaded methods.

The organisation of the chapter is as follows. Section 2.1 provides an informal description of the reference data model. Section 2.2 discusses the challenges of statically type checking overloaded methods in an environment supporting multiple inheritance. Section 2.3 to 2.11 specify the reference data model formally using the specification language Z. Section 2.12 touches on reasoning about the specification. Section 2.13 presents an example database that will be used throughout the thesis. Section 2.14 concludes with a discussion of related works including a comparison of the reference data model with the data models of IRIS, ORION, ONTOS, and O_2.

2.1 Informal Description

The reference data model supports both objects and base values. Objects are identified by object identifiers. Base values, like integers and strings, do not carry an object identifier. Object identifiers and base values are collectively called values. Every base value belongs to a system-defined base type which defines the operations that can be performed on

it. Similarly, an object belongs to a class [1], which can be either system-defined or user-defined. A class uses attributes and methods to model a concept or a phenomenon in the application world. Attributes can be accessed only via methods defined in the class. A method is called when an object receives a message. This dispatching mechanism is generally referred to as message passing.

Classes can be related to one another using the ISA relationship. If a class is related to another class by the ISA relationship, the former class is called a subclass of the latter class while the latter class is referred to as the superclass of the former one. A class can have more than one superclass and should inherit and support all methods defined in its superclasses. As a consequence of inheritance, an object can be used wherever an object of its superclass is expected. There is a root class in the model which is a subclass of no class and a direct or indirect superclass of all other classes. In contrast to the root class, a bottom class is defined as being a subclass of all classes. The unique instance of this class is denoted as *nil*. The bottom class is only used for the purpose of query processing and is therefore an internal object not accessible to the user. The structure formed by the ISA relationship among classes is called a class graph, sometimes mis-called a class hierarchy.

In some models [LK86, Kim90], every class is associated with a set containing all objects of that class, usually called a class extent. The reference data model does not support class extents. In brief, the reference data model supports the following features

- base values
- complex objects
- object identity
- encapsulation
- message passing
- method overloading
- multi-methods
- classes
- class hierarchy
- multiple inheritance
- static type checking
- dynamic binding

2.2 Challenges

Inheritance requires a subclass to support all methods defined in its superclasses in addition to its own methods. If every method is given a unique name, statically type checking messages will be straightforward regardless of the kind of inheritance supported. However, such uniqueness hinders extensibility - one primary strength of the object-oriented paradigm. Hence, many data models, including the reference data model, support method overloading where the same method name can be given to different methods defined in the same class or in different classes. This complicates static type checking as a method name can now represent many different methods. Many data models, but not the reference data model, resolve this problem by using only the type of the receiving object to determine the method corresponding to a message at both compile-time and run-time. However this simple selection scheme has limitations (one limitation is illustrated in Subsection 2.7.1).

[1] In this thesis, *types* include both *base types* and *classes* as defined in the specification. Types and classes are therefore synonyms in most cases.

To enable static type checking some restrictions must be imposed on overloaded methods that can be applied to the same set of arguments (hereafter called *confusable methods*). A well known proposal [Car84] suggested the contravariance rule requiring an overloaded method defined in a subclass to take on more general arguments and returning more specific result. The use of more general arguments in an overloaded method could be argued as "unnatural" [DT88, Mey88, MHH91, Cha92]. In a multiple inheritance environment, a more general rule allowing the ordering of overloaded methods from different superclasses is required.

The selection scheme of the reference data model uses the type of the receiving object together with the types of the arguments for the matching at compile-time and run-time. Following [BDG+88], such overloaded methods are called *multi-methods.* Support of multi-methods allows more flexibility in defining overloaded methods. The reference data model also explores the use of covariance rule where the types of the arguments of an overloaded method defined in a subclass do not need to be more general than that in the superclass. Static type checking will make use of the least general method while dynamic binding will pick the most specific method [2]. This enriches the inheritance mechanism by avoiding the blocking of more general methods by more specific methods.

In an environment supporting multiple inheritance, the ISA relationship alone cannot decide if one method is more specific or general than another. *Inheritance ordering*, which respects the ISA relationship, is introduced to allow such comparison to be made.

The next sections use the Z specification language [PST91, Spi92] to specify the reference data model. Z has been used to describe data models for example the relational model and its algebras [SH85, BH91, Bar93] and an object-oriented data model [MG93]. Z encourages a modular approach to specification. A system can be specified in terms of a number of small "mini-specifications" which can then be combined easily using the *schema calculus.* The resultant specification is often more manageable, concise, readable, and comprehensible. Equally important, Z specifications are amenable to formal reasoning as it has a formal semantics and a set of sound inference rules [Spi88]. In Chapter 6, a query algebra for the reference data model is defined similarly using Z; hence, provides a uniform framework for the reasoning of query expressions.

2.3 Objects + Base Values = Values

The data domain of discourse consists of atomic data drawn from a given set:

$$[VALUE]$$

Elements in $VALUE$ are referred to as *values* and can be partitioned into two disjoint

[2] The meanings of the two bounds differ from that used in [OBBT89] and are defined in the specification.

subsets *Object* and *BaseValue*:

$$\begin{array}{|l} Object : \mathbb{F}\ VALUE \\ BaseValue : \mathbb{P}\ VALUE \\ \hline \langle Object, BaseValue \rangle \text{ partitions } VALUE \end{array}$$

Object is a set of object identifiers with which objects are represented in a database. For this reason the elements in *Object* are referred to as *objects*. *BaseValue* contains data, like integers and strings, that are not represented by object identifiers. These elements are referred to as *base values*. Values in *Object* can be compared for equality while base values are subject to other operations, e.g. the "less than" operator. *Object* can be further divided into two disjoint subsets. The first subset, *OCollection*, contains objects representing groups of homogeneous elements that can be either base values or objects. The second subset, *MonoObject*, holds all non-collection objects.

$$\begin{array}{|l} OCollection : \mathbb{F}\ Object \\ MonoObject : \mathbb{F}\ Object \\ \hline \langle OCollection, MonoObject \rangle \text{ partitions } Object \end{array}$$

The reference data model supports three kinds of collection. They are represented using three subsets of *OCollection*.

$$\begin{array}{|l} OSet : \mathbb{F}\ OCollection \\ OBag : \mathbb{F}\ OCollection \\ OList : \mathbb{F}\ OCollection \\ \hline \langle OSet, OBag, OList \rangle \text{ partitions } OCollection \end{array}$$

2.4 Values and Type Names

In this section, and indeed the next six sections, the theme of the discussion will be on database schemata. In other words, the subject matter is what a generic database schema looks like and what the associated constraints are.

Types are identified by names which are drawn from a given set:

$$[TYPE\_NAME]$$

The two kinds of values introduced in the previous section, base values and objects, belong to different types. Base values belong to *base types* while objects belong to *classes*. It is therefore useful to partition *TYPE_NAME* correspondingly into two disjoint subsets:

$$
\begin{array}{|l}
BaseTypeName : \mathbb{F}\ TYPE\_NAME \\
ClassName : \mathbb{F}\ TYPE\_NAME \\
\hline
\langle BaseTypeName, ClassName \rangle \text{ partitions } TYPE\_NAME
\end{array}
$$

Now the relationship between values and types can be defined using two total functions as the following:

$$
\begin{array}{|l}
typeOf\ \_ : VALUE \rightarrow TYPE\_NAME \\
\hline
\forall\, v : VALUE \bullet \\
\quad v \in BaseValue \Leftrightarrow (typeOf\ v) \in BaseTypeName \\
\quad \wedge \\
\quad v \in Object \Leftrightarrow (typeOf\ v) \in ClassName
\end{array}
$$

Applying *typeOf* to an object returns its defining class name. Applying it to a base value returns its base type name. Since base types are well understood they will not be discussed in the thesis.

$$
\begin{array}{|l}
kindOf\ \_ : OCollection \rightarrow ClassName \\
\hline
\forall\, c : OCollection \bullet \\
\quad c \in OSet \Leftrightarrow (kindOf\ c) = ASet \\
\quad \wedge \\
\quad c \in OBag \Leftrightarrow (kindOf\ c) = ABag \\
\quad \wedge \\
\quad c \in OList \Leftrightarrow (kindOf\ c) = AList
\end{array}
$$

Applying *kindOf* on a collection object reveals its collection kind: set, bag, or list. *ASet*, *ABag*, and *AList* are type names representing the collection kinds.

2.5 Rooted Class Graph

Before discussing classes which govern the behaviour of values in *Object*, the ISA relationship between classes is examined. Note that the relationship is defined over *ClassName* and not the classes themselves.

2.5.1 The ISA Relationship

Classes in a database are related to one another via the ISA relationship ($\prec$). The structure formed by the ISA relationship among classes is called a class graph. The class graph is represented by a relation over class names.

ISA

$$\_ \prec \_ : ClassName \leftrightarrow ClassName$$

$$ClassName = (\mathrm{dom} \prec) \cup (\mathrm{ran} \prec)$$

$$\forall cn : ClassName \bullet \neg (cn \prec^+ cn)$$

The first constraint implies that all classes related by the ISA relationship have their names in the set *ClassName* and every class in *ClassName* is involved in the relationship. It also has the implication that the ISA relationship is finite. The second constraint asserts that the ISA relationship does not relate directly or indirectly ($\prec^+$) a class to itself: the class graph is directed acyclic.

2.5.2 The Root Class

The class graph has a root class which is a subclass of no class and a direct or indirect superclass of all other classes.

ROOT

$$ISA$$

$$root : ClassName$$

$$\forall cn : ClassName \bullet cn \neq root \Leftrightarrow cn \prec^+ root$$

ROOT uses the *ISA* specification given earlier. The constraint asserts that every class, except the root class, can be reached from the root class via the ISA relationship.

2.5.3 The Bottom Class

In contrast to the root class, a bottom class is defined as being a subclass of all classes. The unique instance of this class is denoted as *nil*. The bottom class is only used for the purpose of query processing and is therefore an internal object not accessible to the user.

BOTTOM

$$ISA$$

$$bottom : ClassName$$

$$nil : Object$$

$$\forall cn : ClassName \bullet cn \neq bottom \Leftrightarrow bottom \prec^+ cn$$

$$typeOf\ \ nil = bottom$$

2.5.4 Type Conformance

A class is said to conform to another class if they are related directly or indirectly by the ISA relationship or they are indeed the same class ($\prec^*$). Since relationships between base

types is not studied in the thesis, the conformance relation over base types degenerates to the equality relation over base types.

CLASS_GRAPH

$$
\begin{array}{l}
ROOT \\
BOTTOM \\
\_ \prec\!\!\!\prec \_ : TYPE\_NAME \leftrightarrow TYPE\_NAME \\
\hline
\forall t_1, t_2 : TYPE\_NAME \bullet \\
\quad t_1 \prec\!\!\!\prec t_2 \Leftrightarrow \\
\qquad \{ t_1, t_2 \} \subseteq ClassName \land t_1 \prec^* t_2 \\
\qquad \lor \\
\qquad \{ t_1, t_2 \} \subseteq BaseTypeName \land t_1 = t_2
\end{array}
$$

2.6 Classes

This section examines the definition of a class which is essentially a template for the state of an object and contains operations that can be applied to the state of an object. The ISA relationship induces further constraints on class definition. The constraints are discussed in the last subsection.

2.6.1 Methods

A method is characterised by its name, signature, and semantics. Method names are drawn from a given set:

$[METHOD\_NAME]$

The signature of a method captures the types of the formal arguments and of the result. Each method receives at least one argument object - the object on which the method is called. Every method must return a value as its result. Therefore the signature of a method has at least two type names, and usually more.

Note that seq_1 represents sequences with at least one element, $\mathbb{N}_1$ represents the set of natural numbers without zero, $\langle\ \rangle$ represents a sequence literal, $\frown$ is a Z operation performing sequence concatenation, and $\#$ is another Z operation returning the number of elements in a sequence.

SIGNATURE

$$
\begin{array}{l}
argumentTypes : \mathrm{seq}_1\ TYPE\_NAME \\
resultType : TYPE\_NAME \\
types : \mathrm{seq}_1\ TYPE\_NAME \\
length : \mathbb{N}_1 \\
\hline
types = argumentTypes \frown \langle resultType \rangle \\
length = \#argumentTypes
\end{array}
$$

The semantics of a method captures the meaning of the method (i.e. what the method does). In practice, it captures the implementation of the method. Here an extensional definition is given to the semantics of a method which is represented as a partial function.

METHOD

$$
\begin{array}{l}
CLASS\_GRAPH \\
name : METHOD\_NAME \\
signature : SIGNATURE \\
selfType : ClassName \\
semantics : \mathrm{seq}_1\ VALUE \nrightarrow VALUE \\
\hline
selfType = head\ \ signature.argumentTypes \\
\forall\, vs : \mathrm{seq}\ VALUE \mid \\
\qquad \#vs = signature.length \land \\
\qquad (\forall\, i : 1 \mathinner{..} \#vs \bullet typeOf\ (vs\ i) \lessdot signature.argumentTypes\ i) \bullet \\
\quad typeOf\ (semantics\ vs) \lessdot signature.resultType
\end{array}
$$

The first constraint establishes that the first type name in a signature is the same as the name of the class in which the method is defined. The second constraint ensures that *semantics* is correctly typed. In other words, if the types of the actual arguments conform to the formal argument types the method will return a result conforming to the result type.

2.6.2 Attributes

Every object has its own set of attributes to capture the state of the object. Attributes are not directly accessible. Methods are the only means to manipulate them.

Assume a set for the names of attributes:

$$[ATTRIBUTE\_NAME]$$

The types of the attributes of a class can be defined as a finite partial function. The implications are: (1) there is a finite number of attributes; (2) attribute names are unique within a class; and (3) the value of an attribute can be an object or a base value. When an object is created, this function will be used to create a set of named attribute values

having the same names and value types as specified in this function. Class instantiation is not further discussed in the thesis.

$$
\begin{array}{|l}
\hline
\blacksquare ATTRIBUTE \\
attributes\_ : ATTRIBUTE\_NAME \twoheadrightarrow TYPE\_NAME \\
\hline
\end{array}
$$

The symbol ■ is used as a naming convention to identify the part of a schema which is to be hidden from direct access. In this case, direct access to the attributes of an object is to be prohibited.

2.6.3 Classes

$$
\begin{array}{|l}
\hline
CLASS \\
\blacksquare ATTRIBUTE \\
name : ClassName \\
methods : \mathbb{F}\, METHOD \\
directSuperclasses : \text{seq } ClassName \\
applicableMethods : \mathbb{F}\, METHOD \\
\hline
\forall m_1, m_2 : methods \mid m_1 \neq m_2 \bullet \\
\quad m_1.name = m_2.name \Rightarrow \\
\qquad m_1.signature.argumentTypes \neq m_2.signature.argumentTypes \\
\forall m : methods \bullet m.selfType = name \\
\hline
\end{array}
$$

A class consists of a class name, attributes, methods, a sequence of direct superclass names, and a set of applicable methods. Superclasses are ordered as given in the class definition and this ordering will be used to define the *local inheritance ordering.* Applicable methods include methods defined by the class itself as well as methods inherited from its superclasses. The first constraint asserts that methods having the same name must have different argument types. This allows method overloading. The second constraint asserts that the methods must be correctly typed to the class in which they are defined.

2.6.4 Class Names, Classes, and Inheritance

The relationship between the class names in a class graph and the classes themselves is defined next. *Class* is a set containing all classes in the database. The function, *hasName*, relates a class name to its class.

SCHEMA

$$
\begin{array}{l}
CLASS\_GRAPH \\
Class : \mathbb{F}\, CLASS \\
hasName\_ : ClassName \rightarrowtail\!\!\!\!\!\!\rightarrow Class \\
\hline
\forall\, cn : ClassName \bullet (\, hasName\, cn\,).name = cn \\
\forall\, c, c_{super} : Class \bullet \\
\quad c_{super}.name \in \mathrm{ran}(\, c.directSuperclasses\,) \Leftrightarrow \\
\quad\quad c.name \prec c_{super}.name \\
\forall\, c : Class \bullet \\
\quad c.applicableMethods = c.methods \cup \\
\quad (\bigcup \{\, c_{super} : Class \mid c_{super}.name \in \mathrm{ran}(\, c.directSuperclasses\,) \bullet \\
\quad\quad c_{super}.applicableMethods \,\})
\end{array}
$$

The first constraint specifies that *hasName* takes a class name and returns a class with the same name. Being a total bijective function implies that: (1) there are as many classes as there are class names; (2) every class name is associated with a unique class; and (3) all the classes have unique names. The second constraint insists that superclasses named in a class definition are actually related to the class in the class graph. The third constraint specifies the set of applicable methods for a class. $\bigcup$ represents the distributed union operation which takes a set of sets and returns a set containing all members in the original set elements. In this case, $\bigcup$ combines all methods inherited from the superclasses. $\cup$ then combines this resultant set of methods with the set of methods defined in the class to give a set of all applicable methods.

2.7 Class Ordering

Inheritance permits a class to have more than one superclass and requires it to support all the methods of its superclasses. Since confusable methods may exist some way of choosing a method for type checking and dynamic binding must be provided. This section examines the problems of type checking messages when method overloading is allowed and introduces the *local inheritance ordering*, which is the key to the solution.

2.7.1 Single Inheritance Environment

Figure 2.1 presents parts of a database schema consisting of four classes: A, B, C, and D, where $B \prec A$ (B is a subclass of A) and $D \prec C$. A has a method m, which takes two arguments: an object of class A and an object of class C, and returns an object of class C. B, being a subclass of A, inherits this method from A. However, B also defines a method with the same name. Therefore B has two methods of the same name at its disposal.

Given this database schema, the effect on type checking can be studied using the application program in Figure 2.1. Variables are declared in line (1) at the beginning of

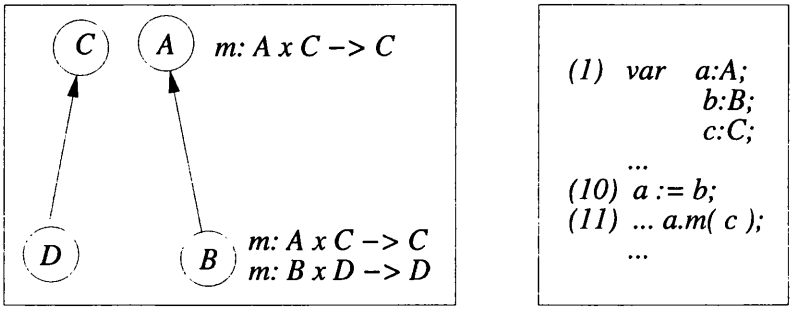

Figure 2.1: Unsafe Static Type Checking.

the program fragment. The assignment statement in line (10) assigns the variable b to a variable of its superclass. This is a valid statement and will pass type checking. In line (11) object a is passed the message m with variable c as an argument whose static type is C. Object a has static type A and therefore the message m corresponds to the only method A has. The message passes type checking; whereas what actually happens at run-time is something quite different. At run-time, variable a actually contains an object of class B. Using the simple selection scheme described in Section 2.2, the method m defined in B (i.e. $m : B \times D \rightarrow D$) will be selected. If c contains an object of class D, it will be accepted as a valid argument to m. On the other hand, if c contains an object of class C, the same as its static type, it will be considered an invalid argument to the method m selected at run-time. To conclude, the program passes type checking but it may fail at run-time!

2.7.2 Multiple Inheritance Environment

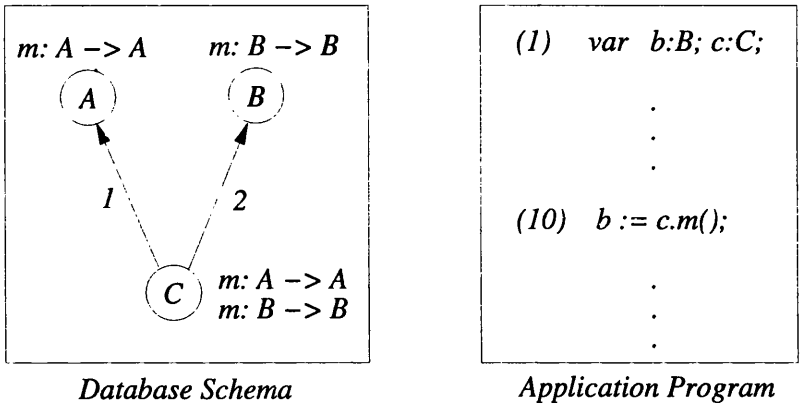

Figure 2.2: Confusable Methods from Incomparable Classes.

Multiple inheritance introduces further problems for type checking and dynamic binding. The database schema in Figure 2.2 demonstrate the problems. In the schema, C inherits two methods named m, one from A and one from B. In the program fragment, variable c, whose static type is C, is passed the message m. The question is which method m should be used. The answer does matter because it determines the result type and consequently affects type checking. In the single inheritance environment, the method defined for the subclass is chosen but here there is no ISA relationship between A and B where the overloaded methods are defined. In other words, classes are partially ordered and A and B are incomparable.

To enable a systematic selection scheme, a better ordering over methods is required. The only sensible way of ordering methods is to use their signatures, which contains primarily type names. An ordering over types, that should subsume the ISA relationship defined in the database schema, is therefore required and will form the base for ordering of methods. The next subsection introduces such an ordering.

2.7.3 Local Inheritance Ordering

Various possible orderings and their effects on type checking overloaded methods are examined in [ADL91]. The reference data model adopts the *local inheritance ordering.* It is local because each ordering only relates a given class to its superclasses; but not classes which are not related to the given class by the ISA relationship. Note that two classes which appear in two different local inheritance orderings may be ordered differently. When a class named cn_1 is more specific than a class named cn_2 with respect to another class named cn, it can be expressed using the local inheritance ordering as the following,

$$cn_1 \ll_{cn} cn_2$$

The relation $\ll_{cn}$ is obtained by applying the total function $\ll$ on the class name cn.

LOCAL_ORDERING

SCHEMA

$$\_ \ll \_\ \_ : \mathit{ClassName} \rightarrow (\ \mathit{ClassName} \leftrightarrow \mathit{ClassName}\ )$$

$$\forall\, c : \mathit{Class};\ cn, cn_1 : \mathit{ClassName} \mid$$
$$cn = c.\mathit{name} \land$$
$$cn_1 = c.\mathit{directSuperclasses}\ 1 \bullet$$
$$cn \ll_{cn} cn_1$$

$$\forall\, c : \mathit{Class};\ cn, cn_j, cn_k : \mathit{ClassName};\ j, k : \mathbb{N}_1 \mid$$
$$cn = c.\mathit{name} \land$$
$$\{ j, k \} \subseteq 1\,..\,\#(\ c.\mathit{directSuperclasses}\ ) \land$$
$$j + 1 = k \land$$
$$cn_j = c.\mathit{directSuperclasses}\ j \land$$
$$cn_k = c.\mathit{directSuperclasses}\ k \bullet$$
$$cn_j \ll_{cn}{}^{+}\ cn_k$$

$$\forall\, c : \mathit{Class};\ cn, cn_j, cn_{j1}, cn_{j2}, cn_k : \mathit{ClassName};\ j, k : \mathbb{N}_1 \mid$$
$$cn = c.\mathit{name} \land$$
$$\{ j, k \} \subseteq 1\,..\,\#(\ c.\mathit{directSuperclasses}\ ) \land$$
$$j + 1 = k \land$$
$$cn_j = c.\mathit{directSuperclasses}\ j \land$$
$$cn_k = c.\mathit{directSuperclasses}\ k \land$$
$$\{\ cn_{j1}, cn_{j2}\ \} \subseteq$$
$$(\ \mathrm{ran}(\ \{c_j.\mathit{name}\} \lhd \prec^{+}\ )\ \setminus$$
$$\bigcup\ \{\ c_i : \mathit{Class} \mid$$
$$(\exists\, i : (j+1)\,..\,\#(\ c.\mathit{directSuperclasses}\ ) \bullet$$
$$c.\mathit{directSuperclasses}\ i = c_i.\mathit{name}\ ) \bullet$$
$$\mathrm{ran}(\ \{c_i.\mathit{name}\} \lhd \prec^{+}\ )\ \}\ ) \land$$
$$cn_{j1} \ll_{c_j} cn_{j2} \bullet$$
$$cn_{j1} \ll_{cn} cn_{j2} \land cn_{j2} \ll_{cn}{}^{+} cn_k$$

The first constraint asserts that a class is more specific than its first superclass. The second constraint says that superclasses are ordered according to the order given in the class definition. This is the basis to allow the comparison of classes that are not related by the ISA relationship. Note that the transitive closure ($\ll_{cn}{}^{+}$) implies that there can be other classes in between. Given a class and one of its superclasses, there can be more than one way the two classes are connected by the ISA relationship, for instance, via different direct superclasses. The last constraint guarantees that only one such connection will be used in the ordering. The last connection as prescribed by the order of the direct superclasses will be used. In other words, a method from an indirect superclass is considered more general and is ordered after the last direct superclass that is connected to it. Apart from this deviation, the local ordering with respect to a class agrees with the local inheritance orderings of its superclasses.

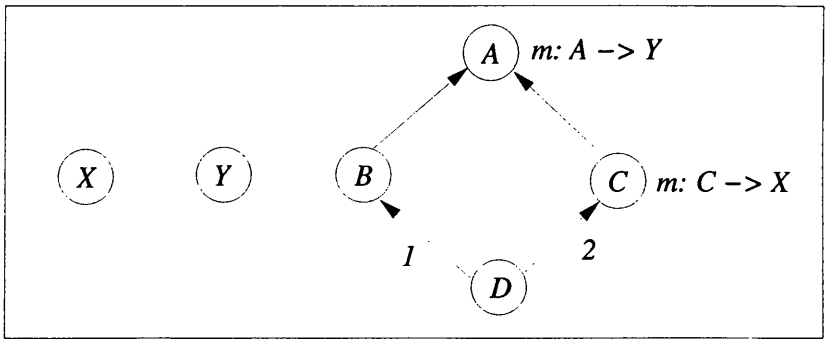

Figure 2.3: Methods Inherited via Multiple Paths.

Given the schema in Figure 2.3 with four classes such that $B \prec A$, $C \prec A$, $D \prec B$, $D \prec C$, and $B \ll_D C$ (it is denoted by labelling the D to B arc with 1 and the D to C arc with 2), the local inheritance ordering will order them in descending order of specificity as D, B, C, A. The choice of such an ordering is a compromise between being consistent with the substitution semantics among ISA-related classes and "localising" the restrictions caused by cycles among classes in the generalised inheritance ordering. The latter issue is discussed in the second half of the next section.

There is however one problem with this schema. If message m is passed to a variable of class A, m of A will be used at compile-time for type checking and class Y will be its result type. Assuming that an object of class C is actually assigned to the variable, naturally m of C will be used at run-time which produces a result of class X. This is similar to the scenario described earlier in subsection 2.7.1. The two classes X and Y are unrelated classes and hence may cause a run-time error. New measures are required to remove this loophole and they are presented next.

2.8 Method Consistency

The problem found in the previous section exposes a more general fault in the development so far: argument types are used to order methods but the result types have not been taken into account. What is lacking is a concept of consistency between the ordering of argument types and the ordering of result types.

2.8.1 Method Confusability

Consistency is essential only for confusable methods. It is not necessary to consider consistency when methods having the same name can never be applied to the same set of arguments; they are not confusable and can never be used in the same context.

CONFUSABILITY

SCHEMA

$\_ isConfusableWith\_ : METHOD \leftrightarrow METHOD$

$$\forall m_1, m_2 : METHOD;\ s_1, s_2 : SIGNATURE \mid$$
$$s_1 = m_1.signature \ \wedge \ s_2 = m_2.signature \bullet$$
$$m_1\ isConfusableWith\ m_2 \Leftrightarrow$$
$$m_1.name = m_2.name \wedge$$
$$s_1.length = s_2.length \wedge$$
$$\forall i : 1 \mathinner{..} s_1.length \bullet$$
$$\exists t : TYPE\_NAME \bullet$$
$$t \prec\!\!\!\prec (s_1.argumentTypes\ i) \wedge$$
$$t \prec\!\!\!\prec (s_2.argumentTypes\ i)$$

The definition of confusability is given above in the form of a relation. Two methods are confusable when: (1) they have the same name; (2) they take the same number of arguments; and (3) for every argument position there is a type which conforms to both argument types at that position.

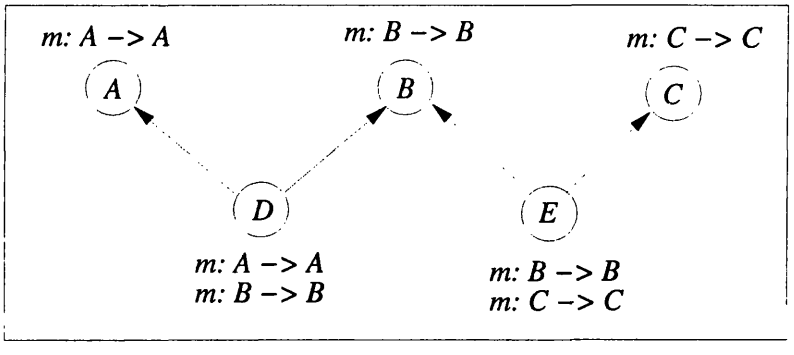

Figure 2.4: Confusability is Not Transitive.

Confusability is not a transitive relation, for example, in Figure 2.4, $m : A \rightarrow A$ and $m : B \rightarrow B$ are confusable, $m : B \rightarrow B$ and $m : C \rightarrow C$ are confusable, but not $m : A \rightarrow A$ and $m : C \rightarrow C$.

2.8.2 Partitioning Confusable Methods

Consistency is important only for confusable methods, it is therefore sensible to partition all methods into sets of confusable methods. Firstly methods can be partitioned by their names. Since overloading allows methods to have the same name but different numbers of arguments. Therefore a set of methods having the same name can be further partitioned according to the number of arguments each method takes. The resultant sets can then be divided based on confusability.

$$
\begin{array}{l}
\textit{PARTITIONS} \\
\hline
\textit{CONFUSABILITY} \\
\textit{allMethods} : \mathbb{F}\,\textit{METHOD} \\
\textit{methodPartitions} : \textit{METHOD\_NAME} \times \mathbb{N}_1 \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow \mathbb{F}\,\mathbb{F}\,\textit{METHOD} \\
\textit{allConfusableSets} : \mathbb{F}\,\mathbb{F}\,\textit{METHOD} \\
\hline
\textit{allMethods} = \bigcup \{\, c : \textit{Class} \bullet c.\textit{methods} \,\} \\
\forall\, m : \textit{allMethods};\ mn : \textit{METHOD\_NAME};\ n : \mathbb{N}_1 \mid \\
\quad mn = m.\textit{name} \ \land\ n = m.\textit{signature}.\textit{length} \Leftrightarrow \\
\qquad \exists_1\, ms : \mathbb{F}\,\textit{METHOD} \bullet m \in ms \ \land\ ms \in \textit{methodPartitions}(mn, n) \\
\textit{allConfusableSets} = \bigcup\ (\operatorname{ran} \textit{methodPartitions}\,) \\
\textit{allMethods} = \bigcup \textit{allConfusableSets} \\
\forall\, ms : \textit{allConfusableSets} \mid \#ms > 1 \bullet \\
\quad \forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet \\
\qquad m_1\ \textit{isConfusableWith}^{+}\ m_2 \\
\hline
\end{array}
$$

Note that *methodPartitions* is a partial injective function. The second constraint asserts that a method is contained in only one confusable set. The third constraint gathers all confusable method sets. The last constraint asserts that methods in a confusable set are linked directly or indirectly by the confusability relation ($\textit{isConfusableWith}^{+}$) - recall that the relation itself is not transitive.

2.8.3 Generalised Inheritance Ordering

The local inheritance ordering defined in the previous Subsection can be used to order methods only when the static types of the actual arguments are available. This information is, however, not available during schema definition time. Therefore ordering of methods and hence the examination of consistency cannot be done during schema definition time using the local inheritance ordering. A generalised form of this ordering is necessary and is specified below:

$$
\begin{array}{l}
\textit{INHERITANCE\_ORDERING} \\
\hline
\textit{LOCAL\_ORDERING} \\
\_ \lll \_ : \textit{ClassName} \leftrightarrow \textit{ClassName} \\
\hline
\forall\, cn_1, cn_2 : \textit{ClassName} \bullet \\
\quad cn_1 \lll cn_2 \Leftrightarrow \\
\qquad \exists\, cn : \textit{ClassName} \bullet cn_1 \ll_{cn}{}^{+}\ cn_2 \\
\hline
\end{array}
$$

Note that the generalised inheritance ordering neither relates classes without subclasses to one another nor a class to itself.

The generalised inheritance ordering allows cycles. In the schema given in Figure 2.5, $A \ll_C B$ and $B \ll_D A$ hold. The generalised ordering gives both $A \lll B$ and $B \lll A$,

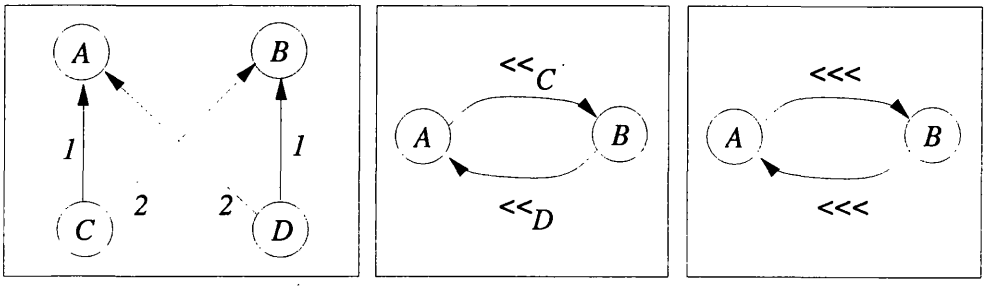

Figure 2.5: Cycles in Generalised Inheritance Ordering.

resulting in a cycle $A \lll B \lll A$. To resolve this problem a stronger notion of consistency is required and is described next.

2.8.4 Consistency of Schema Definition

Methods can now be ordered based on their argument types using the generalised inheritance ordering even without any knowledge of the static types of the actual arguments. A method is more specific than another one if at the first position where their argument types differ the argument type of the former method is more specific than the argument type of the latter method with respect to the generalised inheritance ordering. Method specificity is captured using the relation $\sqsubseteq$ as follows,

$$
\begin{array}{|l}
\textit{ORDERED\_SCHEMA} \\
\textit{INHERITANCE\_ORDERING} \\
\_ \sqsubseteq \_ : \textit{METHOD} \leftrightarrow \textit{METHOD} \\
\hline
\forall\, m_1, m_2 : \textit{METHOD};\ s_1, s_2 : \textit{SIGNATURE} \mid \\
\qquad s_1 = m_1.\textit{signature} \ \land\ s_2 = m_2.\textit{signature} \bullet \\
\qquad m_1 \sqsubseteq m_2 \Leftrightarrow \\
\qquad\quad \exists j : 1 \,..\, s_1.\textit{length} \bullet \\
\qquad\qquad s_1.\textit{argumentTypes}\ j \lll s_2.\textit{argumentTypes}\ j \\
\qquad\qquad \land \\
\qquad\qquad \forall i : 1 \,..\, (j-1) \bullet \\
\qquad\qquad\quad s_1.\textit{argumentTypes}\ i = s_2.\textit{argumentTypes}\ i \\
\hline
\end{array}
$$

Two methods are consistent if the more specific method with respect to method specificity ($\sqsubseteq$) also has a more specific result type with respect to type conformance ($\prec$). The specification for method consistency is given below,

CONSISTENT_SCHEMA

$$
\begin{array}{l}
ORDERED\_SCHEMA \\
\_\ isConsistentWith\_ : METHOD \leftrightarrow METHOD \\
\hline
\forall\, m_1, m_2 : METHOD;\ s_1, s_2 : SIGNATURE \mid \\
\qquad s_1 = m_1.signature \ \wedge\ s_2 = m_2.signature \bullet \\
\qquad m_1\ isConsistentWith\ m_2 \Leftrightarrow \\
\qquad\quad (\, m_1 \sqsubseteq m_2\,) \Rightarrow s_1.resultType \lessdot\!\!\lessdot s_2.resultType \\
\qquad\quad \wedge \\
\qquad\quad (\, m_2 \sqsubseteq m_1\,) \Rightarrow s_2.resultType \lessdot\!\!\lessdot s_1.resultType
\end{array}
$$

In *DATABASE_SCHEMA* it is asserted that methods directly related in a confusable set must be consistent.

DATABASE_SCHEMA

$$
\begin{array}{l}
PARTITIONS \\
CONSISTENT\_SCHEMA \\
\hline
\forall\, ms : allConfusableSets \mid \#ms > 1 \bullet \\
\qquad \forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet \\
\qquad\quad m_1\ isConfusableWith\ m_2 \Rightarrow m_1\ isConsistentWith\ m_2
\end{array}
$$

The consistency of confusable methods ensures that all methods involved in a cycle in the generalised inheritance ordering must have the same result type. A proof of this Theorem can be found in Appendix A.

2.9 Databases

A database can be defined as a consistent database schema and a set of variable names to which persistent objects are attached. Assume that all such variable names are drawn from a given set:

$$[VARIABLE\_NAME]$$

Each variable name represents a persistent object which can be a *base value*, *mono-object*, or *collection*. Variable names must be unique and therefore the binding between variable names and values is defined as a total function.

DATABASE

$$
\begin{array}{l}
DATABASE\_SCHEMA \\
persistentRoot\_ : VARIABLE\_NAME \rightarrow VALUE
\end{array}
$$

2.10 Static Type Checking

Type checking involves checking the static types of the actual arguments of a message against methods collected in a confusable set. Given the message name and the number of arguments, all the relevant confusable sets can be obtained using the *methodPartitions* function. These sets can then be searched for an applicable method that matches the given static argument types. Once a set with an applicable method is found, the search can stop as it is the only set containing all the possible applicable methods. Methods in this set are then sorted by method specificity before being compared with the argument types. The least general method that is applicable to the given argument types is selected and recorded. Its result type will then be used to check against the context in which the message is used.

The situation where no error occurs during type checking is considered first. The first condition for successful type checking is that the message name and the number of arguments do correspond to some method defined in the database schema.

RIGHT_ARGUMENT_LENGTH

$$
\begin{array}{l}
\Xi DATABASE \\
messageName? : METHOD\_NAME \\
argumentTypes? : \mathrm{seq}_1\ TYPE\_NAME \\
ms : \mathbb{F}\ METHOD \\
\hline
(\ messageName?, \#argumentTypes?\ ) \in \mathrm{dom}\ methodPartitions \\
ms \in methodPartitions(\ messageName?, \#argumentTypes?\ )
\end{array}
$$

The second condition for successful type checking requires that at least one method having the same name and taking the same number of arguments can actually be applied on arguments of the given types.

HAS_APPLICABLE_METHOD

$$
\begin{array}{l}
\Xi DATABASE \\
argumentTypes? : \mathrm{seq}_1\ TYPE\_NAME \\
ms : \mathbb{F}\ METHOD \\
\hline
\exists\, m : METHOD \mid m \in ms \bullet \\
\quad \forall\, i : 1\,..\,\#argumentTypes? \bullet \\
\qquad argumentTypes?\ i \prec\!\!\!\ll m.signature.argumentTypes\ i
\end{array}
$$

The set of confusable methods containing an applicable method is represented by *ms* in the two specifications above. Methods in *ms* that are confusable with the argument types of the message are then selected and sorted in descending order of method specificity ($\sqsubseteq$).

$$
\begin{array}{|l}
\textit{SORT\_CONFUSABLE\_METHODS} \\
\hline
\Xi \textit{DATABASE} \\
\textit{argumentTypes}? : \mathrm{seq}_1\ \textit{TYPE\_NAME} \\
\textit{ms} : \mathbb{F}\ \textit{METHOD} \\
\textit{oms} : \mathrm{seq}_1\ \textit{METHOD} \\
\hline
\mathrm{ran}\ \textit{oms} = \{\, m : \textit{METHOD} \mid m \in \textit{ms} \bullet \\
\qquad \forall\, i : 1 \mathinner{..} \#\textit{argumentTypes}? \bullet \\
\qquad\quad \exists\, t : \textit{TYPE\_NAME} \bullet \\
\qquad\qquad t \prec\!\!\!\prec \textit{argumentTypes}?\ i \\
\qquad\qquad \wedge \\
\qquad\qquad t \prec\!\!\!\prec m.\textit{signature.argumentTypes}\ i \,\} \\
\forall\, j, k : 1 \mathinner{..} \#\textit{oms} \mid j + 1 = k \bullet \\
\qquad \textit{oms}\ j \sqsubseteq \textit{oms}\ k \\
\hline
\end{array}
$$

The least general method is defined as the most specific method with respect to method specificity ($\sqsubseteq$) that is applicable to arguments of the given types.

$$
\begin{array}{|l}
\textit{LOCATE\_LEAST\_GENERAL\_METHOD} \\
\hline
\Xi \textit{DATABASE} \\
\textit{argumentTypes}? : \mathrm{seq}_1\ \textit{TYPE\_NAME} \\
\textit{oms} : \mathrm{seq}_1\ \textit{METHOD} \\
g : \mathbb{N}_1 \\
\textit{moreSpecificMethods}! : \mathrm{seq}\ \textit{METHOD} \\
\textit{resultType}! : \textit{TYPE\_NAME} \\
\hline
g \in 1 \mathinner{..} \#\textit{oms} \\
\forall\, i : 1 \mathinner{..} \#\textit{argumentTypes}? \bullet \\
\qquad \textit{argumentTypes}?\ i \prec\!\!\!\prec (\ \textit{oms}\ g\ ).\textit{signature.argumentTypes}\ i \\
\nexists\, i : (g + 1) \mathinner{..} \#\textit{oms} \bullet \\
\qquad \textit{oms}\ g \sqsubseteq \textit{oms}\ i \wedge \textit{oms}\ i \sqsubseteq \textit{oms}\ g \\
\exists\, i : 1 \mathinner{..} (g - 1) \bullet \\
\qquad (\ \textit{oms}\ i \sqsubseteq \textit{oms}\ g \wedge \textit{oms}\ g \sqsubseteq \textit{oms}\ i\ ) \\
\qquad \vee \\
\qquad \exists\, j : 1 \mathinner{..} \#\textit{argumentTypes}? \bullet \\
\qquad\quad \neg\ (\ \textit{argumentTypes}?\ j \prec\!\!\!\prec (\ \textit{oms}\ i\ ).\textit{signature.argumentTypes}\ j\ ) \\
\textit{moreSpecificMethods}! = \{\, 1 \mathinner{..} g \,\} \lhd \textit{oms} \\
\textit{resultType}! = (\ \textit{moreSpecificMethods}!\ g\ ).\textit{signature.resultType} \\
\hline
\end{array}
$$

The least general method from *oms* is selected by identifying its position g in the sequence. The first constraint restricts g to be a position in the sequence. The second constraint specifies that the method at position g is applicable to arguments of the given types. The third constraint asserts that methods after position g cannot form a cycle with the method at g in the method specificity relation. The last constraint asserts

that a method before position g either forms a cycle with the method at g in the method specificity relation or is not applicable to arguments of the given types. Note that methods of the latter kind are confusable with the argument types. Part of the sequence *oms*, up to position g, is returned as a sequence of potential applicable methods. The result type of the method at position g becomes the result type of the message.

The above specifications can now be combined to capture the error-free case of type checking. Note that a message indicating successful type checking is also included.

$$
\begin{aligned}
&PASS\_TYPE\_CHECKING \mathrel{\widehat=} \\
&\quad RIGHT\_ARGUMENT\_LENGTH \land \\
&\quad HAS\_APPLICABLE\_METHOD \land \\
&\quad SORT\_COFUSABLE\_METHODS \land \\
&\quad LOCATE\_LEAST\_GENERAL\_METHOD \land \\
&\quad [\, rep! : MESSAGE \mid rep! = Ok \,]
\end{aligned}
$$

The domain of *rep!*, *MESSAGE*, is defined below.

$$MESSAGE ::= Ok \mid WrongArgumentLength \mid NoApplicableMethod$$

There are two circumstances under which the result type of a message cannot be identified. The first case occurs when the message name does not correspond to any method or the number of arguments does not match that of any method with the same name.

WRONG_ARGUMENT_LENGTH

$$
\begin{aligned}
&\Xi DATABASE \\
&messageName? : METHOD\_NAME \\
&argumentTypes? : \mathrm{seq}_1\ TYPE\_NAME \\
&\hline
&(\, messageName?, \#argumentTypes? \,) \notin \mathrm{dom}\ methodPartitions
\end{aligned}
$$

When an error occurs, an empty list of methods will be returned and the result type of the message will be set to the root class. An error message is also produced.

NOTHING

$$
\begin{aligned}
&\Xi DATABASE \\
&moreSpecificMethods! : \mathrm{seq}\ METHOD \\
&resultType! : TYPE\_NAME \\
&\hline
&moreSpecificMethods! = \langle\rangle \\
&resultType! = root
\end{aligned}
$$

$$ERROR\_ONE \mathrel{\widehat=} WRONG\_ARGUMENT\_LENGTH \land NOTHING \land [rep! : MESSAGE \mid rep! = WrongArgumentLength]$$

The second case occurs when no applicable method can be found.

NO_APPLICABLE_METHOD

$$\begin{array}{l} \Xi DATABASE \\ messageName? : METHOD\_NAME \\ argumentTypes? : \mathrm{seq}_1\ TYPE\_NAME \\ \hline \nexists ms : \mathbb{F}\, METHOD \mid ms \in methodPartitions(messageName?, \#argumentTypes?) \bullet \\ \quad \exists m : METHOD \mid m \in ms \bullet \\ \quad\quad \forall i : 1 \mathinner{..} \#argumentTypes? \bullet \\ \quad\quad\quad argumentTypes?\ i \nprec m.signature.argumentTypes\ i \end{array}$$

$$ERROR\_TWO \mathrel{\widehat=} NO\_APPLICABLE\_METHOD \land NOTHING \land [rep! : MESSAGE \mid rep! = NoApplicableMethod]$$

The various cases can now be combined to provide a full picture of type checking. The hiding operator $\setminus$ is applied to *TYPE_CHECKING* to remove the variables g, *oms*, and *ms* from the declaration part and to existentially quantify them in the predicate part. This signifies that the variables are used for intermediate values generated during the computation.

$$FAIL\_TYPE\_CHECKING \mathrel{\widehat=} ERROR\_ONE \lor ERROR\_TWO$$

$$TYPE\_CHECKING \mathrel{\widehat=} PASS\_TYPE\_CHECKING \lor FAIL\_TYPE\_CHECKING$$

$$STATIC\_TYPE\_CHECKING \mathrel{\widehat=} TYPE\_CHECKING \setminus \{ g, oms, ms \}$$

Consider the schema given in Figure 2.1, $m : A \times C \to C$ and $m : B \times D \to D$ are in the same confusable set and m of B is more specific ($\sqsubseteq$) than m of A. The static types of the arguments in the program fragment are A and C. Since $B \nprec A$ and $D \nprec C$, m of B is not applicable and therefore m of A is selected giving C as the result type.

In Figure 2.2, the static type of the argument is C and both methods m's are applicable. $m : A \to A$ will be selected as it is less general than m of B. Consequently type checking will fail because of the invalid assignment statement. It is because an object of class A cannot be assigned to a variable of class B because A is not a subclass of B.

2.11 Dynamic Binding

At run-time the most specific method, with respect to the actual argument types, is chosen. It is only necessary to check the sequence of methods recorded during static type checking. Applicable methods are examined and the most specific method according to the local inheritance order as determined by the argument types is returned. Using this strategy, one can provide the flexibility of using the most specific method during run-time whose argument types respect that of the least general method used for type checking.

DISPATCHING

$$
\begin{array}{l}
\Xi DATABASE \\
methodList? : \mathrm{seq}_1\ METHOD \\
argumentTypes? : \mathrm{seq}_1\ TYPE\_NAME \\
oms : \mathrm{seq}_1\ METHOD \\
g : \mathbb{N}_1 \\
methodChosen! : METHOD \\
\hline
\mathrm{ran}\ oms = \{\, m : METHOD \mid m \in \mathrm{ran}\ methodList? \bullet \\
\qquad \forall i : 1 \mathinner{..} \#argumentTypes? \bullet \\
\qquad\quad argumentTypes?\ i \prec\!\!\!\prec m.signature.argumentTypes\ i \,\} \\
g \in 1 \mathinner{..} \#oms \\
\forall j, k : 1 \mathinner{..} \#oms \mid j + 1 = k \bullet \\
\quad \exists i : 1 \mathinner{..} \#argumentTypes? \bullet \\
\qquad (\, oms\ j \,).signature.argumentTypes\ i \ll_{argumentTypes?\ i} \\
\qquad\quad (\, oms\ k \,).signature.argumentTypes\ i \\
\qquad \vee \\
\qquad \forall l : 1 \mathinner{..} (i - 1) \bullet \\
\qquad\quad (\, oms\ j \,).signature.argumentTypes\ l = \\
\qquad\qquad (\, oms\ k \,).signature.argumentTypes\ l \\
methodChosen! = head\ oms
\end{array}
$$

$$
DYNAMIC\_BINDING \mathrel{\widehat{=}} DISPATCHING \setminus \{\, g, oms \,\}
$$

The sequence of methods recorded during type checking is represented by *methodList?*. The actual argument types are represented by *argumentTypes?* as a sequence of type names. The first constraint collects all the applicable methods from *methodList?* into *oms*. The second constraint restricts g to be a position of *oms*. The third constraint orders the methods in *oms* using the local inheritance orderings as determined by the actual argument types. The most specific method which is placed at the beginning the *oms* is returned.

In Figure 2.1, if variables a and c actually contain objects of class B and D respectively, the method $m : B \times D \rightarrow D$ will be chosen; otherwise, the method m of A will be used.

2.12 Reasoning about Z Specifications

Properties of the specification, of the reference data model, and of the approach used for static type checking can be studied through formal reasoning. For example, the following properties have been proved formally from the Z specification,

- The computation of *applicableMethods* in *SCHEMA* does terminate (proof by induction).
- The search for an applicable set of confusable methods in *PASS_TYPE_CHECKING* can stop once a set is found (proof by contradiction).
- An object cannot simultaneously be an instance of two disjoint leaf classes according to the function *typeOf* (proof by contradiction).
- Local inheritance ordering is sufficient for ordering confusable methods in a multiple inheritance environment.
- Methods forming a cycle under the method specificity relation do not affect static type checking and will be correctly bound at run-time.

The last property depends on the property that methods forming a cycle in the method specificity relation must have the same result type. A proof of the latter property is given in Appendix A.

Note that the specification defined so far captures only the essential parts of the reference data model that are required for the study of query processing in the coming chapters.

2.13 The Running Example

The example database is a simplified university administration system that records information about students and staff members of a university, its academic departments and courses. The relationships between classes defined in the schema are shown in Figure 2.6.

The class *Person* has two subclasses: *Student* and *Staff*. *VisitingStaff* is a subclass of *Staff*. *Tutor* inherits from both *Student* and *Staff* to represent students doing part-time teaching. Every person and academic department is given an address which is an object of class *Address*. A student can have a principal supervisor, a second supervisor, and so forth. It is therefore modelled as a list of staff members. Every staff member and student are associated to an academic department of class *Department* via *department* and *major* respectively. Courses given by each staff member and taken by each student are also recorded. They are represented by set-valued methods *teaches* and *takes*. A course may have a set of prerequisite courses (*prerequisites*) and is administered by one or more academic departments (*runBy*). A course is an instance of the class *Course*.

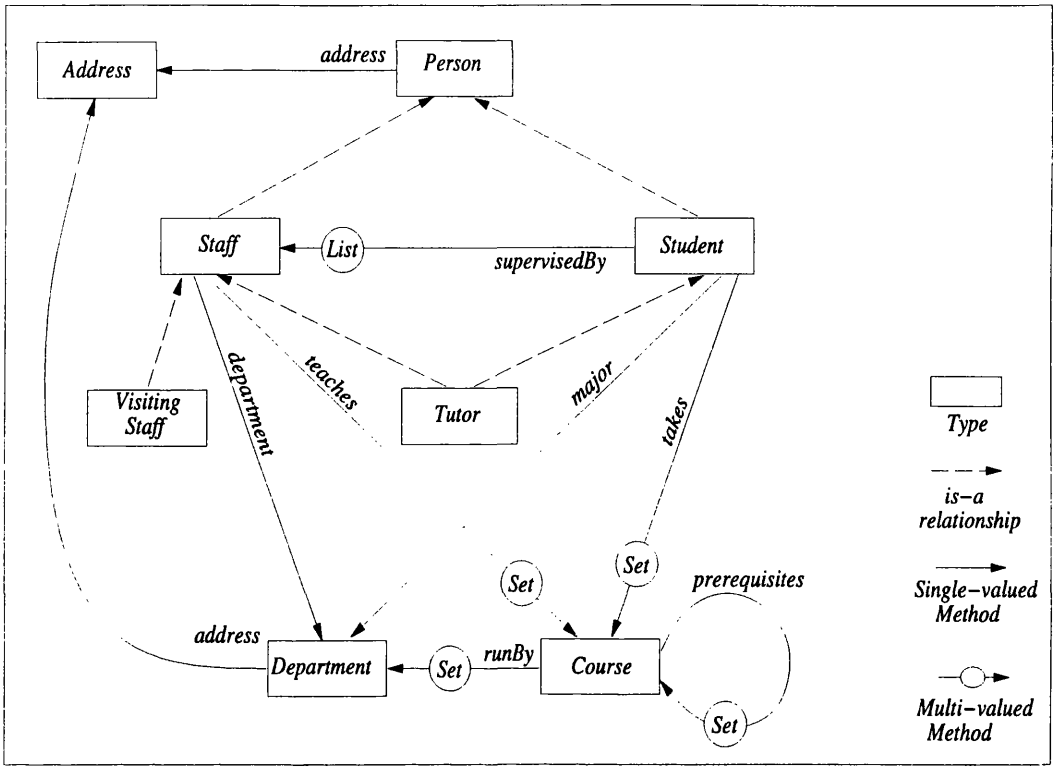

Figure 2.6: Simplified Schema Diagram.

The schema definition is given in Figure 2.7. In order to keep it simple, only the relevant method signatures are given, attributes and method implementations are omitted. *Entity* is the root class. The calculation of the salary of a tutor is different from that of a staff member. This variation is captured by giving an overloaded method *salary* to *Tutor*. Also recorded is the percentage weights of assessments given in each course and the number of credits each course is worth. It is assumed that the database contains six set collections: *Persons*, *Departments*, and *Courses*, containing instances of their corresponding classes that are members of the university; and *StaffMembers*, *Students*, and *Tutors*, containing instances of the corresponding classes that are members in the Science Faculty.

2.14 Discussion

Data models from four prominent object-oriented database systems are chosen to compare with the reference data model. These models are selected because their query languages will be studied extensively later in the thesis. This comparison therefore serves as background information for later chapters. A summary of the comparison is given in Table 2.1.

[3] In [BDK92] set is the only collection class.

```
Class Person isa Entity
methods
   name :→ String,
   address :→ Address,
   age :→ Integer.

Class Staff isa Person
methods
   department :→ Department,
   teaches :→ Set of Course,
   salary :→ Integer.

Class Student isa Person
methods
   major :→ Department,
   supervisedBy :→ List of Staff,
   takes :→ Set of Course.

Class Tutor isa Staff, Student
methods
   salary :→ Integer.

Class VistingStaff isa Staff.

Class Department isa Entity
methods
   name :→ String,
   address :→ Address.

Class Course isa Entity
methods
   code :→ String,
   runBy :→ Set of Department,
   prerequisites :→ Set of Course,
   assessments :→ Bag of Integer,
   credits :→ Integer.

Class Address isa Entity
methods
   street :→ String,
   city :→ String.

Database is
   Persons : Set of Person,
   Departments : Set of Department,
   Courses : Set of Course,
   StaffMembers : Set of Staff,
   Students : Set of Student,
   Tutors : Set of Tutor.
```

Figure 2.7: Simplified Schema Definition.

ONTOS and O_2 support extents as an option. IRIS and ORION provide an extent for each class. In ORION, the extent of a class does not include extents of its subclasses. The reference data model is the only data model supporting multi-methods and static type checking at the same time. The combination of these two features is however supported to various extents in some programming languages.

CLOS [BDG+88] introduces the notion of multi-methods. Its use of local inheritance ordering is very similar to that of the reference data model. However, type checking in CLOS is done at run-time and hence is easier as more information is available at run-time. Kea [MHH91] uses the ISA relationship to determine method specificity. In addition, the lexical order of method definitions is also taken into account. The combined resultant ordering is nevertheless only a partial ordering. In other words, multiple inheritance cannot be fully supported. Unlike other systems, Cecil [Cha92] orders multi-methods using all their argument types in its entirety. However, only a partial ordering based on the ISA relationship is used and consequently the order is rather restrictive.

A reference data model, which serves as the basis of the investigation, has been presented in this chapter. So far only the "structural" part of the reference data model has been described. The intention is to develop a high-level query language to serve as the "op-

Data Model	ONTOS	IRIS	ORION	O_2	Reference
Base Values	✓	✓	✓	✓	✓
Tuples		✓		✓	
Complex Objects	✓	✓	✓	✓	✓
Object Identity	✓	✓	✓	✓	✓
Encapsulation	✓				✓
Method Calling	✓	✓	✓	✓	✓
Overloaded Methods	✓	✓	✓	✓	✓
Multi-Methods					✓
Classes	✓	✓	✓	✓	✓
Class Extents	optional	✓	✓	optional	
Class Hierarchy	✓	✓	✓	✓	✓
Multiple Inheritance		✓	✓	✓	✓
Static Type Checking	✓				✓
Dynamic Binding	✓		✓	✓	✓
Collection Classes	set, list, others	set, bag	set	set, bag, list[3]	set, bag, list

Table 2.1: Comparison of Object-oriented Data Models.

erational" part of the reference data model. The next chapter identifies the requirements of such an object-oriented query language.

Chapter 3

Query Language Requirements

Many object-oriented query languages [Ser87, CDV88, Bee88, DLR88, CDLR89, BM89, BCD90, Kim90, BTA90, Ont91c, DGJ92, KKS92] have been implemented and proposed. Some of these query languages are designed particularly for object-oriented databases, e.g. LIFOO [BM89] and ORION [Kim90]. Many are, however, adapted from other areas: the relational data model and its extensions, e.g. ONTOS SQL [Ont91c]; semantic data models, e.g. OSQL [Bee88]; and object-oriented programming languages, e.g. OPAL [Ser87]. All of them, however, could be improved in one way or another. This chapter establishes a set of functional requirements for object-oriented query languages which can be used to evaluate, compare, and improve existing query languages as well as to direct the design of new languages.

The organisation of this chapter is as follows. Section 3.1 argues for the need for establishing a set of functional requirements for object-oriented query languages. Section 3.2 describes the requirements. Section 3.3 addresses related issues that are not included in the requirements. Section 3.4 presents a summary of evaluating four query languages using the requirements. Section 3.5 concludes.

3.1 Introduction

Relational completeness was proposed in [Cod72] and since then it has served as the yardstick for evaluating the expressive power of relational query languages. Later studies of the generalisation of the relational model extended the relational algebra with extra operations. For instance, *replace* (applying a function over the elements of a collection), *set-collapse* (given a collection of collections return all the elements contained in the "nested" collections), and *powerset*, were introduced in [AB93] to characterise the expressive power of query languages for nested-relational and complex-object models, such as $\neg$1NF [RKB87], NF^2 [PD89], and VERSO [SAB$^+$89]. With the advent of newer data models supporting richer constructs, new definitions of completeness are constantly sought for.

So far, no definition of completeness has been proposed for object-oriented data models. Worse still, the situation is unlikely to change for some time. Attempts have been made to tackle the problem by using different notions of completeness which are independent of data models [ABGvG89], but interpreting them in terms of query language operations is not at all straight forward. Not having a formal definition of completeness makes it difficult to evaluate and compare object-oriented query languages objectively.

Many opinions have been expressed about the central and fundamental issues of object-oriented query languages [Ban89, Kim89, Kim92, BNPS92]. Relevant inputs can also be found from sources taking a slightly different standpoint. For example, an evaluation framework for query algebras is proposed in [YO91] and many of the criteria are applicable to high-level query languages. Language facilities for multimedia data are studied in [Man91]. Almost all the requirements identified there are equally valid for object-oriented query languages in general. It is encouraging that there is a consensus of opinions from these sources.

However, this pool of ideas has several limitations. Firstly, collection operations are not sufficiently characterised nor is the usability aspect of high-level query languages. Secondly, the requirements are stated in rather esoteric terms hence are open to misunderstanding. Thirdly, some of the requirements cannot be measured objectively. Further discussion on collection operations and usability is given in Section 3.2 and Section 3.4. Examples of the second and third problems can be found in Section 3.3. The aim of introducing a new set of requirements is to provide a set of direct and measurable criteria as well as to make up for overlooked issues. The guideline adopted is to include only features that can be expressed at the language level. This shift of emphasis can be compared with the development of database performance benchmarking where the TPC-A benchmark was introduced to replace the DebitCredit benchmark precisely because the latter is vulnerable to interpretations by the implementator [DBCRW92].

3.2 Functional Requirements

The requirements can be classified into four dimensions: *support of object-orientation*, *expressive power*, *support of collections*, and *usability*. Each dimension is defined in terms of a number of criteria.

Support of object-orientation measures the support given to the intrinsic properties of object-oriented data models. There is an almost unanimous agreement in the literature about features under this category. They include

- object identifiers
- method calling
- complex objects
- class hierarchy
- dynamic binding

In the object-oriented paradigm, objects are identified by unique and immutable object identifiers which are independent of the "contents" or composition of the objects. To support objects, a query language needs to operate on object identifiers, for example, the equality over object identifiers.

Objects are *encapsulated*, meaning that their "contents" cannot be accessed directly and all accesses must be done via methods defined for the objects. The association of specific methods to objects is a fundamental tenet of the paradigm, the use of methods in a query should therefore be supported.

In contrast to the simple attributes of a tuple in the relational model, an object can be perceived as a complex entity. Applying a method on such a complex object can result in the return of a base value, an object, or a collection. A query language supporting method calling should therefore also accommodate results of different types.

The class hierarchy defines a classification scheme based on specialisation over classes. This naturally leads to the adoption of the substitutional semantics which conceals the differences of objects originated from different classes along the same specialisation chain. On the other hand, the class hierarchy contains useful information regarding the classification of objects which may form the basis of a query. A query language should therefore provide a mechanism with which the classification information can be exploited.

The class hierarchy also introduces the notion of inheritance where methods defined in a superclass are inherited by all its subclasses. Moreover, method overloading allows different methods to be given the same name. Given the substitutional semantics and the possibility of method overloading, the selection of a method can only be determined dynamically. A query language should therefore support dynamic binding of methods or behave as if methods were dynamically bound.

Expressive power examines the ability to explore and synthesize complex objects and collections. Attention is mainly drawn to the manipulation of individual objects. There is quite a reasonable consensus in the literature regarding features in this category which are listed below,

- multiple generators
- dependent generators
- returning new objects
- nested queries
- quantifiers
- relational completeness
- nested relational extension
- recursion

A query often involves one or more collections. The ability to specify more than one *domain* collection - using multiple generators - in a query is as natural and important as in earlier data models. Not supporting multiple generators will result in a more procedural query language relying heavily on query nesting and query functions, if they are supported. Consequently queries are more difficult to express - a good example is LIFOO [BM89].

A collection can be returned as the result of a method call. To query such a "nested" collection, a query language should be able to express dependency between generators. Generally speaking, in the absence of dependent generators, the fact that an object is an element of a nested collection has to be "re-established" resulting in more verbose queries, for example, see OSQL [Lyn91].

So far, there has been no convincing argument from the modelling perspective about restricting a query language to return only existing objects. Here a query language is required to allow new objects to be created in a query. It is not required that the corresponding classes are created along with the objects. In other words, closure at the instance level should be respected while closure at the class level is not required. To be more precise, no operations for the creation of new classes or the manipulation of the class hierarchy are required. It is so decided because dynamic class creation is still an outstanding problem with no satisfactory solution. Detailed discussion of this controversial issue is given in Chapter 8 where the support of views is discussed.

Nested queries are crucial in the construction of new objects especially complex objects. It has been shown that many nested queries which appear only in the *where* clause (the filters) can be eliminated from SQL queries [Kim82, GW87]. In an object-oriented query language supporting free nesting of queries, it it not obvious how nested queries can be eliminated without other language constructs such as query functions. Query nesting can also be considered as an issue of generality.

Quantifiers can simplify queries and provide optimisation opportunities. Quantifiers can be simulated in many query languages; however, their optimisation always involves matching of large patterns, e.g. [Klu82], which increases the search space for optimisation. Quantifiers can significantly simplify the manipulation of different kinds of collections as shown in Chapter 4.

Object-oriented data models subsume the relational model, so an object-oriented query language should similarly subsume relational completeness. One possible definition of this requirement is that if the data are relations, a query language should be able to express all queries that can be expressed in the relational algebra. However, a more general definition would be more appropriate for comparing query languages for object-oriented databases. Basically, the concept of a relation being a set of tuples can be replaced by a collection of objects. A query language should therefore be able to express whatever can be expressed in the relational algebra for collections and objects. Inevitably, the relational algebra operations will be more restrictive if tuples are not supported and should behave differently for different collection kinds.

Studies of the generalisation of the relational model result in the introduction of three extra operations: *replace*, *set-collapse*, and *powerset*. It has been shown that *powerset* incurs superexponential complexity [HS88] which justifies its omission from the requirement list. The other two operations should be supported and their definitions can be similarly generalised as the relational algebra operations.

In the object-oriented paradigm, cyclic relationships can be defined via one or more methods. Some form of recursion, for instance, transitive closure, should be supported to enable cyclic relationships to be explored.

Many object-oriented database systems support more than one collection kind. New features are required to manipulate these collections. The accent is to find a good set of generic operations that behave consistently for different collection classes. Equally important is the mixing of and conversion between different collection classes. Support of collections looks into the following features:

- collection literals
- collection equality
- aggregate functions
- positioning & ordering
- occurrences & counting
- converting collections
- combining collections
- mixing collections

In some data models including the reference data model, collections are represented as objects hence their comparison is based on object identifiers. However, collections are very often characterised by their contents and behave like base values. This suggests that collections should be allowed to have dual behaviour. One aspect of this duality is to allow collection literals to be expressed. Collection literals can be simulated in some query languages; however, providing direct support simplifies queries as has been shown in SQL

[Dat87]. Using the same argument, it should be possible to compare two collections based on their elements instead of their identifiers.

Aggregate functions return a value from a collection and have been shown very useful in earlier data models. When ordered collections are supported, a query language should be able to express queries related to a position in the order and the ordering between two elements. When collections are allowed to have duplicates, a query language should be able to return objects with a particular number of occurrences and to count the number of occurrences of an object. It is also important to allow collections to be combined, converted, and mixed within a query.

Usability focuses on the ease of use of a query notation which is essential to the success of a high-level query language. The criteria in this category are

- local definitions
- query functions

Long path expressions are not uncommon in object-oriented query expressions. To avoid repeating long path expressions, "shorthands" can be introduced using local definitions. Complicated queries are easier to express in an incremental fashion. Query functions allow a complicated query to be broken down into smaller and more comprehensible subqueries.

3.3 Related Issues

The previous discussion focuses on query language features without addressing the impact of data model on query languages. There has been much discussion about the advantages and disadvantages of supporting extents [Kim89, ABD+90]. Extents create security problem as all instances of a class can be accessed via the class extent and access control on individual objects is difficult and prohibitively expensive. Application modelling often does not require the use of class extents. The provision of class extents has a great impact on what a query language can retrieve. Querying a database becomes easier as every class extent provides an entry point to the database and every object is guaranteed to be directly accessible from at least one class extent. The result is a simpler query language and more optimisation opportunities. Some object-oriented data models, including the reference data model, do not support class extents. To have a set of requirements that are generally applicable, the existence of class extents cannot be assumed. The requirements given in the previous section are derived with no assumption of class extents. For data models supporting class extents, some of the requirements will become superfluous. For example, using multiple generators together with the membership test on collections, dependent generators can be simulated and the class hierarchy can be supported.

Different kinds of equality have been introduced for testing the equivalence of objects based on their contents so that optimisation can be done with more flexibility. The deep-sensitive equivalence rules described in [SZ89] are very complicated and significantly increase the search space of the optimiser. The essence is that these different kinds of equality may be useful in query algebras but their necessity in high-level query languages is questionable.

Many other features of object-oriented query languages have been suggested [Dat84, BZ87, Ban89, YO91, Man91, US92]. They are useful guidelines for the design of query languages. However, using them as requirements for evaluating query languages is less effective and their assessment can be difficult.

- Simple
- General
- Application Independent
- Strong Typed
- Optimisable
- Data Administration
- Rules and Triggers
- Consistent
- Closed
- Orthogonal
- Formal Semantics
- Null Values
- Integrity Constraints
- Versions
- Elegant
- Adequate
- Well Integrated
- Efficient
- Extended Facilities
- Computational Complete
- Schema Evolution

A consistent notation encourages similar concepts or problems to be expressed in similar ways. A query language is general if it allows free composition of constructs and does not impose arbitrary restrictions. Closure refers to the fact that the result of a query can be similarly manipulated by the query language. In a mono-type model. like the relational model, it is a necessary and sufficient requirement. When multiple types are supported, like in object-oriented data models, closure becomes a necessary but not sufficient requirement. Adequacy provides the sufficiency by requiring a query language to operate on all types supported in the data model. Orthogonality is usually used in discussing persistence meaning that a query language can work on both persistent as well as transient data.

3.4 An Evaluation of Existing Query Languages

In this section four well-known query languages are evaluated using the proposed requirements. They are the IRIS [LK86, Bee88, FAC+89, Lyn91], ORION[Kim90], ONTOS [Ont91a, Ont91b, Ont91c], and O_2[Alt89, BDK92] query languages. These query languages are chosen as representative languages mainly because they are well-reported and the most referenced in the literature. The result of the evaluation is summarised in Table 3.1 to Table 3.4.

Method calling is supported by all the four query languages. OSQL, ORION and O_2SQL also support direct access to attributes. OSQL supports the class hierarchy via

	ONTOS SQL	OSQL	ORION	O_2SQL
Object Identifiers	✓	✓	✓	✓
Method Calling	✓	✓	✓	✓
Complex Objects	✓	✓	✓	✓
Class Hierarchy		✓	✓	✓
Dynamic Binding	✓	✓	✓	✓

Table 3.1: Support of Object-Orientation.

class extents and membership test. ORION provides four constructs to support the class hierarchy: (1) class extents and the membership test operation *is-in*; (2) the operations * (meaning including instances of all subclasses), *union*, and *difference* over class extents to form class extent expressions; (3) specifying the class of the object returned by a method call using *class*, this specification can be sandwiched between method calls within the same path expression; and (4) specifying the class of objects used and returned in a recursive query using *is-a*. The ONTOS and O_2 data models support class extents only as an option. ONTOS SQL does not support the class hierarchy properly in its current form. In other words, support given to the class hierarchy partly depends on how the schema is defined. It is however possible to extend ONTOS SQL to support the class hierarchy using the available interfaces for collection classes and the database.

	ONTOS SQL	OSQL	ORION	O_2SQL
Multiple Generators	✓	✓	✓	✓
Dependent Generators			✓	✓
Returning New Objects				
Nested Queries		✓		✓
Existential Quantifier		✓	✓	✓
Universal Quantifier		✓	✓	✓
Selection	✓	✓	✓	✓
Projection	✓	✓		✓
Cartesian Product	✓	✓		✓
Union		✓	✓	✓
Differ		✓	✓	✓
Set-collapse		✓	✓	✓
Replace	✓	✓	✓	✓
Recursion				✓

Table 3.2: Expressive Power.

IRIS supports class extents and hence as explained in Section 3.3 the support of dependent generators in OSQL is not strictly necessary. On the other hand, the expressive power of ONTOS SQL suffers badly because class extents are optional and dependent generators are not supported. None of the query languages supports the return of new objects. It is a result of the limitation of current technology since creating new objects is

such an expensive operation that can significantly slow down query processing. ONTOS SQL can only return either a string or a list of strings, while OSQL and O_2SQL often use tuples to return new "objects". ORION does not seem to offer a solution in this aspect. Nested queries can only appear in filters of an OSQL query. Generally speaking a nested query in a generator can always be eliminated and hence a nested query is most useful when it is used in filters or the result expression.

Quantifiers can be simulated provided that membership test and the cardinality operations of collections are available, and nested queries are supported. ONTOS SQL does not support nested queries and hence cannot simulate quantifiers. OSQL can simulate the use of quantifiers in filters. If quantifiers are used elsewhere "foreign" functions - implemented in a programming language - can be employed.

ONTOS SQL does not support *union* and *differ*. ORION does not seem to support *projection* and the proposal [Kim90] did not make it clear. It cannot return new objects or tuples (tuples are not supported by its data model) and therefore cannot express *cartesian product*. O_2SQL provides *differ* for sets but not lists. ONTOS SQL does not support *set-collapse*. For the other languages, *set-collapse* can be performed in various ways including implicit flattening.

The four query languages all support some form of *replace*. Functions can be used in ONTOS SQL, OSQL, and O_2SQL while methods can be used in all of them. ORION supports traversal recursion that involves traversal of a cyclic relationship; however, it does not support computational recursion where computation is done along the traversal of a cyclic relationship.

	ONTOS SQL	OSQL	ORION	O_2SQL
Collection Literals	✓	✓	✓	✓
Collection Equality	✓	✓	✓	✓
Aggregate Functions				✓
Positioning	✓	-	-	✓
Ordering		-	-	✓
Occurrences		✓	-	✓
Counting		✓	-	✓
Converting Collections			-	✓
Combining Collections		✓	-	✓
Mixing Collections	✓	✓	-	✓

Table 3.3: Support of Collections.

ONTOS SQL supports only set literals that can appear only in generators. OSQL does not support lists and ORION supports only sets, therefore some entries in the table are not applicable to them. They are marked by a dash (-) in the table. Aggregate functions can be supported using foreign functions in the case of OSQL. O_2SQL can decide whether one element precedes another element in a list with the help of a set literal containing all

the positions of the list. It is possible for OSQL and O_2SQL to return objects given the number of occurrence in a collection though in a rather distorted way. Nevertheless, it is simpler for them to return the number of occurrence of a given object. ONTOS SQL does not support conversion between collection classes. The result of a query is either a string or a list of strings. With OSQL the result of a query is always a bag but duplicates in a bag can be eliminated. For O_2SQL, if all the generators are drawn from the same collection kind the result will be of the same kind; otherwise the result is a bag. The function *magic* can turn a set into a list while the function *listoset* and the keyword *distinct* and *unique* turn a list into a set.

	ONTOS SQL	OSQL	ORION	O_2SQL
Local Definition			✓	
Query Function		✓		✓

Table 3.4: Usability.

ONTOS SQL does not support local definitions or query functions. OSQL supports query functions that can appear only in generators and local definitions are not supported. ORION does not support query functions while O_2SQL does not support local definitions.

3.5 Summary

Example queries of the four query languages evaluated can be found in [CHT92b, CHT93a]. Results of the evaluation of other query languages, e.g. EXCESS (EXODUS [CDV88]), CQL++ (ODE [DGJ92]), OQL[X] (Zeitgeist [BTA90]), and XSQL [KKS92], can be found in [CT93, CTW95]. The new SQL3 proposal [Kul93] also includes many of the requirements. Existing object-oriented query languages can be improved along the directions suggested by the requirements. It is also hoped that new query language design can benefit from this set of requirements.

Comprehensions have been demonstrated to be well integrated with programming languages and to have desirable features that are worth exploring in the object-oriented setting. The next chapter introduces a new query language, *object comprehensions*, which is based on comprehensions and satisfies all the requirements presented in this chapter.

Chapter 4

Object Comprehensions

A new query notation called *object comprehensions* is introduced in this chapter. It is based on *list comprehensions* [PJ87], which has been argued to be a good query notation for being clear, concise, powerful, and optimisable [Tri91]. Object comprehensions extend list comprehensions by combining and improving features found in existing object-oriented query languages so as to provide a consistent and general query notation for object-oriented databases. Furthermore they incorporate new features that are missing from existing query languages: (1) treating the class hierarchy as a classification scheme hence allowing selection to be based on such classification; (2) using quantifiers to provide a consistent interface to collection classes hence reducing the syntactic complexity of collection operations; and (3) allowing the resultant collection class to be specified hence facilitating a "complete" specification and resulting in more comprehensible queries. Optimisation opportunities have been identified for some of the new features. Some optimisations allow syntax-level transformation at compile-time while others suggest simplification at run-time. These optimisations not only subsume previous work of the same kind [Str90] but also include many new optimisations.

The organisation of this chapter is as follows. Section 4.1 describes the development of object comprehensions. Section 4.2 presents the syntax of object comprehensions. Section 4.3 demonstrates object comprehensions using a set of example queries. Section 4.4 discusses optimisation of the new features. Section 4.5 concludes.

4.1 Comprehensions: Past & Present

4.1.1 Set, List & Collection Comprehensions

In mathematics the set of squares of all the odd numbers in a set s is conventionally written:

$$\{\, square\ x \mid x \in s \wedge odd\ x \,\}$$

This standard mathematical notation for sets was the inspiration for *comprehensions.* Comprehensions first appeared as *set comprehensions* in an early version of the programming language NPL. This language later evolved into Hope [BMS80] but without comprehensions. Later *list comprehensions* were included in KRC [Tur81], which was also the first to utilise the now familiar set-based syntax. List comprehensions have since been incorporated into several popular functional languages, e.g. Miranda [Tur85] and Haskell [HW90]. A full description of list comprehensions can be found in [PJ87].

Using list comprehensions the above mathematical expression can be written as

$$[\, square\ x \mid x \leftarrow s;\ odd\ x \,]$$

where s stands for a list instead of a set.

Recently, list comprehensions have been generalised to *collection comprehensions*, which provides a uniform and extensible notation for expressing and optimising queries over many collection classes including sets, bags, lists, trees, ordered sets, and so forth [WT91]. The most significant benefit is that, although each primitive operation will require a separate definition for each collection class, only one query notation is needed for all these collection classes; besides, a single definition is all that is required for higher-level operations defined in terms of collection comprehensions. In other words, it significantly reduces the syntactic complexity of the query notation.

Using collection comprehensions the same query can be written as

$$[\, square\ x \mid x \leftarrow s;\ odd\ x \,]_{set}$$

and with *object comprehensions* the above query can be written as

$$Set[\, x \leftarrow s;\ odd\ x \mid square\ x \,]$$

Each collection comprehension query can only involve one kind of collections. Object comprehensions generalise this to allow collections of different kinds to appear in the same query.

The result of evaluating this object comprehension query is a new collection, precisely a set, computed from the existing collection s of class *Set of Integer.* The elements of the new collection are determined by repeatedly evaluating *square x*, as controlled by the qualifier *odd x.* Since the result of *square x* is of type *Integer* the elements in the resultant set are therefore of the same type. The change of the position of the result expression in object comprehensions serves to provide a simpler scoping rule.

In general, a qualifier can be a *filter*, *generator*, or *local definition.* A filter is just a boolean-valued expression expressing a condition that must be satisfied for an element to be included in the result. An example of a filter was *odd x* above, ensuing that only odd values of x are used in computing the result. A generator of the form $V \leftarrow E$, where E is a collection-valued expression, makes the variable V range over the elements of the collection. An example of a generator was $x \leftarrow s$ above, making x range over the elements

of the set s. A local definition of the form N *as* E, introduces a symbolic name N for the value of the expression E.

The meaning of object comprehensions can be understood using nested loops. A good analogy is to think of generators as nested loops where outermost corresponds to leftmost, filters as conditions of an if statement, and local definitions as assignment to new variables. The previous query can be understood as resulting in the set r computed as follows:

$r := Set\{\}$
for each x in s
  if *odd* x then
    $r :=$ r *union* $Set\{$ x $\}$

It has to be stressed that this analogy is to clarify scoping rules and the meaning of object comprehensions; it is not the way to implement object comprehensions efficiently.

4.1.2 Other Extensions & Implementations

An extension to support local definitions in list comprehensions was suggested in [Ham90]. Side-effecting qualifiers were proposed in [GOPT92]. They permit data to be manipulated by side-effects in addition to being queried. The strong point is that such queries can still be optimised. The advantage of comprehensions over SQL, on which many object-oriented query languages are based, becomes clear when side-effecting qualifiers are taken into consideration. It is difficult to see how SQL-based languages can be extended in a similar way to cope with side-effects.

List comprehensions are also included in a new functional database language called PFL [SP91]. In P/FDM [PG90], DAPLEX queries are translated to an abstract form of list comprehensions with which optimisation is carried out. The authors commented that list comprehensions allowed queries to be expressed declaratively while DAPLEX had a navigational style of querying. It is interesting that the optimisation rules which are similar to those in [Tri89] are defined at the abstract comprehension level.

List comprehensions have also been applied to imperative languages such as an experimental version of PS-algol [TCH90]. The AGNA database programming language is evaluated on a dataflow multiprocessor and uses parallel list comprehensions to process database queries at speeds comparable with other multiprocessor database machines [NH91]. SPL is a language that uses comprehensions to evaluate queries in parallel over a distributed database [KMK90]. The new version of Napier [MBCD89] about to be released supports collection comprehensions.

4.1.3 Simplicity, Power & Optimisation

It was argued in [Tri91] that comprehensions are a good query notation for being concise, clear, expressive, and easily optimised. The essence of the argument is as follows.

Comprehensions are concise because they are a declarative specification of query. Comprehensions are clear because they are composed of consistent and general constructs. In [PJ87] the efficiency of list comprehensions was proved by showing that they perform the minimum number of *cons* operations required to produce the result list. More importantly, for each of the well-known optimisation strategies on relational queries, there exists an equivalent list comprehension transformation [TW89] and two of the transformations were demonstrated [TCH90].

4.2 Syntax of Object Comprehensions

The syntax of object comprehensions is given below where terminal symbols are underlined.

Expression	::=	*Expression* union *Expression*
	|	*Expression* differ *Expression*
	|	*Comprehensions*
	|	*Literal*
	|	*Path*
	|	*Call*
	|	*Aggregation Expression*
Aggregation	::=	size
Comprehensions	::=	let *Function* in *Expression*
	|	*Collection* [*Qualifier* | *Expression*]
Collection	::=	Set
	|	Bag
	|	List
Qualifier	::=	Λ
	|	*Generator*
	|	*Filter*
	|	*Definition*
	|	*Qualifier* ; *Qualifier*
Generator	::=	*Identifier* ← *Expression*

Table 4.1: Abstract Syntax of Object Comprehensions.

Filter ::= *Filter* and *Filter*
| *Filter* or *Filter*
| not *Condition*
| *Condition*

Condition ::= (*Filter*)
| *Quantified Operator Quantified*
| *Expression* hasClass *Type*
| *Expression* hasClass *Type* with *Filter*

Quantified ::= *Expression*
| *Quantification Expression*

Operator ::= $\underline{=}$
| $\underline{\sim=}$
| $\underline{<}$
| $\underline{<=}$
| $\underline{>}$
| $\underline{>=}$
| $\underline{==}$
| $\underline{\sim==}$

Quantification ::= some
| every
| just *Expression*
| atleast *Expression*
| atmost *Expression*

Type ::= *Identifier*
| *Collection* of *Type*

Definition ::= *Identifier* as *Expression*

Function ::= *Identifier* be *Expression*
| *Identifier* (*Parameter*) be *Expression*

Parameter ::= Λ
| *Identifier* : *Type*
| *Parameter* , *Parameter*

Table 4.1: Abstract Syntax of Object Comprehensions (continued).

Literal	::=	*String*
	|	*Integer*
	|	*Collection* { *Element* }
Element	::=	Λ
	|	*Expression*
	|	*Element* , *Element*
	|	*Expression* .. *Expression*
Path	::=	*Identifier*
	|	*Identifier* . *Method*
Method	::=	*Identifier*
	|	*Identifier* (*Argument*)
Argument	::=	Λ
	|	*Expression*
	|	*Argument* , *Argument*
Call	::=	*Identifier* (*Argument*)

Table 4.1: Abstract Syntax of Object Comprehensions (continued).

4.3 Object Comprehensions

The following subsections demonstrate object comprehensions using queries on the example database described in Section 2.13. Methods used in the examples are supposed to be without side-effects. Side-effecting methods can be dealt with as proposed in [GOPT92]. The focus of each query is underlined. A discussion is given after each query. Queries involving staff members, students, and tutors should be read as staff members, students, and tutors of the Science Faculty, unless stated otherwise.

4.3.1 Support of Object-Orientation

Method Calling & Dynamic Binding

Q1. Return staff members earning more than £1000 a month.

$$Set[\, s \leftarrow \underline{StaffMembers};\ s.\underline{salary} > 1000 \mid s \,]$$

Encapsulation protects attributes of an object from being accessed directly. Such an access must be made via a method. In Q1, *s.salary* represents the calling of method *salary* on a staff member object *s* drawn from the collection *StaffMembers*. Recall that a

tutor is a staff member whose salary is calculated differently using an overloaded method. Since *StaffMembers* may contain tutor objects, the method to be used will be dynamically determined depending on the type of *s*.

Complex Objects

Q2. Return tutors living in Glasgow.

$$Set[\, t \leftarrow Tutors;\ \underline{t.address.city} = \text{“}Glasgow\text{”} \mid t\,]$$

Support of complex objects implies that a method call may return an object. The returned object can, in turn, receive another method call. This can go on for several method calls until, for instance, a base value is returned. In Q2, *t.address.city* represents the calling of method *city* on the result returned by calling *address* on a tutor object *t*.

Object Identifiers

Q3. Return tutors working and studying in the same department.

$$Set[\, t \leftarrow Tutors;\ t.department \equiv t.major \mid t\,]$$

In the object-oriented paradigm, objects are represented by object identifiers which are essential for object sharing and representing cyclic relationships. Equality between objects is defined by the equality between their identifiers. In Q3, the equality operator, "=", compares two department objects using their object identifiers.

Class Hierarchy

Q4. Return all visiting staff members in the university.

$$Set[\, p \leftarrow Persons;\ p\ \underline{hasClass}\ VisitingStaff \mid p\,]$$

There is no collection in the database containing only objects of class *VisitingStaff*. *StaffMembers* contains only members in the Science Faculty. The only collection that contains all visiting staff members is *Persons*. It is the reason why the *Persons* collection is used in this query. Since a collection can contain heterogeneous elements belonging to different classes, elements of *Persons* can be of class *Person* or its subclasses. One way of selecting elements from such a collection is to specify the class of interest. In Q4, *hasClass* returns true if person object *p* is indeed of class *VisitingStaff*. This operation is essential for data models not supporting class extents.

Q5. Return all visiting staff members in the university who earn more than £1000 a month.

$$Set[\, p \leftarrow Persons;\ p\ \underline{hasClass}\ VisitingStaff\ \underline{with}\ p.salary > 1000 \mid p\,]$$

The method *salary* is defined for visiting staff members but not persons in general. Therefore calling *salary* on a person object may result in an error. To allow selection that is applicable only to objects of a particular class, the *hasClass & with* construct can be used. The role of *with* is similar to that of conjunction. The second condition (e.g. *p.salary > 1000*) is evaluated only if the first condition (e.g. *p hasClass VisitingStaff*) is true; however, the conditions around *with* cannot be swapped. In other words, *with* is a non-symmetric conjunction. This construct is essential for supporting static type checking in the absence of support for class extents.

Local Definitions

Q6. Return students whose major departments are in either Hillhead Street or University Avenue.

Set[*s* ← *Students*; *a as s.major.address.street*;
a = "*Hillhead Street*" *or a* = "*University Avenue*" | *s*]

Local definitions simplify queries by providing symbolic names to expressions. They are particularly useful when an expression is used in more than one place. In Q6, *s.major.address.street* would have been written twice if local definitions were not supported. The use of the symbolic name *a* for the expression saves repeating the long expression twice.

4.3.2 The Result Expression

Returning New Objects

Q7. Return students and the courses taken by them. The result is obtained by creating new objects using the student objects and the sets of courses.

Set[*s* ← *Students* | *AClass.new*(*s*, *s.takes*)]

So far, only queries returning existing objects have been examined. To return "new" information, the corresponding class has to be defined beforehand and the query will create objects of this class as the result. In Q7, the method *new*, which takes two parameters: *s* and *s.takes*, is called on the class *AClass*. If tuples were supported in the reference data model, the encoding of multiple results to a tuple would be obvious.

Nested Queries

Q8. Return students and the courses taken by them that have more than one credit. The result is obtained by creating new objects using the student objects and the sets of courses.

Set[*s* ← *Students* | *AClass.new*(*s*, *Set*[*c* ← *s.takes*; *c.credits* > 1 | *c*])]

Nested queries enable richer data structures to be returned as well as complex selection conditions to be expressed. In Q8, the inner query returns a set of courses and is used as a parameter to the method call in the result expression of the outer query. The generator used in the inner query is referred to as dependent generator and is explained in the next subsection.

4.3.3 Generators

Multiple Generators

Q9. Return students studying in the same department as Steve Johnson.

$$Set[\underline{x \leftarrow Students};\ \underline{y \leftarrow Students};\ x.name = \text{“}Steve\ Johnson\text{”};\ x.major = y.major \mid y]$$

Multiple generators allow relationships that are not explicitly defined in the database schema to be "re-constructed". In Q9, x is ranged over *Students* and y is ranged over the same set but independently. The missing relationship is established using the major departments of x and y. Multiple generators are particularly useful in processing nested collections as shown in the next example.

Dependent Generators

Q10. Return courses taken by the students.

$$Set[\underline{s} \leftarrow Students;\ c \leftarrow \underline{s.takes} \mid c]$$

The result of a method call can be a collection containing many elements. To facilitate querying over the elements in such a "nested" collection, a dependent generator can be used. In Q10, c is ranged over the collection returned by calling *takes* on the current student object s (i.e. the element in *Students* that is currently bound to s). The range of c changes whenever s is given a new object.

Literal Generators

Q11. Return those courses among DB4, AI4, HCI4, OS4, and PL4 which have more than one credit.

$$Set[c \leftarrow Courses;\ x \leftarrow \underline{Set\{\text{“}DB4\text{”}, \text{“}AI4\text{”}, \text{“}HCI4\text{”}, \text{“}OS4\text{”}, \text{“}PL4\text{”}\}};\ c.code = x;\ c.credits > 1 \mid c]$$

Collection literals can simplify queries by making them more concise and arguably clearer. In Q11, a set literal of strings is specified by listing the elements within curly brackets. They are, however, more often used in specifying filters as in the next example.

4.3.4 Quantifiers

In order to provide a coherent notation for querying over different collection classes, object comprehensions rely on quantifiers to express many collection operations. The quantifiers introduced in this subsection concern the occurrences of collection elements and they have the same semantics for sets, bags, and lists. Note that the semantics of a quantified expression is not compositional.

Existential Quantifiers

Q12. Return those courses among DB4, AI4, OS4, and PL4 which have more than one credit.

$$Set[\, c \leftarrow Courses;\ c.code = \underline{some}\ Set\{\text{“}DB4\text{”}, \text{“}AI4\text{”}, \text{“}OS4\text{”}, \text{“}PL4\text{”}\};\ c.credits > 1 \mid c\,]$$

Q13. Return students taking a course given by Steve Johnson.

$$Set[\, l \leftarrow StaffMembers;\ l.name = \text{“}Steve\ Johnson\text{”};\ s \leftarrow Students;\ \underline{some}\ s.takes = \underline{some}\ l.teaches \mid s\,]$$

A restricted form of existential quantification is provided by *some*, which can appear on either side of an operator. In Q12, the first filter succeeds if a course code is one of the members listed in the set literal. That is,

$$\exists x \bullet x \in Set\{\text{“}DB4\text{”}, \text{“}AI4\text{”}, \text{“}OS4\text{”}, \text{“}PL4\text{”}\} \wedge x = c.code$$

In Q13, the filter returns true if there is a common element between the two sets: *s.takes* and *l.teaches* (i.e. an non-empty intersection). That is

$$\exists x \exists y \bullet x \in s.takes \wedge y \in l.teaches \wedge x = y$$

Universal Quantifiers

Q14. Return students taking only courses given by Steve Johnson.

$$Set[\, l \leftarrow StaffMembers;\ l.name = \text{“}Steve\ Johnson\text{”};\ s \leftarrow Students;\ \underline{every}\ s.takes = some\ l.teaches \mid s\,]$$

In Q14, the last filter succeeds if all the course elements in *s.takes* are also in the set *l.teaches*. That is,

$$\forall x \exists y \bullet x \in s.takes \wedge y \in l.teaches \wedge x = y$$

Actually it is the subset relation. Note that the universal quantifier is always bound first if used together with an existential quantifier.

Numerical Quantifiers

Numerical quantifiers are based on numerical quantifiers used in logic [BB89]. They are very useful in dealing with duplicate elements in collections and the number of elements that are common between two collections (i.e. the size of the intersection).

Q15. Return students taking two or more courses given by Steve Johnson.

$$Set[\, l \leftarrow StaffMembers;\ l.name = \text{“}Steve\ Johnson\text{”};\ s \leftarrow Students;\ some\ s.takes = \underline{atleast\ 2}\ l.teaches \mid s\,]$$

Q16. Return students taking exactly two courses given by Steve Johnson.

$$Set[\, l \leftarrow StaffMembers;\ l.name = \text{“}Steve\ Johnson\text{”};\ s \leftarrow Students;\ some\ s.takes = \underline{just\ 2}\ l.teaches \mid s\,]$$

Q17. Return students taking no more than two courses given by Steve Johnson.

$$Set[\, l \leftarrow StaffMembers;\ l.name = \text{“}Steve\ Johnson\text{”};\ s \leftarrow Students;\ some\ s.takes = \underline{atmost\ 2}\ l.teaches \mid s\,]$$

In Q15, the last filter becomes true if there are at least two elements that are common between *s.takes* and *l.teaches*. That is

$$\exists_{\geqslant 2}\ x\, \exists\, y \bullet x \in l.teaches\ \wedge\ y \in s.takes\ \wedge\ x = y$$

where

$$\exists_{\geqslant n}\ x\quad P(x)\ \equiv\ \neg_{\geqslant n-1}\ x \bullet (P(x) \wedge (\exists\ \bullet P(y) \wedge y\quad))$$
$$\exists_{\geqslant 1}\ x \bullet P(x)\ \equiv\ \exists x \bullet P(x).$$

In Q16, the last filter succeeds if there are exactly two elements that are common between the operand sets. That is,

$$\exists_{=n}\ x \bullet P(x) \equiv (\exists_{\geqslant n}\ x \bullet P(x)) \wedge \neg(\exists_{\geqslant n+1}\ x \bullet P(x))$$

While in Q17, the number of common elements must be less than or equal to two. That is,

$$\exists_{\leqslant n}\ x \bullet P(x) \equiv \neg(\exists_{\geqslant n+1}\ x \bullet P(x))$$

Quantifiers are bound in the following order: the universal quantifier then numerical quantifiers and followed by the existential quantifier.

4.3.5 Support of Collections

Aggregate Functions

Q18. Return courses with less than two assessments.

$$Set[\, c \leftarrow Courses;\ (\underline{size}\ c.assessments) < 2 \mid c\,]$$

The aggregate function *size* returns the number of elements in a collection. It is defined for all collection classes. For bags and lists duplicate elements are included in the counting.

Equality

Q19. Return courses requiring no prerequisite courses.

$$Set[\, c \leftarrow Courses;\ c.prerequisites \underline{==} Set\{\} \mid c\,]$$

In many occasions it is necessary to compare two collections based on the elements, their occurrences, and their order. Two bags are equal if for each element drawn from either collection there is equal number of occurrences in both bags. For lists, the number of occurrences and the positions must be the same. In Q19, the filter becomes true if *c.prerequisites* is an empty set. Note that object comprehensions do not support equality on objects that are not collections.

Occurrences & Counting

Q20. Return courses with 4 assessments of the same percentage weight.

$$Set[\, c \leftarrow Courses;\ i \leftarrow c.assessments;\ just\ \underline{4}\ c.assessments = i \mid c\,]$$

Q21. Return the number of assessments worth 25% in the DB4 course.

$$Set[\, c \leftarrow Courses;\ c.code = \text{“}DB4\text{”};\ i \leftarrow List\{\, 1..(size\ c.assessments)\,\};\ just\ \underline{i}\ c.assessments = 25 \mid i\,]$$

In Q20, the selection is based on the number of occurrences (i.e. *4*) of an element (i.e. *i*) in the collection *c.assessments*. The use of numerical quantifiers simplifies retrieval based on occurrences. In Q21, the number of occurrences (i.e. *i*) of a given element (i.e. *25*) in the collection *c.assessments* is returned. The possible number of occurrences are generated using a literal generator ranging from 1 to (*size c.assessments*).

Positioning & Ordering

Q22. Return the first and second supervisors of Steve Johnson.

$$Set[\, s \leftarrow Students;\ s.name = \text{"}Steve\ Johnson\text{"};\ \underline{i \leftarrow List\{\ 1..2\ \}}\ |\ s.supervisedBy.[\underline{i}]\,]$$

Q23. Return students having Steve Johnson before Bob Campbell in their supervisor lists.

$$Set[\, s \leftarrow Students;\ \underline{sup\ as\ List\{\ 1..(size\ s.supervisedBy)\}};\ \underline{i \leftarrow sup};\ s.supervisedBy.[i].name = \text{"}Steve\ Johnson\text{"};\ \underline{j \leftarrow sup};\ s.supervisedBy.[j].name = \text{"}Bob\ Campbell\text{"};\ \underline{i < j}\ |\ s\,]$$

A list allows duplicate elements and keeps track of the order of the elements. Naturally queries involving lists may question on the order or positions of elements: In Q22, the first two elements of the list are returned using a literal generator. In Q23, a list literal is generated. Two generators are then ranged over it to match the given names. The relative order is determined using the range variable i and j.

Union & Differ

Q24. Return students in the Computing Science and Electrical Engineering Departments.

$$Set[\, s \leftarrow Students;\ s.major.name = \text{"}Computing\ Science\text{"}\ |\ s\,]$$
$$\underline{union}$$
$$Set[\, s \leftarrow Students;\ s.major.name = \text{"}Electrical\ Engineering\text{"}\ |\ s\,]$$

The *union* operator combines two collections to form a new collection of the same class but having all the elements. If the two operand collections have different element classes, the least general unique common superclass of the original element classes becomes the element class of the resultant collection. The union of two bags contains all the elements in the operand bags including all duplicates - additive union. The union of a list to another list appends the latter to the former - list concatenation.

Q25. Return cities where students, but no staff, live.

$$Set[\, s \leftarrow Students\ |\ s.address.city\,]$$
$$\underline{differ}$$
$$Set[\, s \leftarrow StaffMembers\ |\ s.address.city\,]$$

The difference between two collections can be expressed using *differ* as in Q25. The class of the resultant elements is determined in the same way as in *union*. The number of occurrences for an element in the resultant collection is the difference of those in the

operand collections. In the case of lists, *differ* removes elements in the second operand list from the first operand list. To be precise, given an element of the second operand list the last occurrence of it in the first operand list will be removed.

Converting Collections

Q26. Return the wages of tutors and keep the possible duplicate values.

$$\underline{Bag}[\, t \leftarrow \mathit{Tutors} \mid t.\mathit{wages}\,]$$

This query is based on a set of tutor objects and therefore the result is naturally a set of integers containing no duplicate values. If duplicates are to be kept the result can be specified to be a *bag*. Explicitly specifying the resultant collection kind provides a high-level mechanism to manage duplicates. Otherwise, implicit conversion rules have to be imposed or explicit conversion of all generators to the resultant collection kind will be required. Converting a collection into a set results in the elimination of duplicates and the loss of the order between elements. Converting a collection into a bag keeps the number of elements unchanged - duplicates are not lost and no new elements are introduced - but the order between the elements is lost. Converting a collection into a list keeps the number of elements and an arbitrary order is assigned to the elements.

Mixing Collections

Q27. Return courses taught by the supervisors of Steve Johnson.

$$\begin{array}{l} \mathit{Set}[\, s \leftarrow \mathit{Students};\ s.\mathit{name} = \text{“}\mathit{Steve\ Johnson}\text{”};\\ \quad \underline{\mathit{sup} \leftarrow s.\mathit{supervisedBy}};\ c \leftarrow \mathit{sup.teaches} \mid c\,] \end{array}$$

If an object-oriented data model supports more than one collection kind, the corresponding query notation should support not only different collection kinds but also the mixing of them in the same query. In Q27, the first generator is drawn from the set *Students*, the second generator from the list *s.supervisedBy*, and the last generator from the set *sup.teaches*. Knowing the resultant collection kind, object comprehensions can automatically convert all generators into the resultant kind.

4.3.6 Query Functions & Recursion

Q28. Return all direct and indirect prerequisite courses for the “DB4” course.

$$\begin{array}{l} \mathit{let}\ \underline{f}(\mathit{cs} : \mathit{Set\ of\ Course})\ \mathit{be}\\ \quad \mathit{cs\ union}\ \mathit{Set}[\, x \leftarrow \mathit{cs};\ y \leftarrow \underline{f}(x.\mathit{prerequisites}) \mid y\,]\\ \mathit{in}\ \mathit{Set}[\, c \leftarrow \mathit{Courses};\ c.\mathit{code} = \text{“}\mathit{DB4}\text{”};\ p \leftarrow \underline{f}(c.\mathit{prerequisites}) \mid p\,] \end{array}$$

In object-oriented data models, it is possible to find cyclic relationships involving one or more classes. This suggests that recursive queries should be supported. With object

comprehensions, recursive queries can be expressed using query functions. In Q28, the result of the query is generated by retrieving elements, p, from a collection returned by a recursive function, $f(c.prerequisites)$. Function f takes a set of courses and returns a set of courses. For each element x drawn from the input collection cs, f is applied recursively on the prerequisite courses of x, $x.prerequisites$, and the result is then combined with the input collection. The recursion stops when f is passed an empty set.

4.4 Semantic Optimisation

Optimisations developed for list comprehensions, which includes all well-known relational optimisation transformations, are also applicable to object comprehensions. Details of which are not repeated here and can be found in [TW89]. This section studies the optimisation of the new features using semantic information. Semantic optimisation rules are meaning-preserving transformation rules. They are used to create equivalent expressions based upon pattern matching and textual substitution. In addition, they also use semantics of the database schema as given by the class definitions and the class hierarchy. The overall goal of expression transformation is to reduce the cost of query evaluation. The focus of this section is on rule specification as opposed to rule application for query optimisation as the latter has been shown to be viable [GD87, HFLP89].

The following tables show the transformation rules for the new features in object comprehensions. The expressions that can be optimised are listed in the second column. The third column provides the equivalent expressions. The conditions under which a transformation can be performed are given in the last column. Definitions of the functions, relations, and domains used in the last column can be found in Chapter 2.

4.4.1 Class Hierarchy

	Optimisable Expression	Transformed Expression	Conditions
1	x hasClass Entity	true	
2	x hasClass c	true	$typeOf(x) \prec\!\!\!\cdot\ c$
3.1	(x hasClass c_1) and (x hasClass c_2)	x hasClass c_2	$c_2 \prec\!\!\!\cdot\ c_1$
3.2		x hasClass c	$\exists_1\ c : ClassName \bullet c \prec\!\!\!\cdot\ c_1 \wedge c \prec\!\!\!\cdot\ c_2$
3.3		false	$\neg\ \exists\ c : ClassName \bullet c \prec\!\!\!\cdot\ c_1 \wedge c \prec\!\!\!\cdot\ c_2$
4.1	not (x hasClass c_1) and (x hasClass c_2)	false	$c_2 \prec\!\!\!\cdot\ c_1$
4.2		x hasClass c_2	$\neg\ \exists\ c : ClassName \bullet c \prec\!\!\!\cdot\ c_1 \wedge c \prec\!\!\!\cdot\ c_2$
5	(x hasClass c_1) or (x hasClass c_2)	x hasClass c_1	$c_2 \prec\!\!\!\cdot\ c_1$

Table 4.2: Optimising Class Testing.

All these transformations can be applied at compile-time. This set of semantic optimisation rules subsumes those found in [Str90]. An interpretation of the rules is given below

and followed by an example of the usage of one of the rules.

Rule 1 Every class is related directly or indirectly to the root class by the ISA relationship. *Entity* is the root class. Therefore every object is also of class *Entity.*

Rule 2 If the class of x is a subclass of c, then x is also of class c.

Rule 3.1 If c_2 is a subclass of c_1, being of class c_2 implies being of c_1. It is therefore sufficient to check just c_2.

Rule 3.2 Let c be the unique common subclass of c_1 and c_2. For any x to be of both classes c_1 and c_2, it is possible only if x is of class c.

Rule 3.3 If c_1 and c_2 do not have a common subclass, no object can be of classes c_1 and c_2 at the same time.

Rule 4.1 It is not possible for an object to be of one class but not a superclass of this class.

Rule 4.2 Given that c_1 and c_2 do not have a common subclass. To check if an object is of class c_2 but not c_1, testing against c_2 will be sufficient as no object can be of the two classes simultaneously.

Rule 5 For an object to be of one class or a superclass of this class, it is sufficient just to check against the superclass.

Q29. Return people in all faculties that are a staff member and a student at the same time.

$$Set[p \leftarrow Persons;\ p\ hasClass\ Staff;\ p\ hasClass\ Student \mid p] \Longrightarrow Set[p \leftarrow Persons;\ p\ hasClass\ Tutor \mid p]$$

Using the database schema given in Section 2.13 and rule 3.2 in Table 4.2 above, it can be deduced that the two filters are true only if a person is a tutor. Applying the transformation will turn the original query from having two filters to just one. Note that ";" is semantically equivalent to *and*.

4.4.2 Quantifiers

In the following table, n stands for an integer expression, x represents an expression returning an object, and e represents any expression including quantified expression. Operators are represented by θ. The label *quantifier* stands for one of the three numerical quantifiers: *atmost*, *just*, or *atleast.*

These rules will be used mostly at run-time. They can be applied at compile-time if some of the operands are literal. An interpretation of the rules is given below.

	Optimisable Expression	Transformed Expression	Conditions
7	every cs θ x	true	*size* cs = 0
8	every cs = x	false	*kindOf*(cs) = *ASet* $\wedge$ (*size* cs) > 1
9	every cs_1 = every cs_2	false	(*kindOf*(cs_1) = *ASet* $\wedge$ (*size* cs_1) > 1) $\vee$ (*kindOf*(cs_2) = *ASet* $\wedge$ (*size* cs_2) > 1)
10	*quantifier* n cs θ e	false	n < 0
11.1	atmost n cs θ e	true	(*size* cs) = 0
11.2		true	(*size* cs) $\leqslant$ n
12	atmost n_1 cs_1 = some cs_2	true	*kindOf*(cs_1) = *ASet* $\wedge$ *kindOf*(cs_2) = *ASet* $\wedge$ (*size* cs_2) < n_1
13	just n cs = x	false	*kindOf*(cs) = *ASet* $\wedge$ n > 1
14.1	just n cs θ e	true	(*size* cs) = 0 $\wedge$ n = 0
14.2		false	(*size* cs) < n
15	just n_1 cs_1 = some cs_2	false	*kindOf*(cs_1) = *ASet* $\wedge$ *kindOf*(cs_2) = *ASet* $\wedge$ (*size* cs_2) < n_1
16	atleast n cs = x	false	*kindOf*(cs) = *ASet* $\wedge$ n > 1
17.1	atleast n cs θ e	true	n = 0
17.2		false	(*size* cs) < n
18	atleast n_1 cs_1 = some cs_2	false	*kindOf*(cs_1) = *ASet* $\wedge$ *kindOf*(cs_2) = *ASet* $\wedge$ (*size* cs_2) < n_1
19	some cs θ e	false	(*size* cs) = 0

Table 4.3: Optimising Quantified Expressions.

Rule 7 Universal quantification over an empty collection is always true.

Rule 8 It is not possible for the elements of a non-singleton set to be identical to the same object as a set does not allow duplicates.

Rule 9 It is not possible for the elements of a non-singleton set to be identical with all the elements of another set.

Rule 10 A numerical quantifier only works with positive numbers.

Rule 11.1 An empty collection will satisfy any limit of occurrence.

Rule 11.2 If the limit of occurrence is larger than the size of the collection, it is always true regardless of the operator.

Rule 12 If the limit of occurrence for a set is larger than the size of the set from which the comparing elements are drawn, it is always true with the equality operator.

Rule 13 It is not possible for an element to occur more than once in a set.

Rule 14.1 It is always true that no element occurs in an empty collection.

Rule 14.2 It is not possible for the number of occurrences of an element to be greater than the size of the collection from which it is drawn.

Rule 15 If the number of occurrences for a set is larger than the size of the set from which the comparing elements are drawn, it is always false with the equality operator.

Rule 16 It is not possible for an element to occur more than once in a set.

Rule 17.1 Any object occurs at least zero times in a collection.

Rule 17.2 It is not possible for the minimum number of occurrences of any element to be larger than the size of the collection.

Rule 18 If the minimum number of occurrences in a set is larger than the size of the set from which the comparing elements are drawn, then it is always false with the equality operator.

Rule 19 Existential quantification over an empty collection is always false.

4.5 Summary

The salient features of object comprehensions have been presented. The example queries demonstrated that sophisticated queries can be expressed using object comprehensions in a clear and concise fashion. This is achieved via the support of a number of powerful predicates, orthogonal composition of constructs, query functions, local definitions, manipulation of different collection classes, and recursion. Despite of being a powerful notation, object comprehensions can be optimised using existing optimisation techniques. Some optimisations for class testing and quantification were identified and presented. In the next chapter, the expressive power of object comprehensions is further demonstrated by providing translation from other query languages to object comprehensions.

Chapter 5

Translating Query Languages to Object Comprehensions

Many query languages have been proposed for object-oriented databases. These query languages vary in expressive power and use different notations. Despite of all these apparent dissimilarities, they share similar underlying semantics. This observation suggests that a single unified scheme can be developed to support these languages. This chapter describes the use of object comprehensions to provide such multi-lingual support for the reference data model. A set of translation schemes from the query languages studied in Chapter 3 to object comprehensions is presented. These translation schemes demonstrate that object comprehensions are as powerful as any of these query languages with respect to the reference data model and therefore can be used to provide a platform to support any or all of these query languages.

The organisation of this chapter is as follows. Section 5.1 introduces the queries used to demonstrate the translation schemes. Section 5.2 explains the notations used in the translation schemes. Section 5.3 and 5.4 present the translation schemes for ONTOS SQL and ORION. An example is given in each section to demonstrate the translation. Section 5.5 and 5.6 provide an example translation for OSQL and O_2SQL (their translation schemes are given in Appendix C). Section 5.7 concludes.

5.1 Example Queries

Multi-lingual support allows a database to be queried using more than one query language. Different users can therefore query the same database using query languages that are familiar to them or more appropriate for the particular environments in which they operate. Multi-lingual query support for conventional databases was investigated in MLDS [DH87]. This chapter discusses multi-lingual support for object-oriented databases using object comprehensions.

Four example queries are used to demonstrate the translations from existing query languages to object comprehensions. They are chosen to demonstrate the interesting features of these query languages. The queries are expressed as object comprehensions as follows.

Q30. Return courses co-run by the Computing Science department and having between 1 and 3 credits.

```
List[ c ← Courses; d ← Departments; d.name = "Computing Science";
      d = some c.runBy; 1 <= c.credits; c.credits <= 3 | c ]
```

Q31. Return students taking some course given by their supervisors.

```
Bag[ s ← Students; c ← Courses; c = some s.takes;
     c = some List[ x ← s.supervisedBy; y ← x.teaches | y ] | s ]
```

Q32. Return students having no supervisors from their major departments.

```
Set[ s ← Students;
    (let f( xs : List of Staff ) be
        Set[ x ← xs | x.department ~= s.major ]
    in every f( s.supervisedBy ) = true) | s ]
```

Q33. Return courses requiring "Logic1" as a direct or indirect prerequisite.

```
Set[ c ← Courses;
    some( let codes( cs : Set of Course ) be
            Set[ c ← cs | c.code ]
          in
    codes( let preqs( cs : Set of Course ) be
             cs union Set[ c ← cs; x ← preqs( c.prerequisites ) | x ]
           in preqs( c.prerequisites ) ) ) = "Logic1" | c ]
```

This chapter focuses on the retrieval capability of query languages and hence data management functions, like update and delete, are not discussed. As a consequence, constructs providing such functions are not included in the translation schemes. Since this research is conducted in the context of the reference data model, modelling notions and their corresponding constructs not supported by the model are naturally excluded from the translation. Discussion of some of the constructs that are left out can be found in Appendix B.

5.2 Translation Notation

The following sections describe the translation of four high-level query languages, namely ONTOS SQL, ORION, OSQL, and O_2SQL into object comprehensions. The transla-

tion schemes are presented in denotational style [Sto77]: using compositional translation functions and the argument to each translation function is put within ⟦ ⟧ brackets.

5.2.1 Translation Functions

Each translation is described in terms of a number of translation functions each dealing with one syntactic category.

TQ Translate a Query
TD Translate a Domain (Generator)
TE Translate an Expression
TO Translate an Operation

For the sake of simplicity, it is assumed that a number of translation functions exist to deal with types.

TT Extract the Type of an Expression
TM Extract the Element Type of Collection
TC Extract the Kind of a Collection
TX Extract the Collection Kind from a Type Expression

The translation functions of each query language are subscribed by *ontos*, *orion*, *osql*, and o_2sql.

5.2.2 Syntactic Categories

Within each language, different syntactic categories are given different symbols:

Q Query
D Domain (Generator)
E Expression
I Identifier
k Constant
S Sort Order

ϕ Operation
ω Relational Operation (Boolean)
ψ Arithmetic Operation
ξ Collection Kind

The four query languages are marginally simplified to make the translation more elegant. The simplifications made are listed and discussed in Appendix B.

5.3 Translating ONTOS SQL

5.3.1 ONTOS SQL Abstract Syntax

$$
\begin{array}{lll}
Q & ::= & \textit{select } E \textit{ from } Ds \textit{ where } E \\
 & | & \textit{select } E \textit{ from } Ds \\
 & & \\
Ds & ::= & D \mid D,\ Ds \\
 & & \\
D & ::= & E\ I \mid \{\ Es\ \}\ I \\
 & & \\
Es & ::= & \Lambda \mid E \mid E,\ Es \\
 & & \\
E & ::= & E \textit{ and } E \mid E \textit{ or } E \mid \textit{not } E \\
 & & \\
 & | & E \textit{ is in } E \mid E \textit{ is not in } E \\
 & | & E \textit{ between } E \textit{ and } E \mid E \textit{ not between } E \textit{ and } E \\
 & | & E\ \omega\ E \mid E\ \psi\ E \\
 & & \\
 & | & E.E \mid I(\ Es\ ) \mid (\ E\ ) \mid I \mid k \\
 & & \\
\omega & ::= & = \mid <> \mid > \mid >= \mid < \mid <= \\
 & & \\
\psi & ::= & * \mid / \mid + \mid -
\end{array}
$$

Table 5.1: ONTOS SQL Abstract Syntax.

5.3.2 ONTOS SQL Translation Rules

Using ONTOS SQL, query Q30 can be expressed as follows.

Q30. Return courses co-run by the Computing Science department and having between 1 and 3 credits.

```
select   c
from     Courses c, Departments d
where    d.name = "Computing Science"
and      d is in c.runBy
and      c.credits between 1 and 3
```

Translating Queries

The ONTOS SQL translation uses the $\mathbf{TQ}_{\text{ontos}}$ function initially. Since an ONTOS SQL query always returns a list as the result, $\mathbf{TQ}_{\text{ontos}}$ produces a list comprehension. In the

list comprehension, all of the domains are translated into generators by $\mathbf{TD}_{ontos}$ [[D]]. Then the filters, if any, are translated into qualifiers by $\mathbf{TE}_{ontos}$ [[E_1]]. Finally the target is translated by $\mathbf{TE}_{ontos}$ [[E]].

$$\mathbf{TQ}_{ontos} \text{ [[select E from D where } E_1 \text{]]} \Rightarrow \mathit{List}[\, \mathbf{TD}_{ontos} \text{ [[D]] ; } \mathbf{TE}_{ontos} \text{ [[} E_1 \text{]] | } \mathbf{TE}_{ontos} \text{ [[E]]]} \quad \text{(ontos.1)}$$

$$\mathbf{TQ}_{ontos} \text{ [[select E from D]]} \Rightarrow \mathit{List}[\, \mathbf{TD}_{ontos} \text{ [[D]] | } \mathbf{TE}_{ontos} \text{ [[E]]]} \quad \text{(ontos.2)}$$

Translating Domains

Two domains are translated individually using $\mathbf{TD}_{ontos}$ and the resulting translations are composed with a semi-colon. A domain can be formed by either a collection-valued expression or a collection literal. In the case of a collection-valued expression, the function $\mathbf{TE}_{ontos}$ is used. For a literal, the elements are translated using $\mathbf{TE}_{ontos}$ and the results form the elements of the literal $\mathit{Set}\{\mathbf{TE}_{ontos}$ [[E]]$\}$. Domain variables remain unchanged and are composed to the new domains using $\leftarrow$.

$$\mathbf{TD}_{ontos} \text{ [[} D_1 \text{ , } D_2 \text{]]} \Rightarrow \mathbf{TD}_{ontos} \text{ [[} D_1 \text{]] ; } \mathbf{TD}_{ontos} \text{ [[} D_2 \text{]]} \quad \text{(ontos.3)}$$

$$\mathbf{TD}_{ontos} \text{ [[E I]]} \Rightarrow I \leftarrow \mathbf{TE}_{ontos} \text{ [[E]]} \quad \text{(ontos.4)}$$

$$\mathbf{TD}_{ontos} \text{ [[\{E\} I]]} \Rightarrow I \leftarrow \mathit{Set}\{\, \mathbf{TE}_{ontos} \text{ [[E]] }\} \quad \text{(ontos.5)}$$

Translating Expressions

A sequence of expressions separated by comma is translated as follow

$$\mathbf{TE}_{ontos} \text{ [[} E_1 \text{ , } E_2 \text{]]} \Rightarrow \mathbf{TE}_{ontos} \text{ [[} E_1 \text{]] , } \mathbf{TE}_{ontos} \text{ [[} E_2 \text{]]} \quad \text{(ontos.6)}$$

Selection conditions can be composed using logical connectives. Both ONTOS SQL and object comprehensions support the same set of connectives except that semi-colon can be used for *and* in object comprehensions. The operand expressions are translated using $\mathbf{TE}_{ontos}$.

$$\mathbf{TE}_{ontos} \text{ [[} E_1 \text{ and } E_2 \text{]]} \Rightarrow \mathbf{TE}_{ontos} \text{ [[} E_1 \text{]] ; } \mathbf{TE}_{ontos} \text{ [[} E_2 \text{]]} \quad \text{(ontos.7)}$$

$$\mathbf{TE}_{ontos} \text{ [[} E_1 \text{ or } E_2 \text{]]} \Rightarrow \mathbf{TE}_{ontos} \text{ [[} E_1 \text{]] } \mathit{or}\ \mathbf{TE}_{ontos} \text{ [[} E_2 \text{]]} \quad \text{(ontos.8)}$$

$$\mathbf{TE}_{ontos} \text{ [[not E]]} \Rightarrow \mathit{not}\ \mathbf{TE}_{ontos} \text{ [[E]]} \quad \text{(ontos.9)}$$

ONTOS SQL provides two membership test operations and they can be represented in object comprehensions using the existential quantifier, *some*. A value is in a collection

if it is equal (=) to some element in the collection. The converse holds when no element (~=) equals that value. The operand expressions are translated using $\mathbf{TE}_{ontos}$.

$$\mathbf{TE}_{ontos}\ [\![\ E_1 \text{ is in } E_2\ ]\!] \Rightarrow \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!] = \textit{some}\ \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] \quad \text{(ontos.10)}$$

$$\mathbf{TE}_{ontos}\ [\![\ E_1 \text{ is not in } E_2\ ]\!] \Rightarrow \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!] \sim= \textit{some}\ \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] \quad \text{(ontos.11)}$$

A value can be tested to see if it falls within or outside a given range using *between* in ONTOS SQL. In object comprehensions, range testing can be done by comparing the given value with the largest and smallest values in the given range.

$$\mathbf{TE}_{ontos}\ [\![\ E \text{ between } E_1 \text{ and } E_2\ ]\!]$$
$$\Rightarrow \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!] <= \mathbf{TE}_{ontos}\ [\![\ E\ ]\!];\ \mathbf{TE}_{ontos}\ [\![\ E\ ]\!] <= \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] \quad \text{(ontos.12)}$$

$$\mathbf{TE}_{ontos}\ [\![\ E \text{ not between } E_1 \text{ and } E_2\ ]\!]$$
$$\Rightarrow \mathbf{TE}_{ontos}\ [\![\ E\ ]\!] < \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!]\ \textit{or}\ \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] < \mathbf{TE}_{ontos}\ [\![\ E\ ]\!] \quad \text{(ontos.13)}$$

Relational and arithmetic operations can be translated easily. For the rest of this chapter, the translation of common operations is captured using a generalised rule in which an operation is represented by ϕ. An example of translating ϕ is given below.

$$\mathbf{TE}_{ontos}\ [\![\ E_1\ \phi\ E_2\ ]\!] \Rightarrow \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!]\ \mathbf{TO}_{ontos}\ [\![\ \phi\ ]\!]\ \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] \quad \text{(ontos.14)}$$

$$\mathbf{TO}_{ontos}\ [\![\ <>\ ]\!] \Rightarrow \sim= \quad \text{(ontos.15)}$$

Both ONTOS SQL and object comprehensions use the dot notation for method calls whose arguments are put within () bracket

$$\mathbf{TE}_{ontos}\ [\![\ E_1\ .\ E_2\ ]\!] \Rightarrow \mathbf{TE}_{ontos}\ [\![\ E_1\ ]\!]\ .\ \mathbf{TE}_{ontos}\ [\![\ E_2\ ]\!] \quad \text{(ontos.16)}$$

$$\mathbf{TE}_{ontos}\ [\![\ I\ (\ E\ )\ ]\!] \Rightarrow I\ (\ \mathbf{TE}_{ontos}\ [\![\ E\ ]\!]\ ) \quad \text{(ontos.17)}$$

Brackets, identifiers, and constants are translated as follows.

$$\mathbf{TE}_{ontos}\ [\![\ (\ E\ )\ ]\!] \Rightarrow (\ \mathbf{TE}_{ontos}\ [\![\ E\ ]\!]\ ) \quad \text{(ontos.18)}$$

$$\mathbf{TE}_{ontos}\ [\![\ E\ ]\!] \Rightarrow E \quad \text{(ontos.19)}$$

5.3.3 Example ONTOS SQL Translation

As an example of the use of the translation rules, Query Q30 presented at the beginning of this section can be translated as follows. The numbers of the translation rules applied are listed after ⇒.

$\mathbf{TQ}_{\text{ontos}}$ ⟦ select c
from Courses c, Departments d
where d.name = "Computing Science"
and d is in c.runBy
and c.credits between 1 and 3 ⟧

⇒ (ontos.1), (ontos.7) twice
List[$\mathbf{TD}_{\text{ontos}}$ ⟦ Courses c, Departments d ⟧;
$\mathbf{TE}_{\text{ontos}}$ ⟦ d.name = "Computing Science" ⟧;
$\mathbf{TE}_{\text{ontos}}$ ⟦ d is in c.runBy ⟧; $\mathbf{TE}_{\text{ontos}}$ ⟦ c.credits between 1 and 3 ⟧ | $\mathbf{TE}_{\text{ontos}}$ ⟦ c ⟧]

⇒ (ontos.3), (ontos.4) twice, (ontos.19) twice
List[*c* ← *Courses*; *d* ← *Departments*;
$\mathbf{TE}_{\text{ontos}}$ ⟦ d.name = "Computing Science" ⟧;
$\mathbf{TE}_{\text{ontos}}$ ⟦ d is in c.runBy ⟧; $\mathbf{TE}_{\text{ontos}}$ ⟦ c.credits between 1 and 3 ⟧ | $\mathbf{TE}_{\text{ontos}}$ ⟦ c ⟧]

⇒ $\mathbf{TQ}_{\text{ontos}}$, (ontos.14), (ontos.16), (ontos.19) 3 times
List[*c* ← *Courses*; *d* ← *Departments*; *d.name* = "*Computing Science*";
$\mathbf{TE}_{\text{ontos}}$ ⟦ d is in c.runBy ⟧; $\mathbf{TE}_{\text{ontos}}$ ⟦ c.credits between 1 and 3 ⟧ | $\mathbf{TE}_{\text{ontos}}$ ⟦ c ⟧]

⇒ (ontos.10), (ontos.16), (ontos.19) 3 times
List[*c* ← *Courses*; *d* ← *Departments*; *d.name* = "*Computing Science*";
d = *some c.runBy*; $\mathbf{TE}_{\text{ontos}}$ ⟦ c.credits between 1 and 3 ⟧ | $\mathbf{TE}_{\text{ontos}}$ ⟦ c ⟧]

⇒ (ontos.12), (ontos.16) twice, (ontos.19) 6 times
List[*c* ← *Courses*; *d* ← *Departments*; *d.name* = "*Computing Science*";
d = *some c.runBy*; 1 <= *c.credits*; *c.credits* <= 3 | $\mathbf{TE}_{\text{ontos}}$ ⟦ c ⟧]

⇒ (ontos.19)
List[*c* ← *Courses*; *d* ← *Departments*; *d.name* = "*Computing Science*";
d = *some c.runBy*; 1 <= *c.credits*; *c.credits* <= 3 | *c*]

5.4 Translating ORION

5.4.1 ORION Abstract Syntax

$$
\begin{array}{lcl}
Q & ::= & Q \textit{ union } Q \mid Q \textit{ intersect } Q \mid Q \textit{ difference } Q \mid (\ Q\ ) \\
 & \mid & \textit{select } E \textit{ from } Ds \textit{ where } E \\
 & \mid & \textit{select } E \textit{ from } Ds \\
 & \mid & \textit{select } E \textit{ where } E \\
 & \mid & \textit{select } E \\
\\
Ds & ::= & D \mid D,\ Ds \\
\\
D & ::= & I : I \textit{ is\_in } E \mid : I \textit{ is } E \\
\\
Es & ::= & E \mid E,\ Es \\
\\
E & ::= & E \textit{ and } E \mid E \textit{ or } E \\
\\
 & \mid & E \textit{ class } I\ \omega\ E \mid E \textit{ class } I\ E\ \omega\ E \\
 & \mid & \textit{each } E\ \omega\ E \mid E \textit{ each } E\ \omega\ E \\
 & \mid & \textit{exists } E\ \omega\ E \mid E \textit{ exists } E\ \omega\ E \\
 & \mid & E\ \omega\ E \mid E\ \psi\ E \\
\\
 & \mid & E \textit{ set\_of } E \mid \textit{set\_of } E \mid E \textit{ (recurse } E\textit{)} \\
 & \mid & E\ E \mid I \mid I(\ Es\ ) \mid (\ E\ ) \\
 & \mid & :I \mid\ '(\ Es\ ) \mid k \\
\\
\omega & ::= & = \mid \neg := \mid > \mid >= \mid < \mid <= \mid \\
 & \mid & \textit{equal} \mid \textit{string\_equal} \mid \textit{string} = \\
 & \mid & \textit{has\_subset} \mid \textit{is\_subset} \mid \textit{is\_equal} \\
 & \mid & \textit{has\_element} \mid \neg : \textit{has\_element} \mid \textit{is\_in} \mid \neg : \textit{is\_in} \\
\\
\psi & ::= & * \mid / \mid + \mid -
\end{array}
$$

Table 5.2: ORION Abstract Syntax.

5.4.2 ORION Translation Rules

Using the ORION query language, query Q33 can be expressed as follows.

Q33. Return courses requiring "Logic1" as a direct or indirect prerequisite.

```
select   :c
from     :c is_in Courses
where    :c exists (recurse set_of prerequisites) code = "Logic1"
```

ORION allows the use of multi-valued methods, recursion, and quantifiers in path expressions. For example, the path expression *:c set-of prerequisites code* returns the course codes of all the prerequisite courses of *:c*. The keyword *set-of* indicates that a multi-valued method *prerequisites* is used in the path expression. This path expression can be enhanced to return the course codes of all direct and indirect prerequisite courses as follows: *:c (recurse set-of prerequisites) code*. The existential quantifier *exists* in Q33 tests the existence of a direct or indirect prerequisite course with the course code *Logic1*.

Translating Queries

The translation for ORION begins with the $\mathbf{TQ}_{\text{orion}}$ function. ORION supports a group of set operations that are used only at the top level of a query. The operation *intersect* is not directly supported in object comprehensions but can be expressed in terms of *differ*. All the operands are translated using $\mathbf{TQ}_{\text{orion}}$.

$$\mathbf{TQ}_{\text{orion}}[\![\ Q_1 \text{ union } Q_2\ ]\!] \Rightarrow \mathbf{TQ}_{\text{orion}}[\![\ Q_1\ ]\!]\ \textit{union}\ \mathbf{TQ}_{\text{orion}}[\![\ Q_2\ ]\!] \quad \text{(orion.1)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ Q_1 \text{ intersec } Q_2\ ]\!] \Rightarrow \mathbf{T}\,\mathbf{Q}_{\text{orion}}[\![\ Q_1\ ]\!]\ \textit{differ}\ (\ \mathbf{TQ}_{\text{orion}}[\![\ Q_1\ ]\!]\ \textit{differ}\ \mathbf{TQ}_{\text{orion}}[\![\ Q_2\ ]\!]\ ) \quad \text{(orion.2)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ Q_1 \text{ difference } Q_2\ ]\!] \Rightarrow \mathbf{TQ}_{\text{orion}}[\![\ Q_1\ ]\!]\ \textit{differ}\ \mathbf{TQ}_{\text{orion}}[\![\ Q_2\ ]\!] \quad \text{(orion.3)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ (\ Q\ )\ ]\!] \Rightarrow (\ \mathbf{TQ}_{\text{orion}}[\![\ Q\ ]\!]\ ) \quad \text{(orion.4)}$$

A query can be formulated in SQL-like syntax except that the domain and condition parts are optional. Set is the only collection kind supported in ORION, all queries are therefore translated into set comprehensions.

$$\mathbf{TQ}_{\text{orion}}[\![\ \text{select E from D where } E_1\ ]\!] \Rightarrow \textit{Set}[\,\mathbf{TD}_{\text{orion}}[\![\ D\ ]\!];\ \mathbf{TE}_{\text{orion}}[\![\ E_1\ ]\!]\ |\ \mathbf{TE}_{\text{orion}}[\![\ E\ ]\!]\,] \quad \text{(orion.5)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ \text{select E from D}\ ]\!] \Rightarrow \textit{Set}[\,\mathbf{TD}_{\text{orion}}[\![\ D\ ]\!]\ |\ \mathbf{TE}_{\text{orion}}[\![\ E\ ]\!]\,] \quad \text{(orion.6)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ \text{select E where } E_1\ ]\!] \Rightarrow \textit{Set}[\,\mathbf{TE}_{\text{orion}}[\![\ E_1\ ]\!]\ |\ \mathbf{TE}_{\text{orion}}[\![\ E\ ]\!]\,] \quad \text{(orion.7)}$$

$$\mathbf{TQ}_{\text{orion}}[\![\ \text{select E}\ ]\!] \Rightarrow \textit{Set}[\,|\ \mathbf{TE}_{\text{orion}}[\![\ E\ ]\!]\,] \quad \text{(orion.8)}$$

Translating Domains

Multiple domains are separated by comma in ORION but semi-colon in object comprehensions. Domain variables are prefixed with a colon which is removed during the translation.

$$\mathbf{TD}_{orion}\ [\![\ D_1\ ,\ D_2\ ]\!] \ \Rightarrow\ \mathbf{TD}_{orion}\ [\![\ D_1\ ]\!]\ ;\ \mathbf{TD}_{orion}\ [\![\ D_2\ ]\!] \qquad \text{(orion.9)}$$

$$\mathbf{TD}_{orion}\ [\![\ :I\ \ is\_in\ E\ ]\!] \ \Rightarrow\ I \leftarrow \mathbf{TE}_{orion}\ [\![\ E\ ]\!] \qquad \text{(orion.10)}$$

If an expression is used more than once in a query it is convenient to give it a name so that further references to the expression can be replaced by the name. In ORION, such a name can be specified in the domain part of a query and must start with a colon which is removed during the translation.

$$\mathbf{TD}_{orion}\ [\![\ :I\ is\ E\ ]\!] \ \Rightarrow\ I\ \textit{as}\ \mathbf{TE}_{orion}\ [\![\ E\ ]\!] \qquad \text{(orion.11)}$$

Translating Expressions

A sequence of expressions separated by comma is translated as follows.

$$\mathbf{TE}_{orion}\ [\![\ E_1\ ,\ E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{orion}\ [\![\ E_1\ ]\!]\ ,\ \mathbf{TE}_{orion}\ [\![\ E_2\ ]\!] \qquad \text{(orion.12)}$$

Logical connectives are translated as follows.

$$\mathbf{TE}_{orion}\ [\![\ E_1\ \text{and}\ E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{orion}\ [\![\ E_1\ ]\!]\ ;\ \mathbf{TE}_{orion}\ [\![\ E_2\ ]\!] \qquad \text{(orion.13)}$$

$$\mathbf{TE}_{orion}\ [\![\ E_1\ \text{or}\ E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{orion}\ [\![\ E_1\ ]\!]\ \textit{or}\ \mathbf{TE}_{orion}\ [\![\ E_2\ ]\!] \qquad \text{(orion.14)}$$

ORION supports class testing using *class* that can be sandwiched between method calls. This embedding of class testing within computation requires a two-staged evaluation in object comprehensions: a class testing on the first part of the expression is carried out first and then the whole expression is evaluated. The translation of a simpler case is given in Appendix C.

$$\mathbf{TE}_{orion}\ [\![\ E_1\ \text{class}\ I\ \ E_2\ \omega\ E_3\ ]\!]$$
$$\Rightarrow\ \mathbf{TE}_{orion}\ [\![\ E_1\ ]\!]\ \textit{hasClass}\ I\ ;\ \mathbf{TE}_{orion}\ [\![\ E_1\ \ E_2\ \omega\ E_3\ ]\!] \qquad \text{(orion.15)}$$

Existential and universal quantifiers are supported in ORION using *exists* and *each* respectively. A representative case is given below and other cases are considered in Appendix C. In the expression below, E_1 E_2 is a collection-valued expression and the existential quantifier is applied over the elements of it. The whole expression returns true if any element x in the collection satisfies the condition (x E_3 ω E_4). The translation turns the expression on the left hand side of ω into an expression involving implicit join whose

translation is given later in this subsection. The comparison operation ω and the expression E_4 on its right hand side are translated using $\mathbf{TO}_{\text{orion}}$ and $\mathbf{TE}_{\text{orion}}$. The application of this rule is demonstrated in the translation example given in the next subsection.

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{exists}\ E_2\ E_3\ \omega\ E_4\ ]\!]\\ &\quad\Rightarrow\ \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{set\_of}\ E_2\ E_3\ ]\!]\ \ \mathbf{TO}_{\text{orion}}\ [\![\ \omega\ ]\!]\ \ \mathbf{TE}_{\text{orion}}\ [\![\ E_4\ ]\!]\end{aligned}\qquad\text{(orion.16)}$$

Operations between two sets or a set and a value are translated as follows. Other operations use $\mathbf{TO}_{\text{orion}}$ and the translation rules are given in Appendix C.

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{has\_subset}\ E_2\ ]\!]\ \Rightarrow\ \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \textit{every}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\qquad\text{(orion.17)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{is\_subset}\ E_2\ ]\!]\ \Rightarrow\ \textit{every}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\qquad\text{(orion.18)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{has\_element}\ E_2\ ]\!]\ \Rightarrow\ \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\qquad\text{(orion.19)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \neg\text{:has\_element}\ E_2\ ]\!]\ \Rightarrow\ \textit{not}\ (\ \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\ )\qquad\text{(orion.20)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{is\_in}\ E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\qquad\text{(orion.21)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \neg\text{:is\_in}\ E_2\ ]\!]\ \Rightarrow\ \textit{not}\ (\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!] = \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\ )\qquad\text{(orion.22)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \phi\ E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ ]\!]\ \ \mathbf{TO}_{\text{orion}}\ [\![\ \phi\ ]\!]\ \ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\qquad\text{(orion.23)}$$

If a method is called upon a collection and the method is applicable to the elements in the collection, the resultant set will contain the results obtained by calling the method on the individual elements. The translation rules below show how such an implicit join can be translated. In the first rule, $E_1\ E_2$ is a collection-valued expression and E_3 represents a method call applicable to the elements of the collection. Note that the keyword *set_of* provides syntactic support to mark the location of the join. The translation generates a query function. The collection $E_1\ E_2$ is passed as an argument to the query function which applies the rest of the expression to each element in the collection. The translation function $\mathbf{TT}_{\text{orion}}$ returns the type of an expression and is used to fill in the type information in the signature of the query function. In the second rule, E_2 represents a collection and E_3 is a method call applicable to the elements of E_2. For other cases, *set_of* simply serves as an alternative form for method calls.

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ E_1\ \text{set\_of}\ E_2\ E_3\ ]\!]\ \Rightarrow\ (\textit{let}\ f(\ xs : \mathbf{TT}_{\text{orion}}\ [\![\ E_1\ E_2\ ]\!]\ )\ \textit{be}\\ &\qquad\qquad Set[\ x\ \leftarrow\ xs\ \mid\ \mathbf{TE}_{\text{orion}}\ [\![\ x\ E_3\ ]\!]\ ]\\ &\qquad\textit{in}\ f(\ \mathbf{TE}_{\text{orion}}\ [\![\ E_1\ E_2\ ]\!]\ ))\end{aligned}\qquad\text{(orion.24)}$$

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ \text{set\_of}\ E_2\ E_3\ ]\!]\ \Rightarrow\ (\textit{let}\ f(\ xs : \mathbf{TT}_{\text{orion}}\ [\![\ E_2\ ]\!]\ )\ \textit{be}\\ &\qquad\qquad Set[\ x\ \leftarrow\ xs\ \mid\ \mathbf{TE}_{\text{orion}}\ [\![\ x\ E_3\ ]\!]\ ]\\ &\qquad\textit{in}\ f(\ \mathbf{TE}_{\text{orion}}\ [\![\ E_2\ ]\!]\ ))\end{aligned}\qquad\text{(orion.25)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \text{set\_of}\ \ \mathrm{E}_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \mathrm{E}_2\ ]\!] \qquad \text{(orion.26)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ \text{set\_of}\ \ \mathrm{E}_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_2\ ]\!] \qquad \text{(orion.27)}$$

Recursive queries can be formulated in ORION using *recurse*. A recursive query can involve one or more method calls that can return either values or collections. The next rule deals with recursive queries involving no collection-valued method calls. In the expression, E_2 is repeatedly applied. Each time the result of the method is checked with the results collected so far. If it is already in the collection, the recursion will stop and the collection will be returned. If it is not in the collection, the results obtained so far will be updated and the method E_2 will be applied again.

$$\begin{aligned}
&\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ (\ \text{recurse}\ \ \mathrm{E}_2)\ ]\!] \\
&\quad\Rightarrow\ (\mathit{let}\ f(\ \mathit{xs} : \mathit{Set}\ \mathit{of}\ \mathbf{TT}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \mathrm{E}_2\ ]\!],\ y : \mathbf{TT}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \mathrm{E}_2\ ]\!]\ )\ \mathit{be} \\
&\qquad\quad \mathit{Set}[\ y\ =\ \mathit{some}\ \mathit{xs};\ \ x\ \leftarrow\ \mathit{xs}\ \mid\ x\ ] \\
&\qquad\quad \mathit{union} \\
&\qquad\quad \mathit{Set}[\ y\ \sim\!=\ \mathit{some}\ \mathit{xs};\ \ z\ \leftarrow\ f(\ (\mathit{Set}\{y\}\ \mathit{union}\ \mathit{xs}),\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{y}\ \mathrm{E}_2\ ]\!]\ )\ \mid\ z\ ] \\
&\qquad \mathit{in}\ \ f(\ \mathit{Set}\{\},\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \mathrm{E}_2\ ]\!]\ ))
\end{aligned} \qquad \text{(orion.28)}$$

Recursion involving collection-valued method calls can be translated using the next rule. In the expression, E_2 returns a collection instead of a single value. Each application of E_2 terminates when an empty set is returned. An example of its application can be found in the translation example.

$$\begin{aligned}
&\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ (\ \text{recurse}\ \ \text{set_of}\ \ \mathrm{E}_2)\ ]\!] \\
&\quad\Rightarrow\ (\mathit{let}\ f(\ \mathit{xs} : \mathbf{TT}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \mathrm{E}_2\ ]\!]\ )\ \mathit{be} \\
&\qquad\quad \mathit{xs} \\
&\qquad\quad \mathit{union} \\
&\qquad\quad \mathit{Set}[\ x\ \leftarrow\ \mathit{xs};\ \ y\ \leftarrow\ f(\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{x}\ \ \mathrm{E}_2\ ]\!]\ )\ \mid\ y\ ] \\
&\qquad \mathit{in}\ \ f(\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \mathrm{E}_2\ ]\!]\ ))
\end{aligned} \qquad \text{(orion.29)}$$

Method calls are delimited by space. Set literals are represented as '(E). Their translations and those for brackets, domain variables, identifiers, and constants are given below.

$$\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \mathrm{E}_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ ]\!]\ .\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_2\ ]\!] \qquad \text{(orion.30)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{I}\ (\ \mathrm{E}\ )\ ]\!]\ \Rightarrow\ \mathit{I}\ (\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}\ ]\!]\ ) \qquad \text{(orion.31)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ (\ \mathrm{E}\ )\ ]\!]\ \Rightarrow\ (\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}\ ]\!]\ ) \qquad \text{(orion.32)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ :\mathrm{I}\ ]\!]\ \Rightarrow\ \mathit{I} \qquad \text{(orion.33)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ '(\ \mathrm{E}\ )\ ]\!]\ \Rightarrow\ \mathit{Set}\{\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}\ ]\!]\ \} \qquad \text{(orion.34)}$$

$$\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}\ ]\!]\ \Rightarrow\ \mathit{E} \qquad \text{(orion.35)}$$

5.4.3 Example ORION Translation

Query Q33 presented at the beginning of this subsection can be translated to object comprehensions as follows.

$\mathbf{TQ}_{\text{orion}}$ ⟦ select :c
from :c is_in Courses
where :c exists (recurse set_of prerequisites) code = "Logic1" ⟧

⇒ (orion.5), (orion.10), (orion.35)

Set[*c* ← *Courses*;
$\mathbf{TE}_{\text{orion}}$ ⟦ :c exists (recurse set_of prerequisites) code = "Logic1" ⟧ | $\mathbf{TE}_{\text{orion}}$ ⟦ :c ⟧]

⇒ (orion.16), $\mathbf{TO}_{\text{orion}}$, (orion.35)

Set[*c* ← *Courses*;
some $\mathbf{TE}_{\text{orion}}$ ⟦ :c set_of (recurse set_of prerequisites) code ⟧) = "*Logic*1" | $\mathbf{TE}_{\text{orion}}$ ⟦ :c ⟧]

⇒ (orion.24)

Set[*c* ← *Courses*;
some (
let *f*(*xs* : $\mathbf{TT}_{\text{orion}}$ ⟦ :c (recurse set_of prerequisites) ⟧) *be*
Set[*x* ← *xs* | $\mathbf{TE}_{\text{orion}}$ ⟦ x code ⟧]
in *f*($\mathbf{TE}_{\text{orion}}$ ⟦ :c (recurse set_of prerequisites) ⟧)) = "*Logic*1" | $\mathbf{TE}_{\text{orion}}$ ⟦ :c ⟧]

⇒ $\mathbf{TT}_{\text{orion}}$, (orion.30), (orion.35) twice

Set[*c* ← *Courses*;
some (
let *f*(*xs* : *Set of Course*) *be*
Set[*x* ← *xs* | *x.code*]
in *f*($\mathbf{TE}_{\text{orion}}$ ⟦ :c (recurse set_of prerequisites) ⟧)) = "*Logic*1" | $\mathbf{TE}_{\text{orion}}$ ⟦ :c ⟧]

⇒ (orion.29)

Set[*c* ← *Courses*;
some (
let *f*(*xs* : *Set of Course*) *be*
Set[*x* ← *xs* | *x.code*]
in
f(*let* *g*(*xs* : $\mathbf{TT}_{\text{orion}}$ ⟦ :c prerequisites ⟧) *be*
xs union Set[*x* ← *xs*; *y* ← *g*($\mathbf{TE}_{\text{orion}}$ ⟦ x prerequisites ⟧) | *y*]
in *g*($\mathbf{TE}_{\text{orion}}$ ⟦ :c prerequisites ⟧))) = "*Logic*1" | $\mathbf{TE}_{\text{orion}}$ ⟦ :c ⟧]

$\Rightarrow$ $\mathbf{TT}_{\text{orion}}$, (orion.30) twice, (orion.33) twice, (orion.35) 3 times

```
Set[ c ← Courses;
     some (
     let f( xs : Set of Course ) be
         Set[ x ← xs | x.code ]
     in
     f( let g( xs : Set of Course ) be
            xs union Set[ x ← xs; y ← g( x.prerequisites ) | y ]
        in g( c.prerequisites ) ) ) = "Logic1" | c ]
```

The abstract syntax and translation rules for OSQL and O_2SQL are given in Appendix C and only an example translation is given in each of the next two sections.

5.5 Translating OSQL

5.5.1 Example OSQL Translation

Query Q31 can be expressed using OSQL as follows.

Q31. Return students taking some course given by their supervisors.

```
select    s
for each  Students s, Courses c
where     c in takes(s)
and       c in teaches(supervisedBy(s))
```

$\mathbf{TQ}_{osql}$ ⟦ select s
for each Students s, Courses c
where c in takes(s)
and c in teaches(supervisedBy(s)) ⟧

⇒ (osql.5), (osql.20)

Bag[$\mathbf{TD}_{osql}$ ⟦ Students s, Courses c ⟧;
$\mathbf{TE}_{osql}$ ⟦ c in takes(s) ⟧; $\mathbf{TE}_{osql}$ ⟦ c in teaches(supervisedBy(s)) ⟧ | $\mathbf{TE}_{osql}$ ⟦ s ⟧]

⇒ (osql.17), (osql.18) twice

Bag[*s* ← *Students*; *c* ← *Courses*;
$\mathbf{TE}_{osql}$ ⟦ c in takes(s) ⟧; $\mathbf{TE}_{osql}$ ⟦ c in teaches(supervisedBy(s)) ⟧ | $\mathbf{TE}_{osql}$ ⟦ s ⟧]

⇒ (osql.22), (osql.24), (osql.34) twice

Bag[*s* ← *Students*; *c* ← *Courses*; *c* = *some s.takes*;
$\mathbf{TE}_{osql}$ ⟦ c in teaches(supervisedBy(s)) ⟧ | $\mathbf{TE}_{osql}$ ⟦ s ⟧]

⇒ (osql.22), (osql.28), (osql.34)

Bag[*s* ← *Students*; *c* ← *Courses*; *c* = *some s.takes*;
c = *some* $\mathbf{TC}_{osql}$ ⟦ supervisedBy(s) ⟧
[*x* ← $\mathbf{TE}_{osql}$ ⟦ supervisedBy(s) ⟧; *y* ← *x.teaches* | *y*]
| $\mathbf{TE}_{osql}$ ⟦ s ⟧]

⇒ $\mathbf{TC}_{osql}$, (osql.24), (osql.34) twice

Bag[*s* ← *Students*; *c* ← *Courses*; *c* = *some s.takes*;
c = *some List*[*x* ← *s.supervisedBy*; *y* ← *x.teaches* | *y*] | *s*]

5.6 Translating O_2SQL

5.6.1 Example O_2SQL Translation

Query Q32 can be expressed using O_2SQL as follows.

Q32. Return students having no supervisors from their major departments.

```
select  s
from    s in Students
where   for all l in s.supervisedBy: (l.department <> s.major)
```

$\mathbf{TE}_{o_2sql}$ ⟦ select s
from s in Students
where for all l in s.supervisedBy:(l.department <> s.major) ⟧

⇒ (o2sql.5)

$\mathbf{TC}_{o_2sql}$ ⟦ s in Students ⟧ [$\mathbf{TD}_{o_2sql}$ ⟦ s in Students ⟧;
$\mathbf{TE}_{o_2sql}$ ⟦ for all l in s.supervisedBy: (l.department <> s.major) ⟧ | $\mathbf{TE}_{o_2sql}$ ⟦ s ⟧]

⇒ $\mathbf{TC}_{o_2sql}$, (o2sql.35), (o2sql.20)

Set[*s* ← *Students*;
$\mathbf{TE}_{o_2sql}$ ⟦ for all l in s.supervisedBy: (l.department <> s.major) ⟧ | $\mathbf{TE}_{o_2sql}$ ⟦ s ⟧]

⇒ (o2sql.15)

Set[*s* ← *Students*;
let *f*(*xs* : $\mathbf{TT}_{o_2sql}$ ⟦ s.supervisedBy ⟧) *be*
$\mathbf{TC}_{o_2sql}$ ⟦ s.supervisedBy ⟧ [*l* ← *xs* | $\mathbf{TE}_{o_2sql}$ ⟦ l.department <> s.major ⟧]
in *every* *f*($\mathbf{TE}_{o_2sql}$ ⟦ s.supervisedBy ⟧) = *true* | $\mathbf{TE}_{o_2sql}$ ⟦ s ⟧]

⇒ $\mathbf{TT}_{o_2sql}$, $\mathbf{TC}_{o_2sql}$

Set[*s* ← *Students*;
let *f*(*xs* : *List of Staff*) *be*
List[*l* ← *xs* | $\mathbf{TE}_{o_2sql}$ ⟦ l.department <> s.major ⟧]
in *every* *f*($\mathbf{TE}_{o_2sql}$ ⟦ s.supervisedBy ⟧) = *true* | $\mathbf{TE}_{o_2sql}$ ⟦ s ⟧]

⇒ (o2sql.17), (o2sql.18) 3 times, (o2sql.20) 7 times

Set[*s* ← *Students*;
let *f*(*xs* : *List of Staff*) *be*
List[*l* ← *xs* | *l.department* ~= *s.major*]
in *every* *f*(*s.supervisedBy*) = *true* | *s*]

5.7 Summary

This chapter - and Appendix C - presented four schemes for translating ONTOS SQL, ORION, OSQL, and O_2SQL queries into object comprehensions. An example translation of the interesting features of each query language was given. The translation schemes demonstrated that object comprehensions are at least as powerful as any of these languages with respect to the reference data model. This claim on expressive power is confined to the context of the reference data model because these query languages support features, mainly structural features, that are not supported in the reference data model. The omissions are discussed in Appendix B. To make the translation more elegant, the query languages were simplified. The simplifications and how they can be integrated into the translation schemes are discussed in Appendix B. It should be noted that the simplifications are mainly syntactic rather than computational.

The ability to translate the four query languages into object comprehensions suggests that they can be supported using a single unified platform. The next chapter introduces the canonical algebra which can support object comprehensions and hence can serve as the unified platform.

Chapter 6

Canonical Algebra

One of the main reasons for designing algebras for data models is to use them as vehicles for query optimisation in systems supporting high-level interfaces such as query languages. A large number of algebras have been proposed for data models richer than the relational model [Osb88, Day89, CDLR89, SZ89, SO90, VD90, DD91, WT91, Van92, Alh92, Nor92, AB93, Mit93, GM93, LW93]. Many of these algebras extend the relational algebra to manipulate richer modelling constructs and object identifiers as well as to provide more expressive and computational powers. Others take a fundamentally different approach and draw on the experience of functional programming languages. The *canonical algebra* presented in this chapter belongs to the functional category. It can manipulate the rich constructs found in object-oriented data models and in fact all object comprehension queries can be expressed in the canonical algebra.

The organisation of this chapter is as follows. Section 6.1 presents informally the operations that constitute the canonical algebra. Section 6.2 describes how the Z specification presented in Chapter 2 can be extended to include specifications of these operations. Section 6.3 shows how object comprehension queries can be translated to the canonical algebra. Section 6.4 presents some transformation rules that can be used to optimise canonical algebra expressions. Section 6.6 concludes.

6.1 Operations of the Canonical Algebra

The canonical algebra is designed to support object comprehensions. To provide such support, an algebra must be able to express a wide variety of useful operations over different collection classes. Even better if it can be extended easily to accommodate new collection classes. However, the complexity of having a new ad hoc set of operations for each collection class would be a hindrance not only to reasoning about the operations and their interaction, but also to reliable implementation. To alleviate such complexity, the similarities between collection classes should be exploited to define operations that have analogues from one collection class to the next. In this way, transformation rules for these

operations can be shared among different collection classes.

The design of the canonical algebra aims for simplicity, regularity, and extensibility. The canonical algebra consists of a small number of collection and non-collection operations. Some of the operations are parameterised with functional arguments. This treatment makes the operations more regular as variations can be captured in the functional arguments. Unlike methods found in the object-oriented paradigm, these functions are system-defined and hence amenable to reasoning and therefore optimisation. An example optimisation is given in Section 6.4. The use of functional arguments together with a regular set of collection operations makes the algebra more extensible as a new collection can be integrated by providing a set of the regular operations. However, it should be noted that the canonical algebra is not a minimal set of operations, some operations can be defined by others, for example, *select* is introduced to capture well-known evaluation strategies.

Studies of query language expressive power suggest a set of operations [BBN91, GM93, LW93] very similar to those of the canonical algebra. Many of the canonical algebra operations are also found in other algebras. The canonical algebra can therefore be seen as a synthesis of other algebras.

The rest of the section begins with an explanation of the symbols used in describing the canonical algebra. The algebraic operations and the functional arguments are then informally described. The semantics of the operations are formally defined in Section 6.2. Examples of the application of the algebraic operations can be found in Section 6.3 and 6.4.

6.1.1 Syntactic Categories

Q	Canonical Algebra Operation
F	Function
E	Expression
C	Collection-valued Expression
I	Identifier
k	Constant
α	Logical Operation
ω	Relational Operation (Boolean)
ϕ	Arithmetic Operation
ξ	Collection Kind

An operation that is subscripted with ξ represents a family of three operations one for each collection kind. A subscript indicates the collection kind of the operand collection

while a superscript represents the collection kind of the resultant collection.

6.1.2 Operations

Binary Operations

The *union* and *differ* operations take two operands of the same collection kind and return a resultant collection of that kind. The most specific unique common superclass of the operand element classes will become the class of the elements in the resultant collection.

$Union_{\xi}(\ C_1,\ C_2\ )$

The *union* operations combine two collections. The cardinality of each resultant element is the sum of its cardinalities in the operand collections except in the case of sets where all elements are unique. Ordering, if respected, will be preserved.

$Differ_{\xi}(\ C_1,\ C_2\ )$

The *differ* operations form a collection by removing elements of the second operand collection from the first operand collection. The cardinality of each resultant element is the difference between its cardinality in the first operand collection and that in the second operand collection. Ordering, if respected, will be preserved.

$Equal_{\xi}(\ E_1,\ E_2\ )$

The *equal* operations compare two collections of the same kind and return true if their elements are the same. Duplication and ordering, if respected, will be taken into account.

Unary Operations

The operations described below are unary in the sense that each takes a collection as one of the operands. Other operands include functions on the elements of the operand collection and functions over results returned by other operand functions.

$Reduce_{\xi}(\ E_0,\ F_1,\ F_{aggregate},\ C\ )$

The *reduce* operations are used to combine elements in a collection. If the operand collection C is empty, E_0 is returned. When the operand collection is not empty, F_1 is applied to each element of C and the results are supplied pairwise to $F_{aggregate}$ which accumulates the results to give a single value.

$Map_{\xi}(F, C)$

The *map* operations apply the operand function F to each element in the operand collection C and form a collection containing the results. The resultant collection and operand collection are of the same collection kind.

$Select_{\xi}(F, C)$

The *select* operation applies the operand boolean function F to each element of the operand collection C and forms a collection of the elements for which F returns true. The resultant collection is of the same kind as the operand collection.

$Make_{\xi}^{set}(C)$, $Make_{\xi}^{bag}(C)$, $Make_{\xi}^{list}(C)$

The *make* operations convert the operand collection from its original collection kind to one of the three collection kinds. Conversion from bag or set to list is non-deterministic as an arbitrary order will be assigned to the elements.

$Index(C, E)$

The *index* operation takes a list C and returns an element of the list at position E.

Simple Operations

The following operations take on arguments which may or may not be a collection.

$Empty^{set}(E)$, $Empty^{bag}(E)$, $Empty^{list}(E)$

The *empty* operations take a value and return an empty collection.

$Single^{set}(E)$, $Single^{bag}(E)$, $Single^{list}(E)$

The *single* operations take a value and return a collection containing that value.

$If(E_{condition}, E_{true}, E_{false})$

The *if* operation is a control operation. If $E_{condition}$ evaluates to true, the value of E_{true} is returned, otherwise the value of E_{false} is returned.

And(E_1, E_2)

The *and* operation takes two boolean expressions and returns true if both of them evaluate to true. This is a non-commutative operation and the operands cannot be swapped.

$Range^{set}$(E_1, E_2), $Range^{bag}$(E_1, E_2), $Range^{list}$(E_1, E_2)

The *range* operations generate a collection containing integers within a given range. An empty collection is returned if the first operand expression is less than the second one.

Being(E_1, E_2)

The *being* operation checks if the expression denoted by E_1 has type E_2 or is of a subclass of the class denoted by E_2.

Function Arguments

The abstract syntax of functions passed as arguments to the canonical algebra operations is given below.

$$
\begin{array}{lll}
F & ::= & \lambda I.E \\
Es & ::= & E \mid E,\ Es \\
E & ::= & E.E \mid I(\ Es\ ) \\
 & \mid & E\ \alpha\ E \mid E\ \omega\ E \mid E\ \psi\ E \\
 & \mid & I \mid k \mid (\ E\ ) \mid Q \\
\alpha & ::= & \wedge \mid \vee \mid \neg \\
\omega & ::= & = \mid \sim = \mid > \mid > = \mid < \mid < = \\
\psi & ::= & * \mid / \mid + \mid -
\end{array}
$$

Table 6.1: Abstract Syntax of Function Argument.

6.2 Specifications of the Operations

The Z specification of the reference data model given in Chapter 2 captures the generic definition of a database schema and refrains from discussing system-defined classes that are often used in a database schema. This section describes how the specification can be extended to include the operations of the canonical algebra which are essentially operations on collection classes. A collection class is a system-defined parametric class which generates a proper collection class when given the element type of the collection. In order to facilitate the discussion on collection operations, an abstract representation of collection classes is given next.

6.2.1 Abstract Representation of Collections

$$
\begin{array}{|l}
\textit{COLLECTION\_REPRESENTATION} \\
\hline
\textit{representation}\ \_ : \textit{OCollection} \rightarrow \mathbb{F}\ \textit{VALUE} \times \text{bag}\ \textit{VALUE} \times \text{seq}\ \textit{VALUE} \\
\textit{elementType}\ \_ : \textit{OCollection} \rightarrow \textit{TYPE\_NAME} \\
\textit{elements}_{set}\ \_ : \textit{OSet} \rightarrow \mathbb{F}\ \textit{VALUE} \\
\textit{elements}_{bag}\ \_ : \textit{OBag} \rightarrow \text{bag}\ \textit{VALUE} \\
\textit{elements}_{list}\ \_ : \textit{OList} \rightarrow \text{seq}\ \textit{VALUE} \\
\textit{elements}\ \_ : \textit{OCollection} \rightarrow \mathbb{F}\ \textit{VALUE} \\
\_\text{in}\_ : \textit{VALUE} \leftrightarrow \textit{OCollection} \\
\textit{occurs}\ \_\ \_ : \textit{VALUE} \times \textit{OCollection} \rightarrow \mathbb{N} \\
\hline
\forall\, o : \textit{OCollection};\ s : \textit{OSet};\ b : \textit{OBag};\ l : \textit{OList}; \\
\quad x : \textit{VALUE};\ xs : \mathbb{F}\ \textit{VALUE};\ n : \mathbb{N} \bullet \\
\quad \textit{elements}_{set}\ s = \textit{first}\ (\textit{representation}\ s) \wedge \\
\quad \textit{elements}_{bag}\ b = \textit{second}\ (\textit{representation}\ b) \wedge \\
\quad \textit{elements}_{list}\ l = \textit{third}\ (\textit{representation}\ l) \wedge \\
\quad \textit{elements}\ o = xs \Leftrightarrow \\
\qquad o \in \textit{OSet} \Rightarrow xs = \textit{elements}_{set}\ o \wedge \\
\qquad o \in \textit{OBag} \Rightarrow xs = \text{dom}\ (\textit{elements}_{bag}\ o) \wedge \\
\qquad o \in \textit{OList} \Rightarrow xs = \text{ran}\ (\textit{elements}_{list}\ o) \wedge \\
\quad x\ \text{in}\ o \Leftrightarrow x \in \textit{elements}\ o \wedge \\
\quad \textit{occurs}\ x\ o = n \Leftrightarrow \\
\qquad \neg\ (x\ \text{in}\ o) \Rightarrow n = 0 \wedge \\
\qquad x\ \text{in}\ o \wedge o \in \textit{OSet} \Rightarrow n = 1 \wedge \\
\qquad x\ \text{in}\ o \wedge o \in \textit{OBag} \Rightarrow n = \textit{count}\ (\textit{elements}_{bag}\ o)\ x \wedge \\
\qquad x\ \text{in}\ o \wedge o \in \textit{OList} \Rightarrow n = \#((\textit{elements}_{list}\ o) \upharpoonright \{x\}) \wedge \\
\quad x\ \text{in}\ o \Rightarrow (\textit{typeOf}\ x) \prec\!\!\!\!\cdot\ (\textit{elementType}\ o) \\
\hline
\end{array}
$$

The abstract representation of collections captures both the extension and intension of collections. The extension of a collection is captured by a total function, *representation*, from its object identifier to a triple. Each slot of the triple is a group of values and can be projected out using the function *first*, *second*, and *third*. Each collection class uses one slot to capture the elements in the collection; set uses the first slot, bag the second, and list the third. To simplify access to the elements of a collection, the function $elements_{set}$, $elements_{bag}$, and $elements_{list}$ are introduced and defined in the first three constraints. The elements can also be obtained as a set of values using the function *elements*. Being a set, the result therefore does not reflect duplication and ordering that may exist in the original collection. Understanding how collections are represented in the Z mathematical toolkit [Spi92] is important to the understanding of the definition of this function. A Z bag is defined as a function from values to natural numbers; the latter keeps track of the number of occurrences. A Z sequence is defined as a function from natural numbers to values; the former indicates the position in the sequence. The set of all elements in a Z bag can therefore be obtained by a projection on its domain using *dom*; as for a Z sequence a projection on its range using *ran* can be used. Two more functions are defined to ease the manipulation of collection objects. The relation *in* can be used to test if a value is an element of a collection. The function *occurs* returns the number of occurrences of a value in a collection. In the case of bags, it is defined in terms of the Z operation *count*. For lists, it is defined as the cardinality (#) of a sequence selected ($\upharpoonright$) from the original sequence.

The intension of a collection is captured by a total function, *elementType*, from its object identifier to a type name. All elements in a collection must be of this type or its subclasses.

6.2.2 Binary Operations

The evaluation of the two operations *union* and *differ* in the canonical algebra requires type inference for the elements of the resultant collection. This is achieved by finding the most specific unique common superclass of the operand element classes.

A class is a common superclass to other classes if either they are the same type or the former is a direct or indirect superclass of all the other classes.

COMMON_SUPERCLASS

$$CLASS\_GRAPH$$
$$\_\ \_\ bothAre\ \_ : TYPE\_NAME \times TYPE\_NAME \leftrightarrow TYPE\_NAME$$

$$\forall t_1, t_2, t_3 : TYPE\_NAME \bullet$$
$$(t_1, t_2)\ bothAre\ t_3 \Leftrightarrow$$
$$t_1 \prec\!\!\!\prec t_3 \land t_2 \prec\!\!\!\prec t_3$$

In the presence of multiple inheritance, there can be more than one most specific com-

mon superclass for two given classes. A class is said to be a unique common superclass if it is connected to all other common superclasses via the ISA relationship. Using unique common superclasses makes the type inference deterministic even in the presence of multiple inheritance.

UNIQUE_SUPERCLASS

COMMON_SUPERCLASS

$\_\ \_\ hasUnique\ \_ : TYPE\_NAME \times TYPE\_NAME \leftrightarrow TYPE\_NAME$

$\forall t_1, t_2, t_3 : TYPE\_NAME \bullet$
$(t_1, t_2)\ hasUnique\ t_3 \Leftrightarrow$
$(t_1, t_2)\ bothAre\ t_3 \land$
$\forall t : TYPE\_NAME \mid (t_1, t_2)\ bothAre\ t \bullet t \lessdot t_3 \lor t_3 \lessdot t$

The most specific unique common superclass is a unique common superclass that is a subclass of all other unique common superclasses.

UNIQUE_COMMON_SUPERCLASS

UNIQUE_SUPERCLASS

$\_ \upuparrows \_ : TYPE\_NAME \times TYPE\_NAME \rightarrow TYPE\_NAME$

$\forall t_1, t_2, t_3 : TYPE\_NAME \bullet$
$(t_1 \upuparrows t_2) = t_3 \Leftrightarrow$
$(t_1, t_2)\ hasUnique\ t_3 \land$
$\forall t : TYPE\_NAME \mid (t_1, t_2)\ hasUnique\ t \bullet t_3 \lessdot t$

The *union* operations are defined using operations in the Z mathematical toolkit, namely set union ($\cup$), bag additive union ($\uplus$), and list concatenation ($\frown$). The most specific unique common superclass of the operand element classes becomes the class of the resultant elements.

UNION

UNIQUE_COMMON_SUPERCLASS
COLLECTION_REPRESENTATION
$union_{set}\ \_\ \_ : OSet \times OSet \rightarrow OSet$
$union_{bag}\ \_\ \_ : OBag \times OBag \rightarrow OBag$
$union_{list}\ \_\ \_ : OList \times OList \rightarrow OList$

$\forall s_1, s_2, s_3 : OSet \bullet$
$union_{set}\ s_1\ s_2 = s_3 \Rightarrow$
$elementType\ s_3 = (elementType\ s_1) \Cap (elementType\ s_2) \land$
$elements_{set}\ s_3 = (elements_{set}\ s_1) \cup (elements_{set}\ s_2)$

$\forall b_1, b_2, b_3 : OBag \bullet$
$union_{bag}\ b_1\ b_2 = b_3 \Rightarrow$
$elementType\ b_3 = (elementType\ b_1) \Cap (elementType\ b_2) \land$
$elements_{bag}\ b_3 = (elements_{bag}\ b_1) \uplus (elements_{bag}\ b_2)$

$\forall l_1, l_2, l_3 : OList \bullet$
$union_{list}\ l_1\ l_2 = l_3 \Rightarrow$
$elementType\ l_3 = (elementType\ l_1) \Cap (elementType\ l_2) \land$
$elements_{list}\ l_3 = (elements_{list}\ l_1) \frown (elements_{list}\ l_2)$

Similarly, each collection kind is given a definition of the *differ* operation. In the case of sets, it is defined in terms of Z set difference ($\setminus$). For bags, the computation is carried out directly on the representation of Z bag. An element in the first operand collection is included in the resultant collection if it has more occurrences in the first operand collection than in the second. Its number of occurrences in the resultant collection is the difference of those in the operand collections. When *differ* is applied to lists, the membership of the resultant collection is computed using a function on Z sequences ($\boxminus$).

DIFFER

$$
\begin{array}{l}
UNIQUE\_COMMON\_SUPERCLASS \\
COLLECTION\_REPRESENTATION \\
differ_{set} \_\ \_ : OSet \times OSet \rightarrow OSet \\
differ_{bag} \_\ \_ : OBag \times OBag \rightarrow OBag \\
differ_{list} \_\ \_ : OList \times OList \rightarrow OList \\
\hline
\forall s_1, s_2, s_3 : OSet \bullet \\
\quad differ_{set}\ s_1\ s_2 = s_3 \Rightarrow \\
\qquad elementType\ s_3 = (elementType\ s_1) \upuparrows (elementType\ s_2) \land \\
\qquad elements_{set}\ s_3 = (elements_{set}\ s_1) \setminus (elements_{set}\ s_2) \\
\forall b_1, b_2, b_3 : OBag \bullet \\
\quad differ_{bag}\ b_1\ b_2 = b_3 \Rightarrow \\
\qquad elementType\ b_3 = (elementType\ b_1) \upuparrows (elementType\ b_2) \land \\
\qquad elements_{bag}\ b_3 = \{x : VALUE;\ n_1, n_2 : \mathbb{N} \mid \\
\qquad\qquad (x, n_1) \in elements_{bag}\ b_1 \land \\
\qquad\qquad (x, n_2) \in elements_{bag}\ b_2 \land \\
\qquad\qquad n_1 > n_2 \bullet \\
\qquad\qquad (x, n_1 - n_2)\} \\
\forall l_1, l_2, l_3 : OList \bullet \\
\quad differ_{list}\ l_1\ l_2 = l_3 \Rightarrow \\
\qquad elementType\ l_3 = (elementType\ l_1) \upuparrows (elementType\ l_2) \land \\
\qquad elements_{list}\ l_3 = (elements_{list}\ l_1) \boxminus (elements_{list}\ l_2)
\end{array}
$$

The function $\boxminus$ is defined recursively. When either of the operand sequences is empty, the first operand sequence is returned. Otherwise, the second operand sequence is scanned from t' last element to the first, every t' ne removing the last element before the next recursive call is made. If the last element exists in the first operand sequence, its last occurrence in the first operand sequence is removed before the next recursive call. In brief, elements of the second operand sequence are removed from the first operand sequence, if they exist, in a last to first basis. The Z sequence operation *last* returns the last element in a sequence and *front* returns a list with its last element removed.

$$
\begin{array}{l}
\_ \boxminus \_ : \text{seq } VALUE \times \text{seq } VALUE \rightarrow \text{seq } VALUE \\
\hline
\forall l_1, l_2, l_3 : \text{seq } VALUE \bullet \\
\quad l_1 \boxminus l_2 = l_3 \Leftrightarrow \\
\qquad l_1 = \langle\rangle \lor l_2 = \langle\rangle \Rightarrow l_3 = l_1 \land \\
\qquad \neg\ ((last\ \ l_2) \in (\text{ran}\ \ l_1)) \Rightarrow l_3 = l_1 \boxminus (front\ \ l_2) \land \\
\qquad (last\ \ l_2) \in (\text{ran}\ \ l_1) \Rightarrow \\
\qquad\quad \exists l_a, l_b : \text{seq } VALUE \mid \\
\qquad\qquad\quad l_1 = l_a \frown \langle last\ \ l_2\rangle \frown l_b \land \neg\ ((last\ \ l_2) \in (\text{ran}\ \ l_b)) \bullet \\
\qquad\qquad l_3 = (l_a \frown l_b) \boxminus (front\ \ l_2)
\end{array}
$$

The *equal* operations compares the contents of two collections. As the operations can be defined in the same way only one specification is given below. The definition below relies on the Z equality operation (=) over Z collections which are essentially sets.

EQUAL

COLLECTION_REPRESENTATION

$equal_{set}\ \_\ \_ : OSet \times OSet \rightarrow \mathbb{B}$

$\forall s_1, s_2 : OSet;\ x : \mathbb{B} \bullet$

$equal_{set}\ s_1\ s_2 \Leftrightarrow (elements_{set}\ s_1 = elements_{set}\ s_2)$

6.2.3 Unary Operations

The *reduce* operations transform and combine elements in the operand collection. It has a recursive definition. If the operand collection is empty the first operand value is returned. The second operand is a transformation function and is applied to all elements in a non-empty collection. The third operand is an accumulation function combining results of the transformation function. Except for the domain of the transformation function, all domains and ranges are of the same type. The operation can be defined similarly for the three collection kinds; hence, one specification suffices.

REDUCE_SET

COLLECTION_REPRESENTATION

$reduce_{set}\ \_\ \_\ \_\ \_ : VALUE \times (VALUE \rightarrow VALUE) \times (VALUE \times VALUE \rightarrow VALUE) \times OSet \rightarrow VALUE$

$\forall s : OSet;\ e_0, x : VALUE;\ f_1 : VALUE \rightarrow VALUE;\ f_{agg} : VALUE \times VALUE \rightarrow VALUE \mid$

$elements\ s \subseteq \text{dom}\ f_1 \wedge$

$first\ (\text{dom}\ f_{agg}) = \text{ran}\ f_{agg} \wedge$

$second\ (\text{dom}\ f_{agg}) = \text{ran}\ f_{agg} \wedge$

$\text{ran}\ f_1 = \text{ran}\ f_{agg} \wedge$

$e_0 \in \text{ran}\ f_{agg} \bullet$

$reduce\ e_0\ f_1\ f_{agg}\ s = x \Rightarrow$

$x = fold_{set}\ e_0\ f_1\ f_{agg}\ (element_{set}\ s)$

$\wedge$

$x \in OCollection \Rightarrow$

$elementType\ x = elementTypeOfResult\ f_{agg}$

The result of an *reduce* operation is produced by the function *fold*. If the result is a collection its element type must be inferred - here the function *elementTypeOfResult* is used. Generally speaking, arguments to this function can take two forms: (1) expressions including method calls and (2) nameless functions constructed during the translation of object comprehension queries to the canonical algebra. The keys of type inference on expressions

have been covered in Chapter 2 and in the specification *UNIQUE_COMMON_CLASS*. Constructed functions are limited to very simple forms and the result type can be inferred easily. The *fold* operations are defined as follows.

$$
\begin{array}{|l}
fold_{set} \_\_\_\_ : VALUE \times (VALUE \rightarrow VALUE) \times \\
\qquad\qquad (VALUE \times VALUE \rightarrow VALUE) \times \mathbb{F}\, VALUE \rightarrow VALUE \\
\hline
\forall r, s : \mathbb{F}\, VALUE;\ e_0, x, y : VALUE;\ f_1 : VALUE \rightarrow VALUE; \\
\qquad f_{agg} : VALUE \times VALUE \rightarrow VALUE \mid s = \varnothing \vee s = \{x\} \cup r \bullet \\
\quad fold_{set}\ e_0\ f_1\ f_{agg}\ s = y \Leftrightarrow \\
\qquad s = \varnothing \Rightarrow y = e_0 \wedge \\
\qquad s = \{x\} \cup r \Rightarrow y = f_{agg}\ (f_1\ x)\ (fold_{set}\ e_0\ f_1\ f_{agg}\ r)
\end{array}
$$

$$
\begin{array}{|l}
fold_{bag} \_\_\_\_ : VALUE \times (VALUE \rightarrow VALUE) \times \\
\qquad\qquad (VALUE \times VALUE \rightarrow VALUE) \times \mathrm{bag}\, VALUE \rightarrow VALUE \\
\hline
\forall r, b : \mathrm{bag}\, VALUE;\ e_0, x, y : VALUE;\ f_1 : VALUE \rightarrow VALUE; \\
\qquad f_{agg} : VALUE \times VALUE \rightarrow VALUE \mid b = [\![\,]\!] \vee s = [\![x]\!] \uplus r \bullet \\
\quad fold_{bag}\ e_0\ f_1\ f_{agg}\ b = y \Leftrightarrow \\
\qquad b = [\![\,]\!] \Rightarrow y = e_0 \wedge \\
\qquad b = [\![x]\!] \uplus r \Rightarrow y = f_{agg}\ (f_1\ x)\ (fold_{bag}\ e_0\ f_1\ f_{agg}\ r)
\end{array}
$$

$$
\begin{array}{|l}
fold_{list} \_\_\_\_ : VALUE \times (VALUE \rightarrow VALUE) \times \\
\qquad\qquad (VALUE \times VALUE \rightarrow VALUE) \times \mathrm{seq}\, VALUE \rightarrow VALUE \\
\hline
\forall l : \mathrm{seq}\, VALUE;\ e_0, x, y : VALUE;\ f_1 : VALUE \rightarrow VALUE; \\
\qquad f_{agg} : VALUE \times VALUE \rightarrow VALUE \bullet \\
\quad fold_{list}\ e_0\ f_1\ f_{agg}\ l = y \Leftrightarrow \\
\qquad l = \langle\,\rangle \Rightarrow y = e_0 \wedge \\
\qquad l \neq \langle\,\rangle \Rightarrow y = f_{agg}\ (f_1\ (head\ l))\ (fold_{list}\ e_0\ f_1\ f_{agg}\ (tail\ l))
\end{array}
$$

A *reduce* operation is a homomorphism from *union* to f_{agg}, if f_{agg} satisfies the laws of *union* for that collection kind and has e_0 as an identity element for f_{agg} [WT91]. Since *union* behaves differently for different collection kinds, f_{agg} is therefore required to possess different properties for different collection kinds. The various properties for f_{agg} and e_0 are described next.

If a collection combines itself with an empty collection of the same kind, the same collection is returned. In other words, combining with extra empty collections will have no effect on a collection. In order to normalise the multiple forms of a collection due to empty collection of the same kind, *identity* requires that the accumulation function ignores the result returned by an empty collection and uses only the other operand.

IDENTITY

$$f_{agg} \_\ \_ : VALUE \times VALUE \rightarrow VALUE$$

$$\forall e_0, x, o : VALUE \bullet$$
$$f_{agg}\ x\ e_0 = o \Rightarrow$$
$$o \in OCollection \Rightarrow (equal\ (f_{agg}\ x\ e_0)\ (f_{agg}\ e_0\ x)) \wedge (equal\ o\ x)$$
$$\wedge$$
$$o \in BaseValue \Rightarrow (f_{agg}\ x\ e_0 = f_{agg}\ e_0\ x) \wedge o = x$$

A function is *associative* if the order of combination does not matter. This property is essential to the reasoning of all collection classes particularly when used together with the other properties.

ASSOCIATIVITY

$$f_{agg} \_\ \_ : VALUE \times VALUE \rightarrow VALUE$$

$$\forall x, y, z, o : VALUE \bullet$$
$$f_{agg}\ x\ (f_{agg}\ y\ z) = o \Rightarrow$$
$$o \in OCollection \Rightarrow equal\ (f_{agg}\ x\ (f_{agg}\ y\ z))\ (f_{agg}\ (f_{agg}\ x\ y)\ z)$$
$$\wedge$$
$$o \in BaseValue \Rightarrow f_{agg}\ x\ (f_{agg}\ y\ z) = f_{agg}\ (f_{agg}\ x\ y)\ z$$

Since elements in a set or a bag are not ordered, the same collection can be constructed in many ways using different permutations of the elements. To counter-balance this non-determinism, the accumulation function must be commutative - able to take on arguments in different orders but still delivering the same result.

COMMUTATIVITY

$$f_{agg} \_\ \_ : VALUE \times VALUE \rightarrow VALUE$$

$$\forall x, y, o : VALUE \bullet$$
$$f_{agg}\ x\ y = o \Rightarrow$$
$$o \in OCollection \Rightarrow equal\ (f_{agg}\ x\ y)\ (f_{agg}\ y\ x)$$
$$\wedge$$
$$o \in BaseValue \Rightarrow f_{agg}\ x\ y = f_{agg}\ y\ x$$

A set can be constructed by combining two other sets. If the two sets have a common element, the construction has a destructive effect of eliminating the duplicates leaving only one occurrence of that element in the resultant set. *Idempotence* captures this destructive effect by ignoring duplicates that may exist in the input.

IDEMPOTENCE

$$f_{agg} \_\ \_ : VALUE \times VALUE \rightarrow VALUE$$

$$\forall x, o : VALUE \bullet$$
$$o \in OCollection \Rightarrow equal\ (f_{agg}\ x\ x)\ x$$
$$\wedge$$
$$o \in BaseValue \Rightarrow f_{agg}\ x\ x = x$$

The *reduce* operations are very powerful and many useful operations, e.g. *powerset*, can be expressed in terms of it. If they are used without restriction the operations will take the canonical algebra out of polynomial time [BBN91]. However, in the context in which the canonical algebra is used, the *reduce* operations are used only in a restricted way to support object comprehensions. Therefore all the operand functions are system-defined and satisfy all the properties mentioned above. An example of the use of *reduce* to define other operations can be found in Section 6.3. Since *map*, *select*, and *make* can be expressed using *reduce*, in order to simplify the specification these unary operations are defined in terms of *reduce* in Subsection 6.2.5. Nevertheless, it does not suggest that they should be implemented using *reduce*. It is only the definitions of the operations that are of interest here.

The *index* operation is a fundamental operation to access the element of a list. It is defined only when the operand value corresponds to a valid position in the operand list.

INDEX

COLLECTION_REPRESENTATION

$$\_\ index\ \_ : OList \times \mathbb{N} \nrightarrow VALUE$$

$$\forall l : OList;\ n : \mathbb{N} \mid 1 \leq n \wedge n \leq \#l \bullet$$
$$l\ index\ n = (elements_{list}\ l)\ n$$

6.2.4 Simple Operations

The *empty* operations take an argument and return an empty collection. They get their element types from the argument. An empty collection is represented by a triple of empty Z set ($\varnothing$), empty Z bag ($[\![]\!]$), and empty Z sequence ($\langle\rangle$). Only one specification is given as others can be defined similarly.

EMPTY

COLLECTION_REPRESENTATION

$$empty^{set}\ \_ : VALUE \rightarrow OSet$$

$$\forall x : VALUE;\ s : OSet \bullet$$
$$empty^{set}\ x = s \Leftrightarrow$$
$$elementType\ s = typeOf\ x \wedge representation\ s = (\varnothing, [\![]\!], \langle\rangle)$$

The *single* operations are similar to the *empty* operations except that one slot in the triple contains a Z collection with one element.

SINGLE

COLLECTION_REPRESENTATION

$single^{set}\ \_\ : VALUE \rightarrow OSet$

$\forall\, x : VALUE;\ s : OSet \bullet$

$single^{set}\ x = s \Leftrightarrow$

$elementType\ s = typeOf\ x \land representation\ s = (\{x\}, [\![]\!], \langle\rangle)$

The operation *if* takes three expressions. If the first expression evaluates to true the value of the second expression is returned. Otherwise, the value of the third expression is returned. The second and the third expressions must be of the same type or one's type must be a subclass of the other.

IF

$if\ \_\ \_\ \_\ : \mathbb{B} \times VALUE \times VALUE \rightarrow VALUE$

$\forall\, e_c : \mathbb{B};\ e_t, e_f, x : VALUE \mid e_t \nprec e_f \lor e_f \nprec e_t \bullet$

$if\ e_c\ e_t\ e_f = x \Leftrightarrow$

$e_c \Rightarrow x = e_t \land$

$\neg\, e_c \Rightarrow x = e_f$

The *and* operation takes two expressions. If the first expression evaluates to true the value of the operation is determined by the second expression. Otherwise, false is returned. This is a non-commutative operations and the operands cannot be swapped.

AND

$and\ \_\ \_\ : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

$\forall\, e_1, e_2, x : \mathbb{B} \bullet$

$and\ e_1\ e_2 = x \Leftrightarrow$

$e_1 \Rightarrow x = e_2 \land$

$\neg\, e_1 \Rightarrow x = false$

The *range* operations construct an integer collection containing elements within the limits as specified by the operand values. No duplicate is introduced. In the case of lists, elements are arranged in ascending order.

RANGE

COLLECTION_REPRESENTATION

$range^{set}\ \_\ \_ : \mathbb{N} \times \mathbb{N} \rightarrow OSet$

$range^{bag}\ \_\ \_ : \mathbb{N} \times \mathbb{N} \rightarrow OBag$

$range^{list}\ \_\ \_ : \mathbb{N} \times \mathbb{N} \rightarrow OList$

$\forall s : OSet;\ b : OBag;\ l : OList;\ e_1, e_2 : \mathbb{N} \bullet$

$range^{set}\ e_1\ e_2 = s \Rightarrow$

$elementType\ s = INTEGER \wedge$

$elements_{set}\ s = e_1 .. e_2$

$range^{bag}\ e_1\ e_2 = b \Rightarrow$

$elementType\ b = INTEGER \wedge$

$elements_{bag}\ b = \{x : \mathbb{N} \mid x \in e_1 .. e_2 \bullet (x, 1)\}$

$range^{list}\ e_1\ e_2 = l \Rightarrow$

$elementType\ l = INTEGER \wedge$

$elements_{list}\ l = \{x : \mathbb{N} \mid x \in e_1 .. e_2 \bullet (x - e_1 + 1, x)\}$

Each *being* operation checks if a given value is of the given type or of a subclass of the given class.

BEING

COLLECTION_REPRESENTATION

$being\ \_\ \_ : VALUE \times TYPE\_NAME \rightarrow \mathbb{B}$

$\forall x : VALUE;\ t : TYPE\_NAME \bullet$

$being\ x\ t = (typeOf\ x) \lessdot t$

6.2.5 Derived Unary Operations

The *map* operations apply the operand function to all elements in the operand collection and return a collection containing the results returned by the function.

MAP

$$COLLECTION\_REPRESENTATION$$
$$map_\xi \ \_\ \_ : (VALUE \rightarrow VALUE) \times OCollection \rightarrow OCollection$$

$$\forall o : OCollection; f : VALUE \rightarrow VALUE \bullet$$
$$map_\xi \ f \ o =$$
$$reduce_\xi \ (empty^\xi \ nil) \ f \ union_\xi \ o$$

The *select* operations filter a collection using the boolean operand function. Duplicate elements and their relative order are preserved.

SELECT

$$COLLECTION\_REPRESENTATION$$
$$select_\xi \ \_\ \_ : (VALUE \rightarrow \mathbb{B}) \times OCollection \rightarrow OCollection$$

$$\forall o : OCollection; f : VALUE \rightarrow \mathbb{B} \bullet$$
$$select_\xi \ f \ o =$$
$$reduce_\xi \ (empty^\xi \ nil) \ (\lambda x.if \ (f \ x) \ (single^\xi \ x) \ (empty^\xi \ x)) \ union_\xi \ o$$

The *make* operations convert a collection from one kind to another keeping the element type unchanged. Converting a collection into a set results in the elimination of duplicates and the loss of the order between elements. Converting a collection into a bag keeps the number of elements unchanged - duplicates are not lost and no new elements are introduced - but the order between the elements is lost. Converting a collection into a list keeps the number of elements and an arbitrary order is assigned to the elements. Only one specification is given below others can be defined similarly.

MAKE

$$COLLECTION\_REPRESENTATION$$
$$make_\xi^{set} \ \_ : OCollection \rightarrow OSet$$

$$\forall o : OCollection \bullet$$
$$make_\xi^{set} \ o =$$
$$reduce_\xi \ (empty^{set} \ nil) \ (\lambda x.single^{set} \ x) \ union_{set} \ o$$

6.3 Translating Object Comprehensions

This section demonstrates how object comprehension queries can be translated to the canonical algebra. The presentation of the translation scheme is the same as in Chapter 5.

6.3.1 Syntactic Categories

E	Expression
X	Qualifier
D	Generator
L	Local Definition
Y	Quantifier
A	Aggregate Function
I	Identifier
k	Constant
ω	Relational Operation (Boolean)
ψ	Arithmetic Operation
ξ	Collection Kind

6.3.2 Abstract Syntax

$$
\begin{array}{lcl}
Es & ::= & E \mid E,\ Es \\
\\
E & ::= & E\ \mathit{union}\ E \\
 & \mid & E\ \mathit{differ}\ E \\
 & \mid & \xi[\ Xs\ \mid\ E\ ] \\
\\
 & \mid & E\ \mathit{and}\ E \mid E\ \mathit{or}\ E \mid \mathit{not}\ E \\
 & \mid & E\ \mathit{hasClass}\ E \mid E\ \mathit{hasClass}\ E\ \mathit{with}\ E \\
 & \mid & Y\ E\ \omega\ Y\ E \mid E\ \psi\ E \\
 & \mid & E.E \mid I(\ Es\ ) \mid I \\
 & \mid & k \mid \mathit{Set}\{\ Es\ \} \mid \mathit{Bag}\{\ Es\ \} \mid \mathit{List}\{\ Es\ \} \\
 & \mid & \mathit{Set}\{\ E..E\ \} \mid \mathit{Bag}\{\ E..E\ \} \mid \mathit{List}\{\ E..E\ \} \\
 & \mid & E.[\ E\ ] \mid A\ E \\
 & \mid & (\ E\ )
\end{array}
$$

Table 6.2: Object Comprehensions Abstract Syntax.

$$
\begin{array}{lll}
Xs & ::= & \Lambda \mid X \mid X;\ Xs \\
X & ::= & D \mid L \mid E \\
D & ::= & I \leftarrow E \\
L & ::= & I\ as\ E \\
Y & ::= & \Lambda \mid some \mid atleast\ E \mid just\ E \mid atmost\ E \mid every \\
A & ::= & size \\
\xi & ::= & Set \mid Bag \mid List \\
\omega & ::= & =|\sim=|>|>=|<|<=|==|\sim== \\
\psi & ::= & * \mid / \mid + \mid -
\end{array}
$$

Table 6.2: Object Comprehensions Abstract Syntax (continued).

The translation of query function is essentially the same as of an ordinary query. Query function is therefore not included in the abstract syntax and not covered in the translation.

6.3.3 Translation Functions

TE Translate an Expression
TO Translate an Operation
TC Extract the Kind of a Collection

6.3.4 Translation Rules

A sequence of expression separated by comma is translated as follows.

$$\mathbf{TE}\ [\![\ \mathrm{E}_1\ ,\ \mathrm{E}_2\ ]\!] \ \Rightarrow\ \mathbf{TE}\ [\![\ \mathrm{E}_1\ ]\!]\ ,\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!] \qquad \text{(comp.1)}$$

The two infix collection operations - *union* and *differ* - can be translated as follows. The subscript to the algebraic operation is obtained using **TC** $[\![\ \mathrm{E}_1\ ]\!]$.

$$\mathbf{TE}\ [\![\ \mathrm{E}_1\ \text{union}\ \mathrm{E}_2\ ]\!] \ \Rightarrow\ \mathit{union}_{\mathbf{TC}\ [\![\ \mathrm{E}_1\ ]\!]}(\ \mathbf{TE}\ [\![\ \mathrm{E}_1\ ]\!],\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!]\ ) \qquad \text{(comp.2)}$$

$$\mathbf{TE}\ [\![\ \mathrm{E}_1\ \text{differ}\ \mathrm{E}_2\ ]\!] \ \Rightarrow\ \mathit{differ}_{\mathbf{TC}\ [\![\ \mathrm{E}_1\ ]\!]}(\ \mathbf{TE}\ [\![\ \mathrm{E}_1\ ]\!],\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!]\ ) \qquad \text{(comp.3)}$$

A comprehension without any qualifier represents a singleton collection containing the value E and has collection kind as specified by ξ.

$$\mathbf{TE}\,[\![\,\xi\,[\ |\,\mathrm{E}\ ]\,]\!] \ \Rightarrow\ \mathit{single}^{\xi}(\ \mathbf{TE}\,[\![\,\mathrm{E}\,]\!]\,) \qquad \text{(comp.4)}$$

A generator can be expressed in the algebra using *map*. The range of the generator is converted to the resultant collection kind ξ using *make* before becoming the operand collection of *map*. The rest of the comprehension expression becomes the operand function of *map*.

$$\begin{aligned}&\mathbf{TE}\,[\![\,\xi[\ \mathrm{I}\ \leftarrow \mathrm{E}_1;\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!]\\ &\quad\Rightarrow\ \mathit{map}_{\xi}\ (\lambda I.\mathbf{TE}\,[\![\,\xi[\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!],\ \mathit{make}^{\xi}_{\mathbf{TC}\,[\![\,\mathrm{E}_1\,]\!]}(\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!]))\end{aligned} \qquad \text{(comp.5)}$$

A local definition introduces a new binding for the rest of the query. This effect can be captured using a generator ranging over a singleton collection. Note that it is a transformation between comprehension expressions rather than a translation into the canonical algebra.

$$\mathbf{TE}\,[\![\,\xi[\ \mathrm{I}\ \mathrm{as}\ \mathrm{E}_1;\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!] \ \Rightarrow\ \mathbf{TE}\,[\![\,\xi[\ \mathrm{I}\ \leftarrow \xi\{\ \mathrm{E}_1\ \};\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!] \qquad \text{(comp.6)}$$

A filter can be expressed using the *if* operation. The rest of the query is evaluated if the filter is true, otherwise an empty collection is returned.

$$\mathbf{TE}\,[\![\,\xi[\ \mathrm{E}_1;\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!] \ \Rightarrow\ \mathit{if}(\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!],\ \mathbf{TE}\,[\![\,\xi[\ \mathrm{Q}\ |\,\mathrm{E}\ ]\,]\!],\ \mathit{empty}^{\xi}(\ \mathit{nil}\ )) \qquad \text{(comp.7)}$$

Logical connectives can be translated as shown below.

$$\mathbf{TE}\,[\![\,\mathrm{E}_1\ \mathrm{and}\ \mathrm{E}_2\,]\!] \ \Rightarrow\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!] \wedge \mathbf{TE}\,[\![\,\mathrm{E}_2\,]\!] \qquad \text{(comp.8)}$$

$$\mathbf{TE}\,[\![\,\mathrm{E}_1\ \mathrm{or}\ \mathrm{E}_2\,]\!] \ \Rightarrow\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!] \vee \mathbf{TE}\,[\![\,\mathrm{E}_2\,]\!] \qquad \text{(comp.9)}$$

$$\mathbf{TE}\,[\![\,\mathrm{not}\ \mathrm{E}\,]\!] \ \Rightarrow\ \neg\ \mathbf{TE}\,[\![\,\mathrm{E}\,]\!] \qquad \text{(comp.10)}$$

Class checking is performed using the *being* operation. The non-commutative *and* is used to capture the conditional evaluation in the second translation rule.

$$\mathbf{TE}\,[\![\,\mathrm{E}_1\ \mathrm{hasClass}\ \mathrm{E}_2\,]\!] \ \Rightarrow\ \mathit{being}(\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!],\ \mathbf{TE}\,[\![\,\mathrm{E}_2\,]\!]\,) \qquad \text{(comp.11)}$$

$$\mathbf{TE}\,[\![\,\mathrm{E}_1\ \mathrm{hasClass}\ \mathrm{E}_2\ \mathrm{with}\ \mathrm{E}_3\,]\!] \ \Rightarrow\ \mathit{and}(\ \mathit{being}(\ \mathbf{TE}\,[\![\,\mathrm{E}_1\,]\!],\ \mathbf{TE}\,[\![\,\mathrm{E}_2\,]\!]),\ \mathbf{TE}\,[\![\,\mathrm{E}_3\,]\!]) \qquad \text{(comp.12)}$$

Filters involving quantifiers can be expressed using *reduce*. In the translation rules below, a quantifier is explicitly written on one side of the operator and the other side which is not elaborated may or may not contain a quantifier. When quantifiers are used

on both sides on an operator, the binding order - universal then numerical followed by existential quantifier - determines the meaning of the filter.

$$\mathbf{TE}\,[\![\,\text{every}\ E_1\ \omega\ E_2\,]\!]$$
$$\Rightarrow\ \mathit{reduce}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathit{true},\ \lambda x.\mathbf{TE}\,[\![\,x\ \omega\ E_2\,]\!],\ \wedge,\ \mathbf{TE}\,[\![\,E_1\,]\!]\ ) \qquad \text{(comp.13)}$$

$$\mathbf{TE}\,[\![\,\text{atleast}\ E\ E_1\ \omega\ E_2\,]\!]$$
$$\Rightarrow\ \mathit{reduce}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ 0,\ \lambda x.\mathit{if}\ (\ \mathbf{TE}\,[\![\,x\ \omega\ E_2\,]\!],\ 1,\ 0),\ +,\ \mathit{make}^{\mathit{bag}}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathbf{TE}\,[\![\,E_1\,]\!]\ ))\ >=\ \mathbf{TE}\,[\![\,E\,]\!] \qquad \text{(comp.14)}$$

$$\mathbf{TE}\,[\![\,\text{just}\ E\ E_1\ \omega\ E_2\,]\!]$$
$$\Rightarrow\ \mathit{reduce}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ 0,\ \lambda x.\mathit{if}\ (\ \mathbf{TE}\,[\![\,x\ \omega\ E_2\,]\!],\ 1,\ 0),\ +,\ \mathit{make}^{\mathit{bag}}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathbf{TE}\,[\![\,E_1\,]\!]\ ))\ =\ \mathbf{TE}\,[\![\,E\,]\!] \qquad \text{(comp.15)}$$

$$\mathbf{TE}\,[\![\,\text{atmost}\ E\ E_1\ \omega\ E_2\,]\!]$$
$$\Rightarrow\ \mathit{reduce}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ 0,\ \lambda x.\mathit{if}(\ \mathbf{TE}\,[\![\,x\ \omega\ E_2\,]\!],\ 1,\ 0\ ),\ +,\ \mathit{make}^{\mathit{bag}}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathbf{TE}\,[\![\,E_1\,]\!]\ ))\ <=\ \mathbf{TE}\,[\![\,E\,]\!] \qquad \text{(comp.16)}$$

$$\mathbf{TE}\,[\![\,\text{some}\ E_1\ \omega\ E_2\,]\!]$$
$$\Rightarrow\ \mathit{reduce}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathit{false},\ \lambda x.\mathbf{TE}\,[\![\,x\ \omega\ E_2\,]\!],\ \vee,\ \mathbf{TE}\,[\![\,E_1\,]\!]) \qquad \text{(comp.17)}$$

Instead of comparing two collections by their object identifiers, the equality operation can be used to compare them based on their elements. The translation of other relational and arithmetic operations is captured by a generalised rule.

$$\mathbf{TE}\,[\![\,E_1 == E_2\,]\!]\ \Rightarrow\ \mathit{equal}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathbf{TE}\,[\![\,E_1\,]\!],\ \mathbf{TE}\,[\![\,E_2\,]\!]\ ) \qquad \text{(comp.18)}$$
$$\mathbf{TE}\,[\![\,E_1 \sim == E_2\,]\!]\ \Rightarrow\ \neg\ \mathit{equal}_{\mathbf{TC}\,[\![\,E_1\,]\!]}(\ \mathbf{TE}\,[\![\,E_1\,]\!],\ \mathbf{TE}\,[\![\,E_2\,]\!]\ ) \qquad \text{(comp.19)}$$

$$\mathbf{TE}\,[\![\,E_1\ \omega\ E_2\,]\!]\ \Rightarrow\ \mathbf{TE}\,[\![\,E_1\,]\!]\ \ \mathbf{TO}\,[\![\,\omega\,]\!]\ \ \mathbf{TE}\,[\![\,E_2\,]\!] \qquad \text{(comp.20)}$$

Method calls, identifiers, and constants are translated as follows.

$$\mathbf{TE}\,[\![\,E_1.E_2\,]\!]\ \Rightarrow\ \mathbf{TE}\,[\![\,E_1\,]\!]\,.\,\mathbf{TE}\,[\![\,E_2\,]\!] \qquad \text{(comp.21)}$$
$$\mathbf{TE}\,[\![\,I\,(\,E\,)\,]\!]\ \Rightarrow\ I\,(\,\mathbf{TE}\,[\![\,E\,]\!]\,) \qquad \text{(comp.22)}$$
$$\mathbf{TE}\,[\![\,E\,]\!]\ \Rightarrow\ E \qquad \text{(comp.23)}$$

The translation of collection literals is captured by the four operations: *empty*, *single*, *union*, and *range*.

$$\mathbf{TE}\,[\![\,\xi\{\,\}\,]\!]\ \Rightarrow\ \mathit{empty}^{\xi}(\ \mathit{nil}\ ) \qquad \text{(comp.24)}$$
$$\mathbf{TE}\,[\![\,\xi\{\,E\,\}\,]\!]\ \Rightarrow\ \mathit{single}^{\xi}(\ \mathbf{TE}\,[\![\,E\,]\!]\,) \qquad \text{(comp.25)}$$
$$\mathbf{TE}\,[\![\,\xi\{\,E_1\ ,\ E_2\,\}\,]\!]\ \Rightarrow\ \mathit{union}_{\xi}(\ \mathbf{TE}\,[\![\,\xi\{\,E_1\,\}\,]\!],\ \mathbf{TE}\,[\![\,\xi\{\,E_2\,\}\,]\!]\ ) \qquad \text{(comp.26)}$$
$$\mathbf{TE}\,[\![\,\xi\{\,E_1..E_2\,\}\,]\!]\ \Rightarrow\ \mathit{range}^{\xi}(\ \mathbf{TE}\,[\![\,E_1\,]\!],\ \mathbf{TE}\,[\![\,E_2\,]\!]\ ) \qquad \text{(comp.27)}$$

Elements in a list can be accessed using their position and is supported by the *index* operation.

$$\mathbf{TE}\ [\![\ \mathrm{E}_1.[\mathrm{E}_2]\ ]\!] \ \Rightarrow\ \mathit{index}(\ \mathbf{TE}\ [\![\ \mathrm{E}_1\ ]\!],\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!]\ ) \qquad \text{(comp.28)}$$

As an example of aggregate functions, the size of a collection can be computed using *reduce*. The transformation function is a constant function that always returns the number one. The accumulation function is addition (+). The operand collection is converted to a bag because addition is not idempotent and hence cannot be used on a set. An empty collection naturally gets a zero as the answer.

$$\mathbf{TE}\ [\![\ \mathrm{size\ E}\ ]\!] \ \Rightarrow\ \mathit{reduce}_{bag}(\ 0,\ \lambda x.1,\ +,\ \mathit{make}^{bag}_{\mathrm{TC}\ [\![\ \mathrm{E}\ ]\!]}(\ \mathbf{TE}\ [\![\ \mathrm{E}\ ]\!]\ )\ ) \qquad \text{(comp.29)}$$

The translation rules from object comprehensions to the canonical algebra can be seen as a definition of object comprehensions in terms of the canonical algebra. It is essentially a specification which enables formal reasoning. The implementation is not dictated by the specification and could be very different.

Note that the *select* operations do not appear as a target operation in any of the translation rules from object comprehensions. It only comes into being through transformation of the *map* operation which is described in the next section. As explained earlier, *select* is introduced to capture well-known evaluation strategies, e.g. index scan, that deliver significant performance improvement.

Recursive functions are not supported by a specific algebraic operation. It is assumed that they are treated by the run-time system just like any other ordinary functions. In other words, no attempt will be made to optimise them.

6.3.5 Example Translation

An example is given below to show how Query Q30 used in Chapter 5 can be expressed in the canonical algebra using the translation rules.

$$
\begin{aligned}
&\mathbf{TE}\ [\![\ \text{list[c } \leftarrow \text{ Courses; d } \leftarrow \text{ Departments;} \\
&\qquad \text{d.name = "Computing Science"; d = some c.runBy;} \\
&\qquad \text{1 <= c.credits; c.credits <= 3 | c] }]\!] \\
&\Rightarrow \text{(comp.5)} \\
&\quad map_{list}(\\
&\quad\ \lambda\, c.\mathbf{TE}\ [\![\ \text{list[d } \leftarrow \text{ Departments;} \\
&\qquad\qquad \text{d.name = "Computing Science"; d = some c.runBy;} \\
&\qquad\qquad \text{1 <= c.credits; c.credits <= 3 | c] }]\!], \\
&\quad make^{list}_{\mathbf{TC}\ [\![\ \text{Courses}\ ]\!]}(\ \mathbf{TE}\ [\![\ \text{Courses}\ ]\!]\ )\ ) \\
&\Rightarrow \mathbf{TC}, \text{ (comp.5)} \\
&\quad map_{list}(\\
&\quad\ \lambda\, c.map_{list}(\\
&\qquad \lambda\, d.\mathbf{TE}\ [\![\ \text{list[d.name = "Computing Science"; d = some c.runBy;} \\
&\qquad\qquad\qquad \text{1 <= c.credits; c.credits <= 3 | c] }]\!], \\
&\qquad\ make^{list}_{\mathbf{TC}\ [\![\ \text{Departments}\ ]\!]}(\ \mathbf{TE}\ [\![\ \text{Departments}\ ]\!]\ )\ ), \\
&\quad make^{list}_{set}(\ Courses\ )\ ) \\
&\Rightarrow \mathbf{TC}, \text{ (comp.7)} \\
&\quad map_{list}(\\
&\quad\ \lambda\, c.map_{list}(\\
&\qquad \lambda\, d.if(\ \mathbf{TE}\ [\![\ \text{d.name = "Computing Science"}\ ]\!], \\
&\qquad\qquad \mathbf{TE}\ [\![\ \text{list[d = some c.runBy; 1 <= c.credits; c.credits <= 3 | c] }]\!], \\
&\qquad\qquad empty^{list}(\ nil\ )), \\
&\qquad\ make^{list}_{set}(\ Departments\ )\ ), \\
&\quad make^{list}_{set}(\ Courses\ )\ ) \\
&\Rightarrow \text{(comp.20), (comp.21), (comp.23) 3 times} \\
&\quad map_{list}(\\
&\quad\ \lambda\, c.map_{list}(\\
&\qquad \lambda\, d.if(\ d.name = \text{"}Computing\ Science\text{"}, \\
&\qquad\qquad if(\ \mathbf{TE}\ [\![\ \text{d = some c.runBy}\ ]\!], \\
&\qquad\qquad\quad \mathbf{TE}\ [\![\ \text{list[1 <= c.credits; c.credits <= 3 | c] }]\!], \\
&\qquad\qquad empty^{list}(\ nil\ )\ ), \\
&\qquad\qquad empty^{list}(\ nil\ )\ ), \\
&\qquad\ make^{list}_{set}(\ Departments\ )\ ), \\
&\quad make^{list}_{set}(\ Courses\ )\ )
\end{aligned}
$$

$\Rightarrow$ (comp.17)

$map_{list}($
$\lambda\, c.map_{list}($
$\lambda\, d.if($ $d.name =$ "*Computing Science*",
$if($ $reduce_{\mathbf{TC}\ [\![\ \text{c.runBy}\ ]\!]}($ *false*, $\lambda\, x.\mathbf{TE}\ [\![$ d = x $]\!]$, $\vee$, $\mathbf{TE}\ [\![$ c.runBy $]\!]$),
$\mathbf{TE}\ [\![$ list[1 <= c.credits; c.credits <= 3 | c] $]\!]$,
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$make_{set}^{list}($ *Departments*)),
$make_{set}^{list}($ *Courses*))

$\Rightarrow$ **TC**, (comp.20), (comp.21), (comp.23) 4 times, (comp.7)

$map_{list}($
$\lambda\, c.map_{list}($
$\lambda\, d.if($ $d.name =$ "*Computing Science*",
$if($ $reduce_{set}($ *false*, $\lambda\, x.d = x$, $\vee$, $c.runBy$),
$if($ $\mathbf{TE}\ [\![$ 1 <= c.credits $]\!]$,
$\mathbf{TE}\ [\![$ list[c.credits <= 3 | c] $]\!]$,
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$make_{set}^{list}($ *Departments*)),
$make_{set}^{list}($ *Courses*))

$\Rightarrow$ (comp.7), (comp.20) twice, (comp.21) twice, (comp.4), (comp.23) 7 times

$map_{list}($
$\lambda\, c.map_{list}($
$\lambda\, d.if($ $d.name =$ "*Computing Science*",
$if($ $reduce_{set}($ *false*, $\lambda\, x.d = x$, $\vee$, $c.runBy$),
$if($ 1 <= $c.credits$,
$if($ $c.credits <= 3$,
$single^{list}($ c),
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$empty^{list}($ *nil*)),
$make_{set}^{list}($ *Departments*)),
$make_{set}^{list}($ *Courses*))

6.4 Transforming Canonical Algebra

This section introduces equivalence preserving transformation rules for the canonical algebra. Expression transformation is often used during query optimisation to study the cost of alternative evaluation plans and to search for more efficient evaluation. Here the focus is on rule specification as opposed to rule application. The latter has been shown to be feasible [GD87].

This section delves into the transformation of operations that are not well studied. Well-understood transformation rules, e.g. commutativity and associativity of $\wedge$, $\vee$, and $union_{set}$, are not included. In other words, the transformation rules presented in this section is not a complete set of transformation rules. It is foreseeable that many other transformation rules will be discovered.

Two notions of equivalence are used in the transformation rules. For collection operations, equivalence is defined over the elements of the collections as is supported by the *equal* operation in the canonical algebra. For non-collection operations, equivalence is defined over either object identifiers or base values. Note that all methods are assumed to have no side-effects otherwise the transformation rules may not hold.

6.4.1 Transformation Rules

By definition, a *select* operation is just a special case of a *reduce* operation. The rules below follow directly from the definition of *select*. Selection over an empty collection returns an empty collection. Selection over a singleton collection is the same as using the *if* operation on the element and returning either a singleton or an empty collection. Selection can be distributed over *union*.

$$select_{\xi}(\ \mathrm{F},\ empty^{\xi}(\ nil\ )\ ) \Rightarrow empty^{\xi}(\ nil\ ) \quad \text{(algebra.1)}$$

$$select_{\xi}(\ \mathrm{F},\ single^{\xi}(\ \mathrm{E}\ )\ ) \Rightarrow if(\ (\ \mathrm{F}\ \mathrm{E}\ ),\ single^{\xi}(\ \mathrm{E}\ ),\ empty^{\xi}(\ \mathrm{E}\ )\ ) \quad \text{(algebra.2)}$$

$$select_{\xi}(\ \mathrm{F},\ union_{\xi}(\ \mathrm{E}_1,\ \mathrm{E}_2\ )\ ) \Rightarrow union_{\xi}(\ select_{\xi}(\ \mathrm{F},\ \mathrm{E}_1\ ),\ select_{\xi}(\ \mathrm{F},\ \mathrm{E}_2\ )\ ) \quad \text{(algebra.3)}$$

The *select* operations can be expressed using *map*.

$$map_{\xi}(\ \lambda x.if(\ (\ \mathrm{F}\ x\ ),\ single^{\xi}(\ x\ ),\ empty^{\xi}(\ nil\ )\ ),\ \mathrm{E}\ ) \Rightarrow select_{\xi}(\ \mathrm{F},\ \mathrm{E}\ ) \quad \text{(algebra.4)}$$

If a constant boolean function is passed as the function argument to *select*, the result is either a collection with the same elements or an empty collection.

$$select_{\xi}(\ \lambda x.true,\ \mathrm{E}\ ) \Rightarrow \mathrm{E} \quad \text{(algebra.5)}$$

$$select_{\xi}(\ \lambda x.false,\ \mathrm{E}\ ) \Rightarrow empty^{\xi}(\ nil\ ) \quad \text{(algebra.6)}$$

Applying the *select* operations over the difference of two collections is equivalent to first restricting one collection and then taking the difference.

$$select_{\xi}(\ F,\ differ_{\xi}(\ E_1,\ E_2\ )\ )\ \Rightarrow\ differ_{\xi}(\ select_{\xi}(\ F,\ E_1\ ),\ E_2\ ) \qquad \text{(algebra.7)}$$

Selection is not affected by conversion. Therefore it can be pushed inside the conversion. The promoted *select* operation is applied to E whose collection kind (ξ') may differ from the collection kind *make* produces (ξ).

$$select_{\xi}(\ F,\ make^{\xi}(\ E\ )\ )\ \Rightarrow\ make^{\xi}(\ select_{\xi'}(\ F,\ E\ )\ ) \qquad \text{(algebra.8)}$$

Nested selection can be reduced to a single application of the operation using the conjunction of the original predicates. The property of conjunction also ensures that the order of selection does not matter.

$$select_{\xi}(\ F_1,\ select_{\xi}(\ F_2,\ E\ )\ )\ \Rightarrow\ select_{\xi}(\ \lambda x.F_1\ x\ \wedge\ F_2\ x,\ E\ ) \qquad \text{(algebra.9)}$$

Similarly, the *map* operations can be defined in terms of *reduce*.

$$reduce_{\xi}(\ empty^{\xi}(\ nil\ ),\ F,\ union_{\xi},\ E\ )\ \Rightarrow\ map_{\xi}(\ F,\ E\ ) \qquad \text{(algebra.10)}$$

The result of applying *empty* and *single* to a collection can be easily shown using the properties of *union* which constructs the results returned by them.

$$map_{\xi}(\ empty^{\xi},\ \top\ )\ \Rightarrow\ empty^{\xi}(\ nil\ ) \qquad \text{(algebra.11)}$$

$$map_{\xi}(\ single^{\xi},\ E\ )\ \Rightarrow\ E \qquad \text{(algebra.12)}$$

Again the rules below follow directly from the definition of the operation.

$$map_{\xi}(\ F,\ empty^{\xi}(\ nil\ )\ )\ \Rightarrow\ empty^{\xi}(\ E\ ) \qquad \text{(algebra.13)}$$

$$map_{\xi}(\ F,\ single^{\xi}(\ E\ )\ )\ \Rightarrow\ single^{\xi}(\ F,\ E\ ) \qquad \text{(algebra.14)}$$

$$map_{\xi}(\ F,\ union_{\xi}(\ E_1,\ E_2\ )\ )\ \Rightarrow\ union_{\xi}(\ map_{\xi}(\ F,\ E_1\ ),\ map_{\xi}(\ F,\ E_2\ )\ ) \qquad \text{(algebra.15)}$$

The order of the application of *map* and conversion does not matter.

$$map_{\xi}(\ F,\ make^{\xi}(\ E\ )\ )\ \Rightarrow\ make^{\xi}(\ map_{\xi'}(\ F,\ E\ )\ ) \qquad \text{(algebra.16)}$$

Nested application of *map* can be reduced to a single application using the composition of the original functions. The composition of function F_1 then function F_2 is represented by $F_2 \circ F_1$.

$$map_\xi(\ F_1, map_\xi(\ F_2,\ E\ )\ ) \ \Rightarrow\ map_\xi(\ F_1 \circ F_2,\ E\ ) \qquad \text{(algebra.17)}$$

Accessing an element in the result of a concatenation of two list does not actually require the concatenation to be performed if the sizes of the two operand lists are known. In the first translation rule below, the *size* operation is used to represent the length of a list. Element in a list literal containing consecutive integers can be computed easily.

$$\begin{aligned} index(\ union_{list}(\ E_1,\ E_2\ ),\ E_3\ ) \ \Rightarrow\ & if(\ size(\ E_1\ ) >= E_3, \\ & \quad index(\ E_1,\ E_3\ ), \\ & index(\ E_2,\ E_3 - size(\ E_1\ )\ ) \end{aligned} \qquad \text{(algebra.18)}$$

$$index(\ range(\ E_1,\ E_2\ ),\ E_3\ ) \ \Rightarrow\ E_1 + E_3 - 1 \qquad \text{(algebra.19)}$$

The *equal* operation is reflexive, symmetric, and transitive while the *and* operation is reflexive, asymmetric, and transitive. Only one rule is given to the *if* operation as it is a form that often occurs in the algebraic expressions.

$$if(\ E_1,\ if(\ E_2,\ E_3,\ E_4\ ),\ E_4\ ) \ \Rightarrow\ if(\ E_1 \wedge E_2,\ E_3,\ E_4\ ) \qquad \text{(algebra.20)}$$

6.4.2 Example Transformation

The algebraic expression obtained from Query Q30 can be further simplified using transformation rules on the canonical algebra. First the "nested" *if* expression is transformed into a "flattened" *if* expression. Applying it to the algebraic expression obtained in the previous subsection gives

$$
\begin{array}{l}
\Rightarrow \text{(algebra.20) twice} \\
\quad map_{list}(\\
\quad\quad \lambda\, c.map_{list}(\\
\quad\quad\quad\quad \lambda\, d.if(\ d.name = \text{"}Computing\ Science\text{"} \\
\quad\quad\quad\quad\quad\quad \wedge\ reduce_{set}(\ false,\ \lambda\, x.d = x,\ \vee,\ c.runBy\ ) \\
\quad\quad\quad\quad\quad\quad \wedge\ 1 <= c.credits \\
\quad\quad\quad\quad\quad\quad \wedge\ c.credits <= 3, \\
\quad\quad\quad\quad\quad\quad single^{list}(\ c\ ), \\
\quad\quad\quad\quad\quad empty^{list}(\ nil\ )\ ), \\
\quad\quad\quad make^{list}(\ Departments\ )\ ), \\
\quad\quad make^{list}(\ Courses\ )\ )
\end{array}
$$

This simplified expression can be further transformed by turning the inner *map* application into a *selection* operation.

$$
\begin{array}{l}
\Rightarrow \text{(algebra.4)} \\
\quad map_{list}(\\
\quad\quad \lambda\, c.select_{list}(\\
\quad\quad\quad\quad \lambda\, d.d.name = \text{"}Computing\ Science\text{"} \\
\quad\quad\quad\quad\quad\quad \wedge\ reduce_{set}(\ false,\ \lambda\, x.d = x,\ \vee,\ c.runBy\ ) \\
\quad\quad\quad\quad\quad\quad \wedge\ 1 <= c.credits \\
\quad\quad\quad\quad\quad\quad \wedge\ c.credits <= 3, \\
\quad\quad\quad\quad make^{list}(\ Departments\ )\ ), \\
\quad\quad make^{list}(\ Courses\ )\ )
\end{array}
$$

6.5 Reasoning about Transformation

The validity of the transformation rules given in the previous subsection can be verified using the definitions of the algebraic operations. To verify a rule involves establishing the equivalence of the expressions on each side of the rule. A simple example showing how rule algebra.4 can be verified is given below.

The proof methodology uses a four-column format. The first column is the line number. In the third column is the assertion that is proved based on the assumption given in

the second column. The last column explains the inference used in each step. When an assumption is made in a proof, it is introduced as an assertion having itself as the assumption.

	Assumption	*Assertion*	*Justification*
(1)	1	$map_\xi(\mathrm{F},\ \mathrm{E}) \Leftrightarrow$ $\quad reduce_\xi(empty^\xi(nil),\ \mathrm{F},\ union_\xi,\ \mathrm{E})$	*Definition of Map*
(2)	2	$select_\xi(\mathrm{F},\ \mathrm{E}) \Leftrightarrow$ $\quad reduce_\xi(empty^\xi(nil),$ $\quad\quad \lambda x.if((\mathrm{F}\ x),\ single^\xi(x),\ empty^\xi(x)),$ $\quad\quad union_\xi,$ $\quad\quad \mathrm{E})$	*Definition of Select*
(3)	1	$map_\xi(\mathrm{F},\ \mathrm{E}) \Rightarrow$ $\quad reduce_\xi(empty^\xi(nil),\ \mathrm{F},\ union_\xi,\ \mathrm{E})$	*Tautology* $(A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$ *line* 1
(4)	1	$map_\xi(\lambda x.if((\mathrm{F}\ x),\ single^\xi(x),\ empty^\xi(x)),\ \mathrm{E}) \Rightarrow$ $\quad reduce_\xi(empty^\xi(nil),$ $\quad\quad \lambda x.if((\mathrm{F}\ x),\ single^\xi(x),\ empty^\xi(x)),$ $\quad\quad union_\xi,$ $\quad\quad \mathrm{E})$	(*Implicit*) *Universal Quantifier Elimination* *line* 3
(5)	2	$reduce_\xi(empty^\xi(nil),$ $\quad\quad \lambda x.if((\mathrm{F}\ x),\ single^\xi(x),\ empty^\xi(x)),$ $\quad\quad union_\xi,$ $\quad\quad \mathrm{E}) \Rightarrow$ $\quad select_\xi(\mathrm{F},\ \mathrm{E})$	*Tautology* $(A \wedge (A \Rightarrow B)) \Rightarrow B$ *line* 2
(6)	1, 2	$map_\xi(\lambda x.if((\mathrm{F}\ x),\ single^\xi(x),\ empty^\xi(x)),\ \mathrm{E}) \Rightarrow$ $\quad select_\xi(\mathrm{F},\ \mathrm{E})$	*Tautology* $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ *line* 4, 5

6.6 Summary

Conventional formal query languages are usually presented as either an algebra or a calculus. The canonical algebra described in this chapter mixes algebraic operations with

functions where the latter are expressed in calculus form. In many conventional algebras, such functions appear as restrictions on individual operations. The canonical algebra presumes an approach where the algebraic operations capture the control structures required for manipulating collections and the functions are left to deal with individual elements in the collection. The abstraction of control structures facilitates the combination of operations sharing the same control structure hence incurs a lower cost than evaluating the operations individually. It can also be argued that the emphasis on control structures would benefit more in the object-oriented paradigm where navigation seems to dominate.

Different variations of *reduce* have been studied, for example, in [Van92]. It is a very powerful operation with which many useful operations can be expressed. For instance, *powerset* can be expressed in terms of *reduce*.

$$
\begin{aligned}
&power_{\xi}(\ E\ ) \\
&\quad \Rightarrow\ \ reduce_{\xi}(\ single^{\xi}(\ empty^{\xi}(\ nil\ )\ ), \\
&\qquad\qquad \lambda\, x.union_{\xi}(\ single^{\xi}(\ empty^{\xi}(\ nil\ )\ ),\ \ single^{\xi}(\ single^{\xi}(\ x\ )\ )\ ), \\
&\qquad\qquad \lambda\, x\ y.map_{\xi}(\ \lambda\, m.map_{\xi}(\ \lambda\, n.single^{\xi}(\ union_{\xi}(\ m,\ \ n\ )\ ),\ \ y\ ),\ \ x\ ),\ \ E\ )
\end{aligned}
$$

(algebra.21)

The *select*, *map*, and *make* operations can be defined using *reduce*. Despite being redundant they correspond to well-known implementations where significant performance gain could be obtained. Evaluating them using *reduce* incurs performance penalty.

Variation in data models is another factor which determines the minimal set of operations in an algebra. In the reference data model collections are objects with their own identifiers while in many data models they are just values where equality is determined using only elements in the collections. The dual behaviour of collections in the reference data model requires a larger minimal set of operations and more complex definitions for the operations in the algebra.

Chapter 7

Conclusion

This chapter begins with a discussion summarising the contributions of the research reported in this thesis. It then discusses the limitations of the proposals and the approach taken. The chapter ends with some directions for future work.

7.1 Discussion

This thesis investigated the design and some aspects of the processing of query languages for object-oriented databases using a reference data model formally defined using the specification language Z.

The functional requirements for object-oriented query languages reported in Chapter 3 were partly derived by comparing the similarities and contrasting the differences of existing object-oriented data models and query languages. To come up with a set of requirements that can be meaningfully applied to any object-oriented query language, it was obvious that the similarities should be captured and the differences should not appear in any assumption. For example, class extent is not supported in all models and therefore is not assumed during the study. On the other hand, it was the interest of this research to pursue a wider scope including useful features not yet well studied. Features to support static type checking and multiple collection classes were therefore included.

The 23 functional requirements identified were classified into four categories: support of object-orientation, expressive power, support of collections, and usability. They were used to evaluate and compare existing query languages and the results were summarised in the thesis. The same evaluation also showed that none of the query languages of ONTOS, ORION, IRIS, and O_2 satisfies all the requirements. More importantly, the requirements can be used to improve existing query languages and direct the design of new query languages. The design of object comprehensions was driven by these requirements.

Object comprehensions were designed as a high-level query language for object-oriented databases, particularly those supporting multiple collection classes and static type checking. The example queries given in Chapter 4 served to illustrate that object comprehen-

sions are concise, clear, and powerful enough to express recursive queries. The expressive power of object comprehensions was further demonstrated by providing four translation schemes from the query languages of ONTOS, ORION, IRIS, and O_2 to object comprehensions in Chapter 5. The very existence of these translations substantiates the claim that object comprehensions are at least as powerful as those four query languages with respect to the reference data model. Object comprehensions can be subject to conventional optimisation techniques similar to that reported in [Tri89, Pou89]. New optimisations for class testing and quantifiers were identified and reported. A procedural algebra was also developed to support object comprehensions.

The canonical algebra is a simple procedural algebra supporting multiple collection classes and to which object comprehensions can be translated. A translation scheme was presented in Chapter 6. A set of transformation rules that can be used for optimisation is also given. The canonical algebra essentially defines a platform for the support of object comprehensions.

All the languages mentioned above were studied in the context of a reference data model to which a formal specification was given in Z in Chapter 2. Only features relevant to the study of query language processing were identified and synthesised into the reference data model. Important features of the data model include multi-methods, multiple inheritance, dynamic binding, and static type checking. The canonical algebra plays the role of the data manipulation language of the reference data model. Operations that constitute the algebra were similarly specified in Z. Some properties of the reference data model were proved using the specification. The experience of using Z suggests that a more concise notation may be more appropriate for the purpose of this research.

To conclude, the functional requirements proposed are meaningful and constructive. They are meaningful because they can be used to evaluate and compare existing object-oriented query languages. They are constructive because they can be used to improve existing query languages and direct the design of new query languages of which object comprehensions are an example. Object comprehensions are powerful and optimisable. They are powerful because multiple collection classes can be dealt with, recursive queries can be expressed, and queries expressible in other query languages can be expressed. They are optimisable because some transformation rules are available. The canonical algebra is simple and powerful. It is simple because it consists of a small set of operations. It is powerful because object comprehensions can be supported. In brief, the research described in this thesis represents a step toward a better understanding of the needs and support of object-oriented query languages.

7.2 Limitations

There are a number of limitations to the approach described in this thesis: (1) inadequate analysis on the completeness of object-oriented query languages; (2) inadequate analysis on the complexity of the algebraic operations; and (3) the lack of implementation evidence on optimisation; and (4) the reference data model and query optimisation assume that all method calls terminate.

The thesis began with the identification of a set of functional requirements for object-oriented databases. The research was carried out carefully but the requirements identified cannot be proved to be adequate and sufficient. A formal study of the completeness of object-oriented query languages would provide a definitive answer to this question on expressive power.

Studies of formal query languages aim to strike a balance between expressive power and efficiency. One result of the study of nested relational algebras showed that the *powerset* operation is outside polynomial time [AB93]. Some attention has then been shifted to finding an algebra that delivers the maximum expressive power but having a polynomial-time complexity. This involves studying the complexity of individual operations as in [LW93]. In object-oriented data models, this study would be significantly complicated by the presence of methods.

The optimisations proposed in this thesis represent a first step to optimisation. They represent a core set of rules that can be used in a rule-based optimiser. Search strategies and related issues in such an optimiser were studied in [Mit93]. Implementation-based optimisation techniques, such as use of indices, clustering, and cost models, are not covered in the thesis. Nor is the generation of execution plan, see [Str90]. Note that the two studies [Mit93, Str90] only considered sets. Recursive queries can be expressed using query functions but their optimisation are not addressed in the thesis. The absence of a suitable platform has precluded the integration of the proposed languages and optimisations into a running system. However, a similar language and its optimisation were prototyped and reported at the early stage of this research [TCH90].

Methods can be non-terminating and the safety of a query language cannot be guaranteed in general. A study of this issue has been recently reported in [PS94]. A consequence of the suggested approach is that a three-valued logic is used excluding some well-known properties of two-valued logics.

7.3 Future Directions

Many avenues of further research, both practical and theoretical, are possible and some are described in the next paragraphs.

Graph Comprehensions. Comprehensions are a promising query notation partly because they are recursively defined and hence could possibly be applied to all recursively

defined collections such as lists and trees. In the object-oriented paradigm, an object can be perceived as a graph of heterogeneous objects - a sort of collection. Unfortunately, graphs cannot be defined recursively and hence do not have an intuitive mapping to comprehensions. One challenge is to extend comprehensions to capture graphs making it a "truly" generic query notation.

Completeness and Tractability. To study the expressive power of object-oriented query languages, it is necessary to have a proper notion and definition of completeness. This is a research area actively pursued by workers on database theories. The development of graph comprehensions would facilitate the reasoning and definition of completeness. An equally actively researched area is the search for more expressive but tractable query languages. This demands more understanding on the interaction of collections, the effect of nested queries, and the complexity of individual operations.

Query Optimisation. The optimisation of object-oriented query languages has been acknowledged as an extremely hard problem. The object-oriented paradigm emphasises extensibility and requires an open architecture for many components in an object-oriented database including the query optimiser. How to handle the extensibility of the paradigm, the richness of the data models in general, and the optimisation of object comprehensions in particular definitely require more research.

A Type System Supporting Multi-Methods. Neither object comprehensions nor the canonical algebra have been given a complete type system. One extension is to add a type system so as to make reasoning easier.

Generic Report Generator. A preliminary study of the development of a generic report generator is being carried out. The objective is to develop a theoretical framework for report generation that can be applied to conventional as well as to advanced data models. The framework is expected to include a *formal report model* which provides a conceptual representation of reports. Generating a report will then involve specifying it in terms of the report model using a *report specification language.* Next, the report generator will automatically generate queries over the corresponding database and perform computation over the results returned by the queries. The data retrieval part of the report specification language is essentially a query language. One of the challenges is to incorporate object comprehensions into the report specification language so that it becomes generic across data models.

View Support. View mechanism is a classic database facility and is traditionally supported using query languages. Whether this collaboration of view support and query languages would suffice in the new generation of object-oriented databases is open to question. The next chapter sets out to answer this question with a view to investigate the feasibility of supporting views using object comprehensions.

Chapter 8

View Support

Views enrich a database with various perspectives through which different applications can access the same underlying database. The semantic complexity of an application is therefore significantly reduced as irrelevant details are hidden. Views have been demonstrated to be a useful tool for managing relational database systems. Object-oriented database systems do not yet support any satisfactory view mechanism. This chapter examines existing proposals, reveals their advantages and disadvantages, and identifies the challenges involved in proposing a satisfactory solution.

The organisation of this chapter is as follows. Section 8.1 argues for the need of view support in object-oriented databases. Section 8.2 explains the nature and use of views in object-oriented databases. Section 8.3 describes and assesses current proposals for view mechanisms. Section 8.4 concludes.

8.1 Rationale

The ability to define user views of a database is a basic requirement. Irrespective of the data model supported, e.g. relational, deductive, or temporal, a database with a schema that serves more than one application program should provide a view mechanism. View mechanisms proposed for object-oriented databases do not emulate those of earlier data models. Such a short-coming damages the usability of object-oriented databases and needs addressing.

Views have been a standard and distinguishing characteristic of databases since the introduction of the ANSI/SPARC architecture [DAF86]. Views must be available in order to make a true progression from file processing to database [EN89]. Without views all applications of the same database use the same general schema; they become contorted by their accommodating the data requirements of other applications which, in turn, leads to inefficiency and error.

Data models other than object-oriented data models have given a higher priority to views. Many relational database management systems facilitate views even when they

have not implemented significant components of the relational model. DB2, for example, does not support referential integrity; database updates and deletions have to be executed carefully. Nevertheless, it does support views. A database with a complex schema of normalised relations can be read by users as though it were application-specific. Attributes or whole tables which are irrelevant to groups of users are hidden. Data, for which users have no access authority, are also hidden.

The class hierarchy of an object-oriented database is just as confusing to a reader as a large relational schema. Nor are object-oriented databases designed to store small quantities of data for single applications. The objects and classes are just as numerous and are intended to be accessed by just as many applications.

However, proposals for support for views in object-oriented databases are unsatisfactory. This chapter explains their limitations and, in at least one case, a conspicuous error. One could argue that they do not offer the limited support that even DB2 can. This can be rectified only by a proper identification of the problems, and, of course, future research for their solutions.

8.2 Views in Object-Oriented Databases

8.2.1 Viewing Object-Oriented Databases

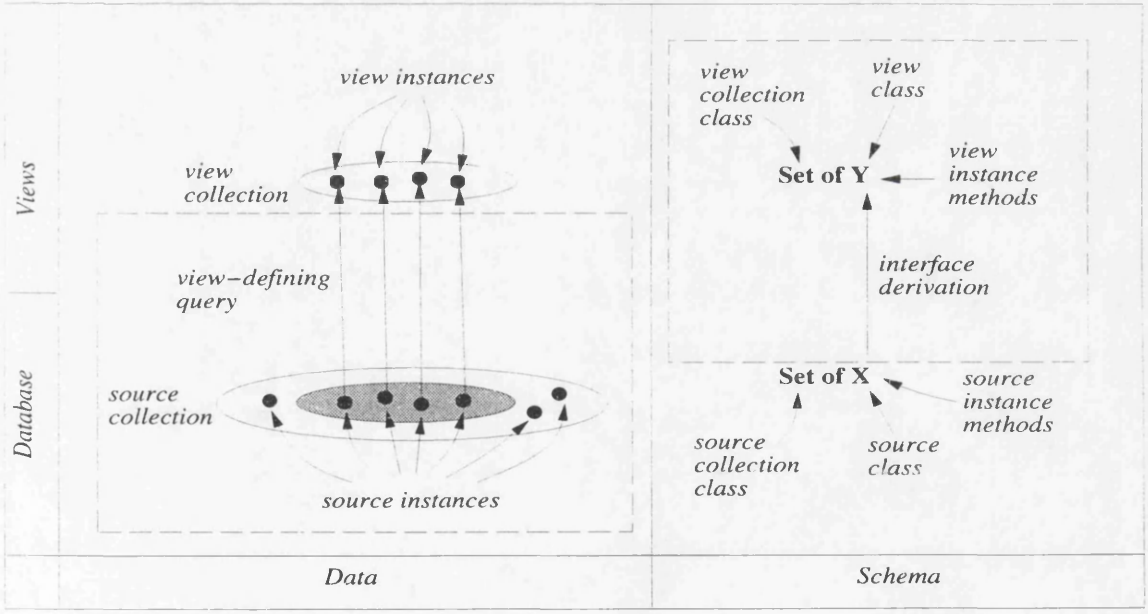

Figure 8.1: Elements of an Object-Oriented View.

Figure 8.1 depicts views in an object-oriented data model. The lower half of the figure depicts a (simplified) database and the upper half depicts a view defined on the database. A view contains a collection of instances that are drawn from a *source collection*. The *view*

instances are the *source instances* that satisfy a *view-defining query.* Type information is given on the right. For the example in Figure 8.1, both the view and the source collections are of class set. The source instances are of class X and the applicable methods are called *source instance methods.* The view instances are of class Y and the applicable methods are called *view instance methods.* The view instance methods can be derived from the source instance methods as will be shown in later sections. In the case of the relational model, only the elements in the left dotted rectangle are sufficient to define a view. This is because the relational model is a structural model and all access methods are generic. For object-oriented databases, all the types must be specified or inferred since each class has its own methods and multiple collection classes are supported. Therefore the additional elements required are the ones in the right dotted rectangle.

In Section 2.14 it was mentioned that an ORION class extent does not include the extents of its subclasses. To define a view, say *People*, to return a class "extent" including instances of *Person* and instances of its subclass *Student*, a query can be used as follows,

$$People = Person \ union \ Student$$

The extents of *Person* and *Student* are the source collections from which the view collection *People* is generated. This is carried out by the view-defining query using the *union* operation. The view instances, elements in *People*, consists of all the elements in the extents of *Person* and *Student* - the source instances. The source instances are of class *Person* or *Student*, which are the source classes, and the view instances are of class *Person* - the view class. Both the source collection classes and the view collection class are set.

3.2.2 Use of vie s

Many suggestions have been made about and reasons given for the possible use of views in object-oriented databases [SS89, HZ90, SLT91, Bra92, Ber92, PMSL94, SAD94]. The ones suggested most often are,

- Information Hiding
- Information Restructuring
- Query Shorthand
- Defining Dynamic Collections
- Testing Schema Changes
- Support of Versions
- Content-based Access Control
- Integrating Heterogeneous Systems
- Data Independence
- Relational Compatibility

Information hiding and restructuring refer to the amount and presentation of information to the users. They focus on individual instances. Query shorthand refers to the construction of a collection that can be used for further querying. Dynamic collections are useful in partitioning an existing collection into smaller collections. They both focus on collections. The effects of schema changes can be studied by simulating the changes using views. If a database contains multiple versions of data a view can be defined to provide

uniform access to the different versions. Access control is usually done by authorisation on a collection of data. Since a view typically contains a collection of elements that satisfy certain criteria, authorisation on such a view collection will effectively provide content-based access control. In a heterogeneous system, views can be used to integrate data from individual systems. As in the relational model the use of views minimises the consequences of schema changes and results in more maintainable systems. Many new object-oriented database users also require access to data stored in their relational systems. Views can be used to provide a gateway between the two systems. This chapter is concerned with four uses of views, namely information hiding, information restructuring, query shorthand, and defining dynamic collections, as they are considered the most important.

8.2.3 The Principal Requirement

The object-oriented paradigm emphasizes reusability. The principal requirement of a view mechanism is therefore to maximise reusability. At the data level it means to minimise the creation of new objects. At the schema level it means to minimise the creation of new classes, introduction of new methods, and the redefinition of existing methods.

8.3 Current Proposals

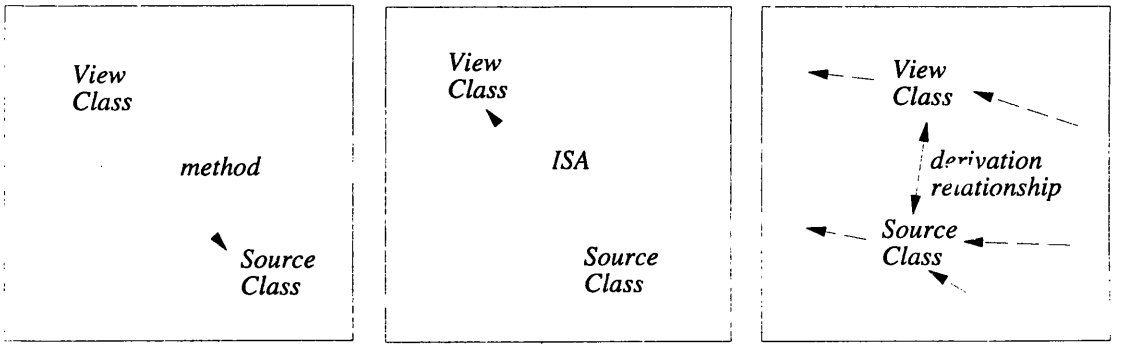

Figure 8.2: The Methodological, Query-Driven, and Schema-Driven Approaches.

The proposals studied in this chapter can be categorised into three approaches. Each category is characterised by the relationship between a view class and a source class (Figure 8.2). In the methodological approach, the relationship is modelled using a method. In the query-driven approach, the relationship is captured by the ISA relationship. The schema-driven approach uses a two-layered structure where a view class is related to the source class not by a method or the ISA relationship but a new *derivation relationship* that is not in the "standard" object-oriented paradigm.

8.3.1 Methodological Approach

Attempts have been made to provide view support by simply applying a methodology that simulates the view mechanism. The major advantage of this approach is that neither new concepts nor implementation are required. However, it is a self-disciplined approach and provides no system support of any kind. The responsibilities for view support rest on the programmer. With the following proposal, for example, that burden eventually proves too great to bear.

A single example of such a strategy suffices. The following discussion, then, is based on [BK93]. For the rest of the section one may equate the approach and this proposal.

Barclay & Kennedy's Proposal

In this proposal, a view is like a subschema which can contain more than one view collection. Using the example database described in Section 2.13, a view containing people under 65 in the set *Persons* and restricting access to the method *get_name* can be defined as in Figure 8.3 (a hypothetical syntax is used).

```
Class A_View_Class                    /* just an ordinary class */
methods                               /* contains only methods */
  young_people → Set of Person        /* returns the view collection */
      be select p                     /* using a view_defining query */
         from   p in Persons          /* on Persons, i.e. the */
         where  p.age < 65,           /* source collection */

  get_name(p : Person) → String       /* view instance method */
      be p.name.
```

Figure 8.3: A View Definition (Barclay & Kennedy).

The view is defined as a class, *A_View_Class*, containing only methods. The view collection is defined by a query and returned by the method *young_people*. In other words, a view collection is populated by existing objects and no new objects are generated. The method *get_name*, which takes a *Person* object as argument, is defined to manipulate the view instances. Using a view involves creating an instance of the class defining the view, invoking the method *young_people* to return the view instances, and applying the method *get_name* to them (Figure 8.4).

```
A_View = A_View_Class.new        /* a new view class object */

select A_View.get_name(x)        /* view instance method */
from   x in A_View.young_people/* view collection */
```

Figure 8.4: Using A View (Barclay & Kennedy).

Two benefits to using this approach are revealed by the example. Query shorthand is supported, i.e. queries on the view instances of the example need not include the selection condition, *age* < *65*. Also, the view is populated by existing objects rather than newly created ones. This makes updating view instances easier.

There are however a number of serious problems. The use of existing objects may make updates straightforward, but it is not clear how to achieve restructuring. These views are based on collections rather than instances. As such, use of the whole collection, as in shorthand querying, will always be simple but use of individual view instances, such as writing methods with view instances as arguments or results, is not. There is no information hiding. The view instances in the example given in Figure 8.4 are still clearly of class *Person*. Therefore, although only *get_name* is supposed to be permitted, any method defined in *Person* may be invoked on a view instance. No new class is defined for the view instances. The instances of *A_View_Class* are complete views not view instances. Thus if a method is written and a view instance is an argument or result, the method writer is compelled to write *Person* in the signature. It is due to the fact that the abstraction for a view is defined at the collection level rather than at the element level.

A further problem relates to the class hierarchy. Since these views are just classes they can form hierarchies. However, a subclass could legally redefine the method which returns the view collection. In the example, a new view could be defined as a subclass of *A_View_Class* with a definition of *young_people* which returns some objects whose *age* is greater than 65, that is, it is possible for a subview to contain instances not in the super-view. This violates the inclusion semantics of inheritance and hence creates a consistency problem.

In summary, the proposal introduces no new facilities to support views so defining a view is rather tedious. Dynamic collections and query shorthand are easily defined but information hiding and restructuring are not provided. View interfaces are provided at too high an abstraction level and view instance interface is distorted and unnatural. The semantics for many possible uses of views, such as consistency and constraints, are poor.

8.3.2 Query-Driven Approach

The query-driven approach covers the proposals in which the definition of a view is primarily a query. As in the relational model this provides a simple mechanism for views but in the new context it becomes rather restrictive. Recall that instances of a class can only be manipulated by methods defined for the class. If a view defined by a query is to be of any use, applicable methods must be defined on the view class. If the view class is going to be a proper class, it must be placed in the class hierarchy. To maximise reusability at the class definition level this often implies that a view class should be placed as close to the source class as possible. In addition, the ISA relationship also asserts that an instance of a subclass is also an instance of its superclasses. To maximise reusability at the instance level, the generation of new instances should be minimised. The ability to maximise reusability is a determining factor for a good query language for view support. However, inserting the view class into the class hierarchy and preserving class instances are both problematic as will be seen in the following proposals.

In describing different proposals, the term "non-class-generating" refers to operations that do not result in the generation of a new class, "class-generating" operations generate either a subclass or a superclass of the source classes, while "new-class-generating" operations generate a direct subclass of the root class irrespective of the position of the source classes in the class hierarchy. A discussion of the various proposals in this approach is given at the end of this subsection.

Davis & Delcambre's Proposal

A query algebra which is formally defined using denotational semantics was proposed by Davis and Delcambre [DD91]. The algebra contains five class-generating operations: $\cup$ (union), $\cap$ (intersection), $-$ (difference), ρ_f (select for all), σ_f (select for some), and two new-class-generating operations: π_f (project) and $\times$ (cross product).

The effect on the class hierarchy after executing the queries: Q1 ($\sigma_{age\ <\ 65}$ *Person*) and Q2 (π_{name} *Person*), where *Person* represents both a class and its extent, is shown in Figure 8.5.

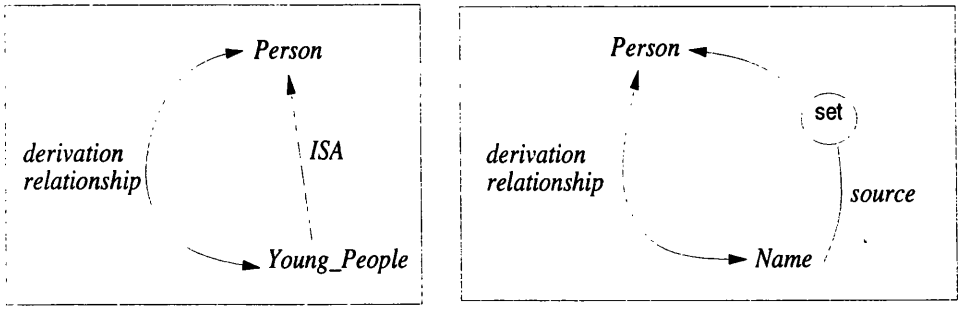

Figure 8.5: Q1: $\sigma_{age\ <\ 65}$ *Person* and Q2: π_{name} *Person*.

Selection (σ_f) results in the generation of a new class, *Young_People*, which is a subclass of the source class, *Person*. The extent of *Young_People* is formed by the elements of the extent of *Person* whose *age* is less than 65. Subclass-generating selection is a widely adopted approach where the extent of a subclass is *reversely* populated according to the extent of its superclass. Such a derivation relationship violates the object-oriented principle of populating a superclass with its subclasses - paradigm problem.

Reverse population is also used by the operations: $\cap$, $-$, and ρ_f. Projection (π_f) generates a new class, in this case *Name*, which is a subclass of the root class. Instances of the new class are created by applying the projecting function f on the elements in the source extent, eliminating the duplicates, and then creating an object for each of them. There is also a system-defined method that links a new object to its source objects (represented by the method *source* in Figure 8.5). Note that this is a set-valued method implying that the one-to-one connection is lost which may cause problems in propagating updates from the view objects back to the source objects - propagation problem. Strictly speaking the generation of new objects is not necessary in some circumstances. The elimination of duplicates is a result of forming tuples in the first step of a projection. This approach, however, allows the restructuring that the previous proposal does not.

Scholl's Proposal

The algebra proposed in [SS90, SLT91] contains two non-class-generating operations: σ_f (select) and $-$ (difference), and the following class-generating operations: π_f (project), ϵ_f (extend), $\cup$ (union), and $\cap$ (intersection).

Selection (σ_f) creates a new collection whose elements satisfy the condition f. Projection (π_f) generates a superclass of the source class and a view collection containing the same elements as in the source collection. The rationale of this tactic is that the ISA relationship can be used to maximise reusability and minimise the generation of new classes.

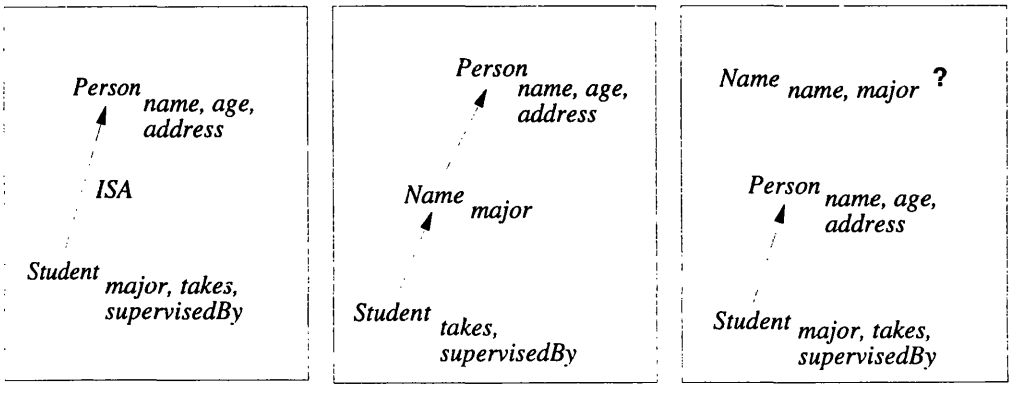

Figure 8.6: Q3: $\pi_{name,address,age,major}$ *Students*, and Q4: $\pi_{name,major}$ *Students*.

The effect on the class hierarchy after executing Q3 ($\pi_{name,address,age,major}$ *Students*) is shown in Figure 8.6 where the new class is inserted between *Person* and *Student.* However, this tactic is not generally applicable as a projection may result in a class that cannot be fitted as a superclass of the source class - class insertion problem. A case in hand is Q4 ($\pi_{name,major}$ *Students*) in Figure 8.6.

This treatment is unsatisfactory even if the new class can be fitted into the hierarchy. In the case of Q3, it is obvious that some kind of schema evolution is happening. The class definition of *Student* is factorised into two parts: one becomes part of the new definition of *Name* and the another becomes the new definition for *Student.* If that is allowed it is not unreasonable to assume that the class definition of *Person* can be updated, say to add the method *ss#*. Now, *Name* has one more method inherited from *Person* even though it is not specified in its view-defining query. This raises concern over equating the ISA relationship with the derivation relationship - evolution problem.

Extension (ϵ_f) adds derived functions to a collection of objects. It essentially defines a subclass by extending the source class with derived methods. It is suggested that the effect of the "join" operation can be achieved by using ϵ_f on the source collections extending their elements with a multi-valued method. This way a many-to-many relationship will be represented as two one-to-many relationships. The other operations: $\cup$, $\cap$, and $-$ have similar semantics as Davis & Delcambre's.

Alhajj's Proposal

In this proposal [Alh92, AA92], there is one non-class-generating operation, σ_f (select), five class-generating operations, $\cup$ (union), $-$ (difference), π_f (project), ϵ_f (extend), and $\times$ (cross product), as well as two new-class-generating operations, α_f (apply) and $\times$ (cross product).

Difference ($-$) has peculiar semantics. If the first source class is the same or a superclass of the second source class, the view class will be the same as the first source class. Assuming *Persons* and *Students* are collections of *Person* and *Student* objects respectively, *Persons* $-$ *Students* returns a view collection containing *Person* objects because *Person* is a superclass of *Student.* This part of the semantics is perfectly acceptable. However, if the condition is not true, the view class will be derived by removing all the methods of the second source class from the first source class. For example, *Students* $-$ *Persons* will give a collection whose elements can be manipulated by only three methods: *major*, *supervisedBy*, and *takes*, but not *name*, *address*, or *age.*

Application (α_f) is a more general form of projection (π_f). It applies the function f to a source collection and generates a new collection containing the results returned by f. No connection from the view instances to the source instances is maintained.

Cross product ($\times$) may be class-generating or new-class-generating depending on the source classes. If they do not have any atomic-valued methods, it generates a subclass of

the source classes. Otherwise, it behaves as Scholl's ϵ or Davis & Delcambre's $\times$. Unlike the latter, no link to the source instances is maintained. It was argued that this context dependent semantics makes $\times$ associative and its implementation efficient. However, when the subclass is generated the semantics is erroneous. It implies that an instance can be migrated to a subclass forming multiple instances of the subclass, each of them sharing the same identifier. This is certainly impossible. Having said that it is believed that most of the time the operation will be of the other kind. That would make it practically the same as Davis & Delcambre's $\times$. Other operations in the algebra are similar to Scholl's.

Discussion

Other algebras have been proposed for querying object-oriented databases, many of them are however retrieval-based and are not designed to support updateable views where maximum reusability and link to the source are essential. Straube's algebra [Str90] cannot generate new objects and does not maintain source links. Dayal's algebra [Day89] and Shaw & Zdonik's algebra [SZ90] rely on tuples and keep no source links. Osborn's algebra [Osb88] and Vandenberg & DeWitt's algebra [VD90] does not maintain source links.

It was mentioned earlier that views can be used to restructure information. To support that using a query-driven approach requires a powerful query language. As expressive power has been covered briefly in previous chapters, the centre of the following discussion is on information hiding and update propagation.

	Scheme A Subclass & Extent	Scheme B Collection
create	dynam	dynamic
insert	must be an instance of the subclass and hence always satisfies the constraints	an instance is of the source class and therefore must be checked against the constraints and propagated to the source collection
update - meet constraints - do not meet constraints	OK insert to the source class	OK reject
delete	remove from the database	remove from the collection
schema update	one subclass	one collection

Table 8.1: A Comparison of the Subclass & Extent Scheme and the Collection Scheme.

It is argued in [Day89] that selection should not result in the generation of a subclass and should be handled using collections. Figure 8.1 contrasts two schemes in terms of the different kinds of updates that can be performed on a view collection and the database

schema. Whether these operations should be supported at all is debatable: Scholl [SLT91] and Abiteboul & Bonner [AB91] expressed very different opinions on the issue. The operations are included here mainly for the purpose of exposition. The semantics suggested for the operations is representative though not definitive, see also [SLT91].

Davis & Delcambre adopt scheme A while Scholl and Alhajj use scheme B. The insertion semantics of scheme B is more complicated and requires constraint checking as well as some trigger mechanism to populate the source collection. Update in scheme A is in some sense information-preserving even though it means an inserted element may not appear in the view collection. The notion of *value-closure* is advocated in [HZ90] meaning that only constraint satisfying updates should be allowed. Deletion in scheme A is difficult and requires an expensive computation. On the contrary, deletion in scheme B is straightforward. Scheme A induces schema updates and an instance in a source extent may end up in many view extents. In other words, the very same object can be of many disjoint classes. This is not supported in many data models. Scheme B generates a dynamic subset for each query on a source collection.

In general scheme B is more desirable as it decouples collections from the class hierarchy and hence minimises changes to the hierarchy. On balance, it could be argued that scheme B is a reasonable compromise between simplicity, flexibility, and functionality. Operations: $\cup$, $\cap$, and $-$ can function in scheme B and take the unique common superclass of the source classes as the class for the elements in the resultant collection.

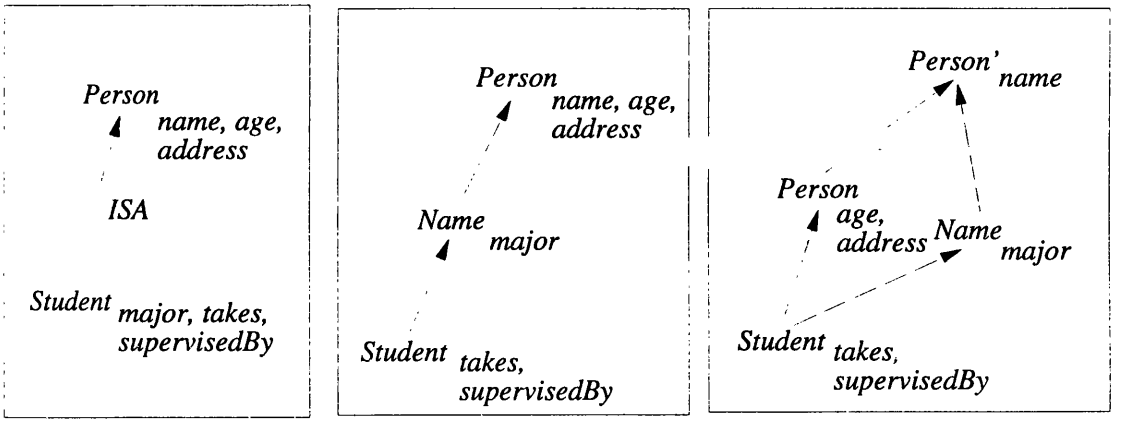

Figure 8.7: Q3: $\pi_{name,address,age,major}$ *Students*, and Q4: $\pi_{name,major}$ *Students*.

Projection has been shown to be rather problematic. Suggestions have been made to rectify the situation [MS89, Alh92, AD94]. For example, in [AD94] an algorithm is given to factorise the class hierarchy so that the view class can be properly placed to inherit methods from existing or factorised classes. Figure 8.7 shows the result of the factorisation caused by the view definition defined by Q4 ($\pi_{name,major}$ *Students*). Note that the resultant hierarchy is arranged in such a way that the view class is immutable to future changes to existing classes. In other words, the evolution problem is eliminated. Also note that

such factorisation may result in change of signatures for methods "promoted" to a new superclass.

Application (α_f) can be used to express both selection and projection. The only concern is how the connection between the source instances and view instances is maintained. Losing this connection will prohibit updates to the view instances from being propagated back to the source instances.

Cross product is used to re-establish relationships not represented explicitly in the database schema. It generates a new object for each pair of source instances. So far the problem of generating new objects, as in π_f, α_f, and $\times$, has not been discussed. Imagine a view named *Student_and_Advisor* defined using $\times$ over a set of *Student* objects and a set of *Staff* objects. To return those pairs where the advisor is from the Computing Science Department and the student is from the Mathematics Department, two possible queries can be used as shown in Figure 8.8.

```
Q5 :   select x
       from   x in Student_and_Advisor
       where  x.advisor.department.name = "Computing Science"
       and    x.student.major.name = "Mathematics"

Q6 :   select x
       from   x in Student_and_Advisor
       where  x.advisor.department.name = "Computing Science"
       and    x in (select y
                    from   y in Student_and_Advisor
                    where  y tudent.major.name = "Mathem cs")
```

Figure 8.8: The View Freezing Problem.

The two queries appear to be the same; however, they may return different results depending on how new objects are generated for the view *Student_and_Advisor*. If new objects are generated each time a view is accessed, two successive accesses to the same view will generate two sets with completely different elements. In query Q6, the range of x and the range of y will be different even though their contents are the same. The fundamental question is should the new identifiers, at least within the same transaction, somehow depend on and freeze with the source objects from which the new objects are generated. A possible solution is to specify some kind of key on the view objects which freezes their object identifiers.

Integrating the query-driven approach with the object-oriented paradigm is not as smooth and easy as it may seem. Its integration does require extra facilities that are not usually explicitly represented in object-oriented data models and, more often than

not, may not be available at all. These facilities include schema evolution operations, instance migration operations, triggers in the case of using the collection scheme, constraint checking, and system-defined methods for the source link. The next section presents a more powerful approach in which a view is not only defined by a query but also augmented with method implementations and so forth.

8.3.3 Schema-Driven Approach

The schema-driven approach covers those proposals in which the derivation of a view consists of a query and a schema-like definition. This approach results in very sophisticated view mechanisms that overcome many problems found in the previous approaches. Nevertheless, this approach is rather verbose and therefore less convenient to use. In order to facilitate comparison a non-specific syntax is used.

Heiler & Zdonik's Proposal

```
Class A_View_Class                    /* just an ordinary class */
attributes
  p : Person,                         /* source instance & class */
methods
  get_name → String                   /* view instance method */
      be p.name.                      /* uses the source instance */

View Young_People = { (               /* view collection */
      A_View_Class,                   /* view class */
      select p                        /* view_defining query */
      from   p in Persons             /* source collection */
      where  p.age < 65
      ) }
```

Figure 8.9: A View Definition (Heiler & Zdonik).

In this proposal [HZ90], a view is defined by first declaring a class for the view instances and then populating a collection with objects of that class using a query. In Figure 8.9 an ordinary class *A_View_Class* is defined whose only attribute is an object of class *Person*. The method *get_name* is implemented by calling *name* on the attribute *p*. The view collection *Young_People* is populated by a query using *Persons* as the source collection. The objects returned by the query are of class *Person* and they are cast to *A_View_Class* before becoming members of *Young_People*. This casting occurs because selection (σ_f) is used. Projection (π_f) and join ($\times$) will result in generation of new objects. This approach relies on the programmer to ensure the consistency between the two parts of the definition:

the class definition and the population of the view collection. Namely, the source class must be the same as the class of the attribute and the selection predicate used in the view-defining query must be included in the appropriate method implementations. The view freezing problem is not addressed.

Saake & Jungclaus's Proposal

```
View Young_People                    /* a view class */
source
  Person p                           /* source class & instance */
query
  p.age < 65                         /* view_defining query */
methods                              /* attributes can be defined */
  get_name → String                  /* view instance method */
constraints
  age < 65.                          /* satisfied by all instances */
```

Figure 8.10: A View Definition (Saake & Jungclaus).

Using Saake & Jungclaus's proposal [SJ92], the view *Young_People* will be defined as in Figure 8.10. Note that this proposal is designed for defining views on class extents. The source class is explicitly specified and the extent of the source class is used as the source collection. A label can be given to the source class as a symbolic name for the source instance. This label can be used within the definition and is like *self* in many programming systems. Unlike the previous proposal, methods that are simply named in the view definition will be inherited and no explicit implementation is required. A construct is provided to specify constraints that must be satisfied by all instances of the view class. Join views can be defined and their instances are values. A source link is maintained in these views. The view freezing problem is not addressed.

Bertino's Proposal

This proposal [Ber92] is very similar to the last one. Both of them are designed to work on class extents. The source classes are inferred in this case and no label is provided to refer to the source instance. The lack of such a label makes it difficult to use view instances as the result of view methods - association problem. Special syntax is introduced in this proposal to get around this problem. Whether a view class will generate new objects can be decided by the user. Constraints are however not supported. Special syntax is provided to make method inheritance easier to specify. Also introduced is the concept of key which controls duplicate elimination. For example, the "join" view used in Figure 8.8 can be defined as in Figure 8.11.

```
View Student_and_Advisor                    /* a join view class */
query
  select s, l                               /* view_defining query */
  from   s in Student, l in Staff
  where  s.advisor = l
  and    s.major.name = "Mathematics"
  and    l.department.name = "Computing Science"
methods
  AllMethods of Student, Staff              /* view instance methods */
generating
  true                                      /* if new objects are generated */
identity
  Student, Staff.                           /* key */
```

Figure 8.11: A View Definition (Bertino).

Abiteboul's Proposal

```
View Young_People(Ps : Set of Person)
query                                       /* a parameterised view class */
  select p                                  /* view_defining query */
  from   p in Ps                            /* argument as source collection */
  where  p.age < 65
methods
  hide age                                  /* inherit all methods but age */
generating
  false.                                    /* if new objects are generated */
```

Figure 8.12: A View Definition (Abiteboul).

This proposal [AB91, SAD94] differs from the previous proposal primarily in the support of parameterised views. As shown in Figure 8.12, the view *Young_People* can be materialised with different sets of *Person* objects on different activations. Syntactic sugar is provided for method hiding. While Bertino's proposal introduces special syntax to deal with the association problem, Abiteboul's proposal provides no support for that. The view freezing problem is fixed by freezing an identifier with the attribute values of a tuple.

Discussion

Heiler & Zdonik's proposal is compositional in nature (similar in spirit to the methodological approach) because view classes are defined as ordinary classes. Casting is used

to turn an attribute into a view instance and at the same time keeps the object identifier. All projected methods have to be redirected and hence explicitly specified. The programmer is required to ensure consistency between the view-defining query and the view class definition, including the proper update semantics. This proposal also works with the collection scheme.

Saake & Jungclaus's proposal takes a quite different perspective as views are defined on top of existing classes without resorting to the ISA relationship and object composition mechanism. Methods can be inherited from the source class easily. Constraints can be specified and are maintained by the system. Basically no new objects are generated and join views are populated with values.

In Bertino's proposal it is even easier to inherit methods. Explicit control is given over the generation of new objects. Like in Heiler & Zdonik's case, the programmer has more responsibility in maintaining the consistency of various parts of a view definition. It is suggested that views can form hierarchies but consistency between them is not discussed. Abiteboul's proposal allows parameterised views to be defined and hence is more appropriate for using with the collection scheme. The view freezing problem is addressed.

The query-driven approach aims to define view classes that behave in the same way as ordinary classes. Methods can be defined as in ordinary classes and overloaded methods are resolved by using the derivation relationship and in some cases, like Bertino's and Abiteboul's, before extending it to the ISA relationship. To support complex objects, a view method should be allowed to return real as well as view instances as the result. This association problem is addressed inelegantly in Bertino's approach and is not addressed in the other proposals. The support of object identifiers demands a solution to the view freezing problem. Abiteboul's proposal seems to be too restrictive while Bertino's may be non-deterministic when two tuples have the same key but otherwise differ in their non-key attributes. A badly addressed area is the formation of a class hierarchy from view classes. In all the proposals studied, a view consists of both a class and a collection. The ISA relationship between view classes does not seem to create any problem. The corresponding view collections are expected to satisfy the subset relationship. However the latter constraint is not imposed in any proposal. To check this constraint, subsumption analysis [BJNS94] could be used but the problem in general is intractable.

8.4 Summary

No object-oriented database system has yet provided a satisfactory view mechanism. Many of the existing proposals can only handle a very small class of view definitions. Reusability is a key idea behind the object-oriented paradigm; however, how the view mechanism can take advantage of this aspect of the paradigm is not yet well understood. Updates in object-oriented views should be more manageable than in the relational model as propagation becomes easier with objects. More research is definitely required in the different aspects of views and hopefully a useful mechanism will be available before long.

Appendix A

Reasoning about Z Specifications

This appendix shows how a property of the reference data model can be studied using a formal proof methodology.

The object-oriented paradigm emphasises reusability and one way to support reusability is to allow incremental development of classes including refinement of methods and method overloading. Choosing between overloaded methods requires some kind of ordering to be imposed on the methods. The local inheritance ordering is used as the basis of ordering in the reference data model. However, it can be applied only at rum-time when the argument types are known. To support static type checking, the generalised inheritance ordering is used at compile-time. However this ordering may result in methods forming a cycle under the method specificity relation. To enable type checking to be carried out at compile-time, the consistency between such methods has to be maintained. The reference data model requires such methods to have the same result type. To ensure that the reference data model actually has this property, a formal proof of a theorem characterising this property is given in this appendix. Three lemmata are used to provide auxiliary results for the proof of the theorem.

A.1 Lemma 1

Lemma 1 deduces that if two methods are consistent and form a cycle under the method specificity relation ($\sqsubseteq$), their result types will similarly form a cycle under the type conformance relation ($\prec\!\!\!\prec$).

	Assumption	*Assertion*	*Justification*
(1)	1	$\forall m_1, m_2 : METHOD;\ s_1, s_2 : SIGNATURE \mid$ $s_1 = m_1.signature \land s_2 = m_2.signature \bullet$ $m_1\ isConsistentWith\ m_2 \Leftrightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\land$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Consistent Schema*
(2)	1	$\forall m_1, m_2 : METHOD;\ s_1, s_2 : SIGNATURE \bullet$ $s_1 = m_1.signature \land s_2 = m_2.signature \Rightarrow$ $m_1\ isConsistentWith\ m_2 \Leftrightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\land$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Definition of* $\mid$ *line* 1
(3)	1	$s_1 = m_1.signature \land s_2 = m_2.signature \Rightarrow$ $m_1\ isConsistentWith\ m_2 \Leftrightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\land$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Universal Quantifier Elimination line* 2 (4 *times*)
(4)	4	$s_1 = m_1.signature$	*Assumption*
(5)	5	$s_2 = m_2.signature$	*Assumption*

	Assumption	*Assertion*	*Justification*
(6)	1, 4, 5	m_1 *isConsistentWith* $m_2 \Leftrightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\wedge$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Tautology* $(A \wedge B \wedge$ $((A \wedge B) \Rightarrow C))$ $\Rightarrow C$ *line* 4, 5, 3
(7)	1, 4, 5	m_1 *isConsistentWith* $m_2 \Rightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\wedge$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Tautology* $(A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$ *line* 6
(8)	8	m_1 *isConsistentWith* m_2	*Assumption*
(9)	1, 4, 5, 8	$(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\wedge$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Tautology* $(A \wedge (A \Rightarrow B)) \Rightarrow B$ *line* 8, 7
(10)	10	$m_1 \sqsubseteq m_2$	*Assumption*
(11)	11	$m_2 \sqsubseteq m_1$	*Assumption*
(12)	1, 4, 5, 8, 10, 11	$s_1.resultType \prec\!\!\!\prec s_2.resultType$ $\wedge$ $s_2.resultType \prec\!\!\!\prec s_1.resultType$	*Tautology* $(A \wedge B \wedge$ $((A \Rightarrow X) \wedge (B \Rightarrow Y)))$ $\Rightarrow (X \wedge Y)$ *line* 10, 11, 9

A.2 Lemma 2

Lemma 2 deduces that if one type conforms to another under type conformance relation ($\prec\!\!\!\ll$), the two types are either equal or transitively related under the ISA relationship ($\prec^*$).

	Assumption	*Assertion*	*Justification*
(1)	1	$\forall t_1, t_2 : TYPE\_NAME \bullet$	*Class*
		$t_1 \prec\!\!\!\ll t_2 \Leftrightarrow$	*Graph*
		$\{t_1, t_2\} \subseteq ClassName \land t_1 \prec^* t_2$	
		$\lor$	
		$\{t_1, t_2\} \subseteq BaseTypeName \land t_1 = t_2$	
(2)	1	$t_1 \prec\!\!\!\ll t_2 \Leftrightarrow$	*Universal*
		$\{t_1, t_2\} \subseteq ClassName \land t_1 \prec^* t_2$	*Quantifier*
		$\lor$	*Elimination*
		$\{t_1, t_2\} \subseteq BaseTypeName \land t_1 = t_2$	*line* 1
			(*twice*)
(3)	1	$t_1 \prec\!\!\!\ll t_2 \Rightarrow$	*Tautology*
		$\{t_1, t_2\} \subseteq ClassName \land t_1 \prec^* t_2$	$(A \Leftrightarrow B) \Rightarrow (A \Rightarrow B)$
		$\lor$	*line* 2
		$\{t_1, t_2\} \subseteq BaseTypeName \land t_1 = t_2$	
(4)	4	$t_1 \prec\!\!\!\ll t_2$	*Assumption*
(5)	1.4	$\{t_1, t_2\} \subseteq ClassName \land t_1 \prec^* t_2$	*Tautology*
		$\lor$	$(A \land (A \Rightarrow B)) \Rightarrow B$
		$\{t_1, t_2\} \subseteq BaseTypeName \land t_1 = t_2$	*line* 4, 3

A.3 Lemma 3

Lemma 3 develops on Lemma 2 and proves that if two types form a cycle under the type conformance relation ($\prec\!\!\!\prec$), they must be the same type.

	Assumption	*Assertion*	*Justification*
(1)	1	$\forall t_1, t_2 : TYPE\_NAME \bullet$ $t_1 \prec\!\!\!\prec t_2 \Leftrightarrow$ $\{t_1, t_2\} \subseteq ClassName \wedge t_1 \prec^* t_2$ $\vee$ $\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2$	*Class Graph*
(2)	2	$t_1 \prec\!\!\!\prec t_2$	*Assumption*
(3)	3	$t_2 \prec\!\!\!\prec t_1$	*Assumption*
(4)	1, 2	$(\{t_1, t_2\} \subseteq ClassName \wedge t_1 \prec^* t_2)$ $\vee$ $(\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2)$	*Lemma* 2 *line* 1, 2
(5)	1, 3	$(\{t_2, t_1\} \subseteq ClassName \wedge t_2 \prec^* t_1)$ $\vee$ $(\{t_2, t_1\} \subseteq BaseTypeName \wedge t_2 = t_1)$	*Lemma* 2 *line* 1, 3
(6)	1, 3	$(\{t_1, t_2\} \subseteq ClassName \wedge t_2 \prec^* t_1)$ $\vee$ $(\{t_1, t_2\} \subseteq BaseTypeName \wedge t_2 = t_1)$	*Property of Set* *line* 5 *(twice)*
(7)	1, 3	$(\{t_1, t_2\} \subseteq ClassName \wedge t_2 \prec^* t_1)$ $\vee$ $(\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2)$	*Property of Equality* *line* 6
(8)	1, 2, 3	$((\{t_1, t_2\} \subseteq ClassName \wedge t_1 \prec^* t_2)$ $\vee (\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2))$ $\wedge$ $((\{t_1, t_2\} \subseteq ClassName \wedge t_2 \prec^* t_1)$ $\vee (\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2))$	*Tautology* $(A \wedge B) \Rightarrow (A \wedge B)$ *line* 4, 7
(9)	1, 2, 3	$((\{t_1, t_2\} \subseteq ClassName \wedge t_1 \prec^* t_2)$ $\wedge (\{t_1, t_2\} \subseteq ClassName \wedge t_2 \prec^* t_1))$ $\vee$ $(\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2)$	*Tautology* $((X \vee A) \wedge (Y \vee A))$ $\Rightarrow ((X \wedge Y) \vee A)$ *line* 8

	Assumption	*Assertion*	*Justification*
(10)	1, 2, 3	$(\{t_1, t_2\} \subseteq ClassName$ $\wedge\ t_1 \prec^* t_2$ $\wedge\ t_2 \prec^* t_1)$ $\vee$ $(\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2)$	*Tautology* $(((A \wedge X) \wedge (A \wedge Y))$ $\vee B) \Rightarrow$ $((A \wedge X \wedge Y) \vee B)$ *line* 9
(11)	11	$\{t_1, t_2\} \subseteq ClassName$ $\wedge\ t_1 \prec^* t_2 \wedge t_2 \prec^* t_1$	*Assumption*
(12)	11	$\{t_1, t_2\} \subseteq ClassName$	*Tautology* $(A \wedge B) \Rightarrow A$ *line* 11
(13)	11	$t_1 \prec^* t_2 \wedge t_2 \prec^* t_1$	*Tautology* $(A \wedge B) \Rightarrow B$ *line* 11
(14)	11	$(t_1 = t_2 \vee t_1 \prec^+ t_2)$ $\wedge$ $(t_2 = t_1 \vee t_2 \prec^+ t_1)$	*Definition of* * *line* 13 (*twice*)
(15)	11	$(t_1 = t_2 \vee t_1 \prec^+ t_2)$ $\wedge$ $(t_1 = t_2 \vee t_2 \prec^+ t_1)$	*Property of Equality* *line* 14
(16)	11	$t_1 = t_2$ $\vee$ $(t_1 \prec^+ t_2 \wedge t_2 \prec^+ t_1)$	*Tautology* $((A \vee X) \wedge (A \vee Y)) \Rightarrow$ $(A \vee (X \wedge Y))$ *line* 15
(17)	17	$t_1 \prec^+ t_2 \wedge t_2 \prec^+ t_1$	*Assumption*
(18)	17	$t_1 \prec^+ t_1$	*Definition of* $^+$ *line* 17
(19)		$(t_1 \prec^+ t_2 \wedge t_2 \prec^+ t_1) \Rightarrow t_1 \prec^+ t_1$	*Conditional Proof line* 18
(20)	11	$t_1 = t_2 \vee t_1 \prec^+ t_1$	*Tautology* $((A \vee X) \wedge (X \Rightarrow Y)) \Rightarrow$ $(A \vee Y)$ *line* 16, 19

	Assumption	*Assertion*	*Justification*
(21)	21	$\forall\, cn : \mathit{ClassName} \bullet \neg\, (cn \prec^{+} cn)$	*ISA*
(22)	11	$t_1 \in \mathit{ClassName}$ $\wedge$ $t_2 \in \mathit{ClassName}$	*Property of Set* *line* 12
(23)	11	$t_1 \in \mathit{ClassName}$	*Tautology* $(A \wedge B) \Rightarrow A$ *line* 22
(24)	11, 21	$\neg\, (t_1 \prec^{+} t_1)$	*Universal* *Quantifier* *Elimination* *line* 21
(25)	11, 21	$t_1 = t_2$	*Tautology* $((A \vee B) \wedge \neg\, B) \Rightarrow A$ *line* 20, 24
(26)	21	$(\{t_1, t_2\} \subseteq \mathit{ClassName} \wedge$ $t_1 \prec^{*} t_2 \wedge$ $t_2 \prec^{*} t_1) \Rightarrow$ $t_1 = t_2$	*Conditional Proof* *line* 25
(27)	1, 2, 3, 21	$t_1 = t_2$ $\vee$ $(\{t_1, t_2\} \subseteq \mathit{BaseTyName} \wedge t_1 = t_2)$	*Tautology* $((X \vee B) \wedge (X \Rightarrow Y)) \Rightarrow$ $(Y \vee B)$ *line* 10, 26
(28)	1, 2, 3, 21	$t_1 = t_2$	*Tautology* $(A \vee (B \wedge A)) \Rightarrow A$ *line* 27

A.4 Theorem

This Theorem proves that if two confusable and consistent methods form a cycle under the method specificity relation ($\sqsubseteq$), their result types must be the same.

	Assumption	*Assertion*	*Justification*
(1)	1	$\forall\, ms : allConfusableSets \mid \#ms > 1 \bullet$ $\forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Database Schema*
(2)	1	$\forall\, ms : allConfusableSets \bullet$ $\#ms > 1 \Rightarrow$ $\forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Definition of* | *line* 1
(3)	1	$\#ms > 1 \Rightarrow$ $\forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Universal Quantifier Elimination* *line* 2
(4)	4	$\#ms > 1$	*Assumption*
(5)	1, 4	$\forall\, m_1, m_2 : ms \mid m_1 \neq m_2 \bullet$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Tautology* $(A \wedge (A \Rightarrow B)) \Rightarrow B$ *line* 4, 3
(6)	1, 4	$\forall\, m_1, m_2 : ms \bullet$ $m_1 \neq m_2 \Rightarrow$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Definition of* | *line* 5
(7)	1, 4	$m_1 \neq m_2 \Rightarrow$ $m_1\ isConfusableWith\ m_2 \Rightarrow$ $m_1\ isConsistentWith\ m_2$	*Universal Quantifier Elimination* *line* 6 (*twice*)
(8)	8	$m_1 \neq m_2$	*Assumption*

	Assumption	*Assertion*	*Justification*
(9)	1, 4, 8	m_1 *isConfusableWith* $m_2 \Rightarrow$ m_1 *isConsistentWith* m_2	*Tautology* $(A \wedge (A \Rightarrow B))$ $\Rightarrow B$ *line* 8, 7
(10)	10	m_1 *isConfusableWith* m_2	*Assumption*
(11)	1, 4, 8, 10	m_1 *isConsistentWith* m_2	*Tautology* $(A \wedge (A \Rightarrow B))$ $\Rightarrow B$ *on line* 10, 9
(12)	12	$m_1 \sqsubseteq m_2$	*Assumption*
(13)	13	$m_2 \sqsubseteq m_1$	*Assumption*
(14)	14	$\forall m_1, m_2 : METHOD;\ s_1, s_2 : SIGNATURE \mid$ $s_1 = m_1.signature \wedge s_2 = m_2.signature \bullet$ m_1 *isConsistentWith* $m_2 \Leftrightarrow$ $(m_1 \sqsubseteq m_2) \Rightarrow$ $s_1.resultType \lessdot s_2.resultType$ $\vee$ $(m_2 \sqsubseteq m_1) \Rightarrow$ $s_2.resultType \lessdot s_1.resultType$	*Consistent* *Schema*
(15)	15	$s_1 = m_1.signature$	*Assumption*
(16)	16	$s_2 = m_2.signature$	*Assumption*
(17)	14, 15, 16, 1, 4, 8, 10, 12, 13	$s_1.resultType \lessdot s_2.resultType$ $\wedge$ $s_2.resultType \lessdot s_1.resultType$	*Lemma* 1 *on line* 14, 15, 16, 11, 12, 13
(18)	18	$\forall t_1, t_2 : TYPE\_NAME \bullet$ $t_1 \lessdot t_2 \Leftrightarrow$ $\{t_1, t_2\} \subseteq ClassName \wedge t_1 \prec^* t_2$ $\vee$ $\{t_1, t_2\} \subseteq BaseTypeName \wedge t_1 = t_2$	*Class* *Graph*

	Assumption	*Assertion*	*Justification*
(19)	19	$\forall\, cn : ClassName \bullet \neg\, (cn \prec^{+} cn)$	*ISA*
(20)	14, 15, 16, 1, 4, 8, 10, 12, 13, 18, 19	$s_1.resultType = s_2.resultType$	*Lemma* 3 18, 17, 19
(21)	14, 15, 16, 1, 4, 8, 18, 19	$(m_1\ isConfusableWith\ m_2 \land$ $m_1 \sqsubseteq m_2 \land$ $m_2 \sqsubseteq m_1) \Rightarrow$ $s_1.resultType = s_2.resultType$	*Conditional Proof line* 20 (3 *times*)
(22)	14, 1, 4, 18, 19	$(m_1 \neq m_2 \land s_1 = m_1.signature \land$ $s_2 = m_2.signature) \Rightarrow$ $((m_1\ isConfusableWith\ m_2 \land$ $m_1 \sqsubseteq m_2 \land$ $m_2 \sqsubseteq m_1) \Rightarrow$ $s_1.resultType = s_2.resultType)$	*Conditional Proof line* 21 (3 *times*)
(23)	14, 1, 4, 18, 19	$\forall\, m_1, m_2 : ms;\ s_1, s_2 : SIGNATURE \bullet$ $(m_1 \neq m_2 \land$ $s_1 = m_1.signature \land$ $s_2 = m_2.signature) \Rightarrow$ $((m_1\ isConfusableWith\ m_2 \land$ $m_1 \sqsubseteq m_2 \land$ $m_2 \sqsubseteq m_1) \Rightarrow$ $s_1.resultType = s_2.resultType)$	*Universal Quantifier Introduction line* 22 (4 *times*)
(24)	14, 1, 4, 18, 19	$\forall\, m_1, m_2 : ms;\ s_1, s_2 : SIGNATURE \mid$ $m_1 \neq m_2 \land$ $s_1 = m_1.signature \land$ $s_2 = m_2.signature \bullet$ $m_1\ isConfusableWith\ m_2$ $\land\ m_1 \sqsubseteq m_2$ $\land\ m_2 \sqsubseteq m_1 \Rightarrow$ $s_1.resultType = s_2.resultType$	*Definiton of* | *line* 23

	Assumption	*Assertion*	*Justification*
(25)	14,	$\#ms > 1 \Rightarrow$	*Conditional*
	1,	$\forall m_1, m_2 : ms;\ s_1, s_2 : SIGNATURE \mid$	*Proof*
	18, 19	$m_1 \neq m_2 \wedge$	*line* 24
		$s_1 = m_1.signature \wedge$	
		$s_2 = m_2.signature \bullet$	
		$m_1\ isConfusableWith\ m_2$	
		$\wedge\ m_1 \sqsubseteq m_2$	
		$\wedge\ m_2 \sqsubseteq m_1 \Rightarrow$	
		$s_1.resultType = s_2.resultType$	
(26)	14,	$\forall ms : allConfusableSets \bullet \#ms > 1 \Rightarrow$	*Universal*
	1,	$\forall m_1, m_2 : ms;\ s_1, s_2 : SIGNATURE \mid$	*Quantifier*
	18, 19	$m_1 \neq m_2 \wedge$	*Introduction*
		$s_1 = m_1.signature \wedge$	*line* 25
		$s_2 = m_2.signature \bullet$	
		$m_1\ isConfusableWith\ m_2$	
		$\wedge\ m_1 \sqsubseteq m_2$	
		$\wedge\ m_2 \sqsubseteq m_1 \Rightarrow$	
		$s_1.resultType = s_2.resultType$	
(27)	14, 1, 18, 19	$\forall ms : allConfusableSets \mid \#ms > 1 \bullet$	*Definition of* $\mid$
		$\forall m_1, m_2 : ms;\ s_1, s_2 : SIGNATURE \mid$	*line* 26
		$m_1 \neq m_2 \wedge$	
		$s_1 = m_1.signature \wedge$	
		$s_2 = m_2.signature \bullet$	
		$m_1\ isConfusableWith\ m_2$	
		$\wedge\ m_1 \sqsubseteq m_2$	
		$\wedge\ m_2 \sqsubseteq m_1 \Rightarrow$	
		$s_1.resultType = s_2.resultType$	

Appendix B

Simplifications

B.1 Simplifications of ONTOS SQL

Multiple Targets. A list of expressions and the wildcard character (*) can be used in the target of an ONTOS SQL query. Since the reference data model does not support tuples therefore these features are excluded from the translation to object comprehensions.

Implicit Domain Variable. When a domain is represented by an identifier, the domain variable can be omitted and the identifier can serve as the domain variable. Whereas object comprehensions require an explicit variable for each domain. One way to deal with such a "shorthand" in the translation is to introduce some resolution mechanism for method calling. If an identifier representing a domain appears at the beginning of an expression it can be replaced by the corresponding domain variable introduced by the translation. For the sake of simplicity and clarity, it is chosen not to include such a resolution mechanism to the translation.

Implicit Receiving Object. Another "shorthand" supported by ONTOS SQL is that a method can be called without a receiving object. The receiving object can be resolved among the domain classes. This can be dealt with in a similar way to implicit domain variables.

B.2 Simplifications of ORION

Value Equality. ORION supports two groups of operations. One group uses object identity and the other group uses value equality to eliminate duplicates. Value equality is considered a violation of the object-oriented principle of encapsulation and is therefore not supported in the reference data model and therefore operations using value equality do not appear in the translation.

Class Extents. Class extents are supported by ORION and can form class expressions using the operations: * (meaning including all subclass extents), *union*, and *difference*. A domain ranged over a class extent can be specified as $I : I$ where the first I represents a

class name and the second *I* denotes an identifier. Class extents are not supported in the reference data model and all these features therefore have no meaningful counterparts in object comprehensions.

Implicit Receiving Object. Similar to ONTOS SQL a method can be called without a receiving object. As discussed in the previous section, this is excluded from the translation.

B.3 Simplifications of OSQL

Data Management. OSQL is designed as a completed data manipulation language that can be used to define new classes, introduce a new method to a class, create new objects, change the class of an object, and so forth. Only the retrieval part of OSQL is covered in the translation and the data management functions are therefore not included.

Multiple Targets. A list of expressions representing the attributes of a tuple can be used in the target of an OSQL query. Since the reference data model does not support tuples therefore this is excluded from the translation.

Grouping. OSQL supports *group by* and *having* as found in SQL which returns tuples rather than objects and therefore are not included in the translation. However, if tuples were supported in the reference data model they could be expressed in object comprehensions. Grouping could be achieved by projecting the grouping keys and then collecting the relevant objects for each key.

Query Functions. In OSQL, functions are used to represent both stored as well as derived data. Derivation can be done in three ways: (1) using a foreign function implemented using a programming language; (2) using a procedural function implemented partly using the control and update statements in OSQL; and (3) using a derived function whose body is pure querying code. Foreign functions have no counterpart in object comprehensions. Procedural functions are essentially used for their side-effects and hence are not considered a querying component for the purpose of this study. A derived function can be represented as a query function in object comprehensions. Translating a derived function is essentially the same as for an ordinary OSQL query and is therefore not covered.

B.4 Simplifications of O_2SQL

Grouping. O_2SQL supports a very powerful grouping operation that returns tuples as the result. If tuples were supported in the reference data model the same operation could be expressed in object comprehensions. Grouping could be achieved by projecting the grouping keys, after computation if necessary, and then collecting the relevant objects for each key.

Query Functions. Non-recursive query functions can be defined in O_2SQL but there is no linking word between a query function and the query that uses it. To translate to

object comprehensions the linking word *in* can be introduced to put the query function in scope for the query in question. As in OSQL, the translation is essentially the same as for an ordinary O_2SQL query and is therefore not covered.

Appendix C

More Translation Rules

C.1 ORION Translation Rules

The simplest case of class testing is translated as follows.

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \text{class}\ \ \mathrm{I}\ \ \omega\ \ \mathrm{E}_2\ ]\!]\\ &\quad\Rightarrow\ \ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ ]\!]\ \textit{hasClass}\ I\ ;\ \ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \omega\ \ \mathrm{E}_2\ ]\!]\end{aligned}\qquad\text{(orion.34)}$$

The existential quantifier *exists* can be used at the beginning of an expression. The translation is shown below.

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ \text{exists}\ \ \mathrm{E}_2\ \ \mathrm{E}_3\ \omega\ \ \mathrm{E}_4\ ]\!]\\ &\quad\Rightarrow\ \ \textit{some}\ \mathbf{TE}_{\text{orion}}\ [\![\ \text{set\_of}\ \ \mathrm{E}_2\ \mathrm{E}_3\ ]\!]\ \ \mathbf{TO}_{\text{orion}}\ [\![\ \omega\ ]\!]\ \ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_4\ ]\!]\end{aligned}\qquad\text{(orion.35)}$$

The universal quantifier *each* can be dealt with in a similar way and the translation of the two possible forms are as follows.

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \ \text{each}\ \ \mathrm{E}_2\ \ \mathrm{E}_3\ \omega\ \ \mathrm{E}_4\ ]\!]\\ &\quad\Rightarrow\ \ \textit{every}\ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_1\ \text{set\_of}\ \mathrm{E}_2\ \mathrm{E}_3\ ]\!]\ \ \mathbf{TO}_{\text{orion}}\ [\![\ \omega\ ]\!]\ \ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_4\ ]\!]\end{aligned}\qquad\text{(orion.36)}$$

$$\begin{aligned}&\mathbf{TE}_{\text{orion}}\ [\![\ \text{each}\ \ \mathrm{E}_2\ \ \mathrm{E}_3\ \omega\ \ \mathrm{E}_4\ ]\!]\\ &\quad\Rightarrow\ \ \textit{every}\ \mathbf{TE}_{\text{orion}}\ [\![\ \text{set\_of}\ \mathrm{E}_2\ \mathrm{E}_3\ ]\!]\ \ \mathbf{TO}_{\text{orion}}\ [\![\ \omega\ ]\!]\ \ \mathbf{TE}_{\text{orion}}\ [\![\ \mathrm{E}_4\ ]\!]\end{aligned}\qquad\text{(orion.37)}$$

Specific relational operations between two collections or between a collection and a value are translated as follows.

$$\mathbf{TO}_{\text{orion}}\ [\![\ \neg{:}{=}\ ]\!]\ \Rightarrow\ \sim{=}\qquad\text{(orion.38)}$$

$$\mathbf{TO}_{\text{orion}}\ [\![\ \text{equal}\ ]\!]\ \Rightarrow\ =\qquad\text{(orion.39)}$$

$$\mathbf{TO}_{\text{orion}}\ [\![\ \text{string-equal}\ ]\!]\ \Rightarrow\ =\qquad\text{(orion.40)}$$

$$\mathbf{TO}_{\text{orion}}\ [\![\ \text{string=}\ ]\!]\ \Rightarrow\ =\qquad\text{(orion.41)}$$

$$\mathbf{TO}_{\text{orion}}\ [\![\ \text{is-equal}\ ]\!]\ \Rightarrow\ ==\qquad\text{(orion.42)}$$

C.2 OSQL Translation Rules

C.2.1 OSQL Abstract Syntax

$$
\begin{array}{lcl}
Q & ::= & Q \textit{ union all } Q \mid Q \textit{ union } Q \mid Q \textit{ order by } Ss \\
 & \mid & \textit{select } E \textit{ for each } Ds \textit{ where } E \\
 & \mid & \textit{select } E \textit{ for each } Ds \\
 & \mid & \textit{select } E \\
 & \mid & \textit{select distinct } E \textit{ for each } Ds \textit{ where } E \\
 & \mid & \textit{select distinct } E \textit{ for each } Ds \\
 & \mid & \textit{select distinct } E \\
 & & \\
Ss & ::= & S \mid S,\ Ss \\
 & & \\
S & ::= & E \textit{ asc} \mid E \textit{ desc} \\
 & & \\
Ds & ::= & D \mid D,\ Ds \\
 & & \\
D & ::= & I\ I \\
 & & \\
Es & ::= & E \mid E,\ Es \\
 & & \\
E & ::= & E \textit{ and } E \mid E \textit{ or } E \\
 & \mid & E\ \omega\ E \mid E\ \psi\ E \\
 & \mid & I(\ Es\ ) \mid I \mid (\ E\ ) \\
 & \mid & \{\ Es\ \} \mid [\ Es\ ] \mid [|\ Es\ |] \mid k \\
 & \mid & \textit{sum}(\ E\ ) \mid \textit{max}(\ E\ ) \mid \textit{min}(\ E\ ) \mid \textit{count}(\ E\ ) \\
 & \mid & \textit{occurs}(\ E,\ E\ ) \mid \textit{head}(\ E\ ) \mid \textit{tail}(\ E\ ) \\
 & & \\
\omega & ::= & =|<>|>|>=|<|<=| \textit{ in} \\
 & & \\
\psi & ::= & *\mid /\mid +\mid -
\end{array}
$$

Table C.1: OSQL Abstract Syntax.

C.2.2 OSQL Translation Rules

Translating Queries

A bag is often used to return the result of an OSQL query. However, it is possible eliminate duplicates in a bag. The operation *union all* represents additive union on bags and *union* behaves like set union. The former operation corresponds to bag union in object comprehensions. The latter operation requires duplicate elimination and this effect can

be achieved in object comprehensions by collecting the result in a set and then turning it into a bag.

$$\mathbf{TQ}_{osql}\ [\![\ Q_1\ \text{union all}\ Q_2\ ]\!]$$
$$\Rightarrow\ \mathbf{TQ}_{osql}\ [\![\ Q_1\ ]\!]\ \mathit{union}\ \mathbf{TQ}_{osql}\ [\![\ Q_2\ ]\!] \qquad \text{(osql.1)}$$

$$\mathbf{TQ}_{osql}\ [\![\ Q_1\ \text{union}\ Q_2\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ x \leftarrow \mathit{Set}[\ y \leftarrow \mathbf{TQ}_{osql}\ [\![\ Q_1\ ]\!]\ \mathit{union}\ \mathbf{TQ}_{osql}\ [\![\ Q_2\ ]\!]\ |\ y\ ]\ |\ x\ ] \qquad \text{(osql.2)}$$

A collection can be sorted to ascending or descending order according to the result returned by some method call. The resultant collection is naturally a list. The next rule shows how sorting to descending order involving a single key can be translated into object comprehensions. The collection to be sorted is first turned into a list using the function *tolist* before the recursive function *order* is applied on it. The recursive function divides the given list (*xs*) into two sublists: one containing the largest elements (*top(xs)*) and another containing the rest of the list (*rest(xs)*). The result returned by the recursive call on the rest of the list (*order(rest(xs))*) is then appended to the first sublist (*top(xs)*). The function *keys* projects the sort key from the given collection. $\mathbf{TT}_{osql}$ returns the type of the input collection Q and $\mathbf{TM}_{osql}$ returns the type of its elements.

$$\begin{aligned}
&\mathbf{TQ}_{osql}\ [\![\ Q\ \text{order by E desc}\ ]\!]\\
&\Rightarrow\ \mathit{let}\ \mathit{keys}(\ xs : \mathit{List\ of}\ \mathbf{TM}_{osql}\ [\![\ Q\ ]\!]\ )\ \mathit{be}\\
&\qquad \mathit{Set}[\ x \leftarrow xs\ |\ \mathbf{TE}_{osql}\ [\![\ x\ E\ ]\!]\ ]\\
&\quad \mathit{in}\\
&\quad \mathit{let}\ \mathit{top}(\ xs : \mathit{List\ of}\ \mathbf{TM}_{osql}\ [\![\ Q\ ]\!]\ )\ \mathit{be}\\
&\qquad \mathit{List}[\ x \leftarrow xs;\ \mathbf{TE}_{osql}\ [\![\ x\ E\ ]\!]\ >=\ \mathit{every\ keys}(\ xs\ )\ |\ x\ ]\\
&\quad \mathit{in}\\
&\quad \mathit{let}\ \mathit{rest}(\ xs : \mathit{List\ of}\ \mathbf{TM}_{osql}\ [\![\ Q\ ]\!]\ )\ \mathit{be}\\
&\qquad \mathit{List}[\ x \leftarrow xs;\ \mathit{not}(\ x = \mathit{some\ top}(\ xs\ )\ )\ |\ x\ ]\\
&\quad \mathit{in}\\
&\quad \mathit{let}\ \mathit{order}(\ xs : \mathit{List\ of}\ \mathbf{TM}_{osql}\ [\![\ Q\ ]\!]\ )\ \mathit{be}\\
&\qquad \mathit{top}(\ xs\ )\ \mathit{union}\ \mathit{List}[\ \mathit{rest}(\ xs\ ) \sim== \mathit{List}\{\};\ y \leftarrow \mathit{order}(\ \mathit{rest}(\ xs\ )\ )\ |\ y\ ]\\
&\quad \mathit{in}\\
&\quad \mathit{let}\ \mathit{tolist}(\ xs : \mathbf{TT}_{osql}\ [\![\ Q\ ]\!]\ )\ \mathit{be}\\
&\qquad \mathit{List}[\ x \leftarrow xs\ |\ x\ ]\\
&\quad \mathit{in}\ \ \mathit{order}(\ \mathit{tolist}(\ \mathbf{TQ}_{osql}\ [\![\ Q\ ]\!]\ )\ ) \qquad \text{(osql.3)}
\end{aligned}$$

The translation rule for multiple sort keys is essentially the same. Each sort key can be treated as one level of sorting. Each sublist generated by *top* is sorted using the next sort key. This can be dealt with using $\mathbf{TQ}_{osql}$ recursively as shown below.

$\mathbf{TQ}_{osql}$ ⟦ Q order by E_1desc, E_2desc ⟧

⇒ *let* *keys*(*xs* : *List of* $\mathbf{TM}_{osql}$ ⟦ Q ⟧) *be*

Set[*x* ← *xs* | $\mathbf{TE}_{osql}$ ⟦ x E ⟧]

in

let *top*(*xs* : *List of* $\mathbf{TM}_{osql}$ ⟦ Q ⟧) *be*

List[*x* ← *xs*; $\mathbf{TE}_{osql}$ ⟦ x E ⟧ >= *every keys*(*xs*) | *x*]

in

let *rest*(*xs* : *List of* $\mathbf{TM}_{osql}$ ⟦ Q ⟧) *be*

List[*x* ← *xs*; *not*(*x* = *some top*(*xs*)) | *x*]

in

let *order*(*xs* : *List of* $\mathbf{TM}_{osql}$ ⟦ Q ⟧) *be*

List[*tops as top*(*xs*); *x* ← $\mathbf{TQ}_{osql}$ ⟦ tops order by E_2 desc ⟧ | *x*]

union *List*[*rest*(*xs*) ~== *List*{}; *y* ← *order*(*rest*(*xs*)) | *y*]

in

let *tolist*(*xs* : $\mathbf{TT}_{osql}$ ⟦ Q ⟧) *be*

List[*x* ← *xs* | *x*]

in *order*(*tolist*($\mathbf{TQ}_{osql}$ ⟦ Q ⟧)) (osql.4)

Other basic query forms are translated to bag comprehensions as follows.

$\mathbf{TE}_{osql}$ ⟦ select E for each D where E_1 ⟧ ⇒ *Bag*[$\mathbf{TD}_{osql}$ ⟦ D ⟧; $\mathbf{TE}_{osql}$ ⟦ E_1 ⟧ | $\mathbf{TE}_{osql}$ ⟦ E ⟧] (osql.5)

$\mathbf{TE}_{osql}$ ⟦ select E for each D ⟧ ⇒ *Bag*[$\mathbf{TD}_{osql}$ ⟦ D ⟧ | $\mathbf{TE}_{osql}$ ⟦ E ⟧] (osql.6)

$\perp \mathbf{E}_{osql}$ ⟦ select ⊔ ⟧ ⇒ *Bag*[| $\mathbf{TE}_{osql}$ ⟦ E ⟧] (osql.7)

When the keyword *distinct* is specified duplicates in the resulting bag will be eliminated. This is achieved in object comprehensions by producing the result as a set and then turning it into a bag.

$\mathbf{TE}_{osql}$ ⟦ select distinct E for each D where E_1 ⟧

⇒ *Bag*[*x* ← *Set*[$\mathbf{TD}_{osql}$ ⟦ D ⟧; $\mathbf{TE}_{osql}$ ⟦ E_1 ⟧ | $\mathbf{TE}_{osql}$ ⟦ E ⟧] | *x*] (osql.8)

$\mathbf{TE}_{osql}$ ⟦ select distinct E for each D ⟧

⇒ *Bag*[*x* ← *Set*[$\mathbf{TD}_{osql}$ ⟦ D ⟧ | $\mathbf{TE}_{osql}$ ⟦ E ⟧] | *x*] (osql.9)

$\mathbf{TE}_{osql}$ ⟦ select distinct E ⟧

⇒ *Bag*[| $\mathbf{TE}_{osql}$ ⟦ E ⟧] (osql.10)

If the result of a query is a bag of collections, OSQL will combine elements in these collections and return instead a bag of objects. This *flattening* effect is captured in the next three rules using a generator over the individual result.

$$\mathbf{TE}_{osql}\ [\![\ \text{select E for each D where } E_1\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ \mathbf{TD}_{osql}\ [\![\ D\ ]\!];\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!];\ x \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ x\ ] \quad \text{(osql.11)}$$
$$\mathbf{TE}_{osql}\ [\![\ \text{select E for each D}\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ \mathbf{TD}_{osql}\ [\![\ D\ ]\!];\ x \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ x\ ] \quad \text{(osql.12)}$$
$$\mathbf{TE}_{osql}\ [\![\ \text{select E}\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ x \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ x\ ] \quad \text{(osql.13)}$$

If duplicates are to be eliminated, sets will be used for the intermediate results as shown below.

$$\mathbf{TE}_{osql}\ [\![\ \text{select distinct E for each D where } E_1\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ x \leftarrow \mathit{Set}[\ \mathbf{TD}_{osql}\ [\![\ D\ ]\!];\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!];\ y \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ y\ ]\ |\ x\ ] \quad \text{(osql.14)}$$
$$\mathbf{TE}_{osql}\ [\![\ \text{select distinct E for each D}\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ x \leftarrow \mathit{Set}[\ \mathbf{TD}_{osql}\ [\![\ D\ ]\!];\ y \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ y\ ]\ |\ x\ ] \quad \text{(osql.15)}$$
$$\mathbf{TE}_{osql}\ [\![\ \text{select distinct E}\ ]\!]$$
$$\Rightarrow\ \mathit{Bag}[\ x \leftarrow \mathit{Set}[\ y \leftarrow \mathbf{TE}_{osql}\ [\![\ E\ ]\!]\ |\ y\ ]\ |\ x\ ] \quad \text{(osql.16)}$$

Choosing among the four sets of translation rules presented above depends partly on the type of the resultant collection. This choice is not captured in the rules themselves but can be enforced as a condition of application.

Translating Domains

$$\mathbf{TD}_{osql}\ [\![\ D_1\ ,\ D_2\ ]\!]\ \Rightarrow\ \mathbf{TD}_{osql}\ [\![\ D_1\ ]\!];\ \mathbf{TD}_{osql}\ [\![\ D_2\ ]\!] \quad \text{(osql.17)}$$
$$\mathbf{TD}_{osql}\ [\![\ I_1\ I_2\ ]\!]\ \Rightarrow\ I_2 \leftarrow I_1 \quad \text{(osql.18)}$$

Translating Expressions

A sequence of expressions is translated as follows.

$$\mathbf{TE}_{osql}\ [\![\ E_1\ ,\ E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!]\ ,\ \mathbf{TE}_{osql}\ [\![\ E_2\ ]\!] \quad \text{(osql.19)}$$

Logical connectives and operations are dealt with as shown below.

$$\mathbf{TE}_{osql}\ [\![\ E_1 \text{ and } E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!];\ \mathbf{TE}_{osql}\ [\![\ E_2\ ]\!] \quad \text{(osql.20)}$$
$$\mathbf{TE}_{osql}\ [\![\ E_1 \text{ or } E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!]\ \mathit{or}\ \mathbf{TE}_{osql}\ [\![\ E_2\ ]\!] \quad \text{(osql.21)}$$
$$\mathbf{TE}_{osql}\ [\![\ E_1 \text{ in } E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!] = \mathit{some}\ \mathbf{TE}_{osql}\ [\![\ E_2\ ]\!] \quad \text{(osql.22)}$$
$$\mathbf{TE}_{osql}\ [\![\ E_1\ \phi\ E_2\ ]\!]\ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ E_1\ ]\!]\ \ \mathbf{TO}_{osql}\ [\![\ \phi\ ]\!]\ \ \mathbf{TE}_{osql}\ [\![\ E_2\ ]\!] \quad \text{(osql.23)}$$

Method calling in OSQL has a functional style. The first argument of a method call

is the receiving object and the rest are the "actual" arguments. This is captured by the next two rules below.

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}\ )\ ]\!] \ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ .\ I \qquad \text{(osql.24)}$$

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}_1\ ,\ \mathrm{E}_2\ )\ ]\!] \ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_1\ ]\!]\ .\ I\ (\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_2\ ]\!]\ ) \qquad \text{(osql.25)}$$

Implicit join is supported in method calls. If the first argument is a collection and the method is applicable to its elements, the method together with the "actual" arguments will be applied to the elements of this collection. This is captured by the next two rules where $\mathbf{TC}_{osql}$ returns the collection kind of a collection-valued expression.

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}\ )\ ]\!] \ \Rightarrow\ \mathbf{TC}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ [\ x\ \leftarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ \mid\ x.I\ ] \qquad \text{(osql.26)}$$

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}_1\ ,\ \mathrm{E}_2\ )\ ]\!] \ \Rightarrow\ \mathbf{TC}_{osql}\ [\![\ \mathrm{E}_1\ ]\!]\ [\ x\ \leftarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_1\ ]\!]\ \mid\ x.I(\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!]\ )\ ] \qquad \text{(osql.27)}$$

If in addition the method itself returns a collection, OSQL will combine elements of the collections returned from the successive calls of the method.

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}\ )\ ]\!] \ \Rightarrow\ \mathbf{TC}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ [\ x\ \leftarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!];\ y\ \leftarrow\ x.I\ \mid\ y\ ] \qquad \text{(osql.28)}$$

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{I}\ (\ \mathrm{E}_1\ ,\ \mathrm{E}_2\ )\ ]\!] \ \Rightarrow\ \mathbf{TC}_{osql}\ [\![\ \mathrm{E}_1\ ]\!]\ [\ x\ \leftarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_1\ ]\!];\ y\ \leftarrow\ x.I(\ \mathbf{TE}\ [\![\ \mathrm{E}_2\ ]\!]\ )\ \mid\ y\ ] \qquad \text{(osql.29)}$$

Different notations are used to represent different kinds of collection literals. Collection literals, brackets, constants, and identifiers are translated by the following rules.

$$\mathbf{TE}_{osql}\ [\![\ \{\ \mathrm{E}\ \}\ ]\!] \ \Rightarrow\ \mathit{Set}\{\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ \} \qquad \text{(osql.30)}$$

$$\mathbf{TE}_{osql}\ [\![\ [\ \mathrm{E}\ ]\ ]\!] \ \Rightarrow\ \mathit{Bag}\{\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ \} \qquad \text{(osql.31)}$$

$$\mathbf{TE}_{osql}\ [\![\ [|\ \mathrm{E}\ |]\ ]\!] \ \Rightarrow\ \mathit{List}\{\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ \} \qquad \text{(osql.32)}$$

$$\mathbf{TE}_{osql}\ [\![\ (\ \mathrm{E}\ )\ ]\!] \ \Rightarrow\ (\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]\ ) \qquad \text{(osql.33)}$$

$$\mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!] \ \Rightarrow\ E \qquad \text{(osql.34)}$$

Some examples of translating aggregate functions and collection kind specific operations are given below.

$$\mathbf{TE}_{osql}\ [\![\ \text{count E}\ ]\!] \ \Rightarrow\ \mathit{size}\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!] \qquad \text{(osql.35)}$$

$$\mathbf{TE}_{osql}\ [\![\ \text{occurs E}_1\ \text{E}_2\ ]\!] \ \Rightarrow\ \mathit{size}\ \mathit{Bag}[\ x\ \leftarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_1\ ]\!];\ x = \mathbf{TE}_{osql}\ [\![\ \mathrm{E}_2\ ]\!]\ \mid\ x\ ] \qquad \text{(osql.36)}$$

$$\mathbf{TE}_{osql}\ [\![\ \text{head E}\ ]\!] \ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!].[1] \qquad \text{(osql.37)}$$

$$\mathbf{TE}_{osql}\ [\![\ \text{tail E}\ ]\!] \ \Rightarrow\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!].[\mathit{size}\ \mathbf{TE}_{osql}\ [\![\ \mathrm{E}\ ]\!]] \qquad \text{(osql.38)}$$

C.3 O_2SQL Translation Rules

C.3.1 O_2SQL Abstract Syntax

$$
\begin{array}{rcl}
Es & ::= & \Lambda \mid E \mid E,\ Es \\
\\
E & ::= & E\ union\ E \mid E\ intersect\ E \mid E\ minus\ E \\
& \mid & select\ E\ from\ Ds\ where\ E \\
& \mid & select\ E\ from\ Ds \\
& \mid & select\ distinct\ E\ from\ Ds\ where\ E \\
& \mid & sort\ all\ I\ in\ E\ on\ Es \\
& \mid & flatten\ E \mid listtoset\ E \mid magic(\ E\ ) \\
\\
& \mid & E\ and\ E \mid E\ or\ E \mid not(\ E\ ) \\
\\
& \mid & for\ all\ I\ in\ E:\ (\ E\ ) \mid there\ exists\ I\ in\ E:\ (\ E\ ) \\
& \mid & E\ \omega\ E \mid E\ \psi\ E \\
\\
& \mid & E.E \mid I(\ Es\ ) \mid I \mid (\ E\ ) \\
& \mid & set(\ Es\ ) \mid list(\ Es\ ) \mid list(\ E..E\ ) \mid k \\
\\
& \mid & concat(\ E,\ E\ ) \mid sublist(\ E,\ E,\ E\ ) \\
& \mid & head(\ E,\ E\ ) \mid tail(\ E,\ E\ ) \\
\\
& \mid & ith(\ E,\ E\ ) \mid first(\ E\ ) \mid last(\ E\ ) \mid element(\ E\ ) \\
& \mid & sum(\ E\ ) \mid count(\ E\ ) \mid avg(\ E\ ) \mid min(\ E\ ) \mid max(\ E\ ) \\
\\
Ds & ::= & D \mid D,\ Ds \\
\\
D & ::= & I\ in\ E \\
\\
\omega & ::= & =|<>|>|>=|<|<=|\ in \\
\\
\psi & ::= & * \mid / \mid + \mid - \mid mod \mid div
\end{array}
$$

Table C.2: O_2SQL Abstract Syntax.

C.3.2 O_2SQL Translation Rules

O_2SQL is functional in the sense that it allows free composition of constructs. A query is just an expression, therefore the translation begins with the function $\mathbf{TE}_{o_2sql}$.

Translating Expressions

A sequence of expressions is translated as follows.

$$\mathbf{TE}_{o_2sql}\ [\![\ E_1\ ,\ E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ ,\ \mathbf{TE}_{o_2sql}\ [\![\ E_2\ ]\!] \qquad \text{(o2sql.1)}$$

Set operations are dealt with as in ORION.

$$\mathbf{TE}_{o_2sql}\ [\![\ E_1 \text{ union } E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ \textit{union}\ \mathbf{TE}_{o_2sql}\ [\![\ E_2\ ]\!] \qquad \text{(o2sql.2)}$$

$$\mathbf{TE}_{o_2sql}\ [\![\ E_1 \text{ intersect } E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ \textit{differ}$$

$$(\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ \textit{differ}\ \mathbf{TE}_{o_2sql}\ [\![\ E_2\ ]\!]\ ) \qquad \text{(o2sql.3)}$$

$$\mathbf{TE}_{o_2sql}\ [\![\ E_1 \text{ minus } E_2\ ]\!] \ \Rightarrow\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ \textit{differ}\ \mathbf{TE}_{o_2sql}\ [\![\ E_2\ ]\!] \qquad \text{(o2sql.4)}$$

The following rules deal with the basic query forms. $\mathbf{TC}_{o_2sql}$ maps the collection kind of a domain to the range collection kind of the result. If all the domains are of the same kind, the collection kind of the result will be of the same collection kind. Otherwise it will be a bag.

$$\mathbf{TE}_{o_2sql}\ [\![\ \text{select E from D where } E_1\ ]\!]$$

$$\Rightarrow\ \mathbf{TC}_{o_2sql}\ [\![\ D\ ]\!]\ [\ \mathbf{TD}_{o_2sql}\ [\![\ D\ ]\!];\ \mathbf{TE}_{o_2sql}\ [\![\ E_1\ ]\!]\ |\ \mathbf{TE}_{o_2sql}\ [\![\ E\ ]\!]\ ] \qquad \text{(o2sql.5)}$$

$$\mathbf{TE}_{o_2sql}\ [\![\ \text{select E from D}\ ]\!]$$

$$\Rightarrow\ \mathbf{TC}_{o_2sql}\ [\![\ D\ ]\!]\ [\ \mathbf{TD}_{o_2sql}\ [\![\ D\ ]\!]\ |\ \mathbf{TE}_{o_2sql}\ [\![\ E\ ]\!]\ ] \qquad \text{(o2sql.6)}$$

A set is represented as a bag without duplicates. The keywords *distinct* can be used to eliminate duplicat in the res t collection. The fir rule below ised when all the domains are lists and the second rule covers all other cases. The first rule uses a recursive function f which takes two lists: the first list xs represents a sorted list while the second list ys represents an unsorted list. Effectively, the elements in ys are scanned and appended to xs if they are not already in xs.

$$\mathbf{TE}_{osql}\ [\![\ \text{select distinct E from D where } E_1\ ]\!]$$

$$\Rightarrow\ \textit{let}\ f(\ xs : \textit{List of}\ \mathbf{TT}_{o_2sql}\ [\![\ E\ ]\!],\ ys : \textit{List of}\ \mathbf{TT}_{o_2sql}\ [\![\ E\ ]\!]\ )\ \textit{be}$$

$$\textit{List}[\ ys == \textit{List}\{\};\ x\ \leftarrow\ xs\ |\ x\ ]$$

$$\textit{union}$$

$$\textit{List}[\ ys.[1] = \textit{some}\ xs;\ z\ \leftarrow\ f(\ xs,\ ys\ \textit{differ}\ \textit{List}\{ys.[1]\})\ |\ z\ ]$$

$$\textit{union}$$

$$\textit{List}[\ ys.[1] \sim= \textit{some}\ xs;\ z\ \leftarrow\ f(\ xs\ \textit{union}\ \textit{List}\{ys.[1]\},\ ys\ \textit{differ}\ \textit{List}\{ys.[1]\})\ |\ z\ ]$$

$$\textit{in}\ f(\ \textit{List}\{\},\ \textit{List}[\ \mathbf{TD}\ [\![\ D\ ]\!];\ \mathbf{TE}\ [\![\ E_2\ ]\!]\ |\ \mathbf{TE}\ [\![\ E_1\ ]\!]\ ]\ ) \qquad \text{(o2sql.7)}$$

$$\begin{aligned}&\mathbf{TE}_{o_2sql}[\![\text{ select distinct E from D where } E_1]\!]\\ &\quad\Rightarrow\ \mathit{Bag}[\, x \leftarrow \mathit{Set}[\, \mathbf{TD}_{o_2sql}[\![D]\!];\ \mathbf{TE}_{o_2sql}[\![E_1]\!] \mid \mathbf{TE}_{o_2sql}[\![E]\!] \,] \mid x \,]\end{aligned} \tag{o2sql.8}$$

Sorting is supported in O_2SQL and can be translated as in OSQL, hence is not given here.

Given a collection of collections, *flatten* combines the collections and the class of the resultant collection will be the same as the original collection elements except that a set is returned when given a set or a bag of lists. The $\mathbf{TX}_{o_2sql}$ function extracts the collection kind from the type expression returned by $\mathbf{TM}_{o_2sql}$. The operation *listtoset* turns a list into a set and *magic* turns a set into a list.

$$\mathbf{TE}_{o_2sql}[\![\text{ flatten(E)}]\!] \Rightarrow \mathbf{TX}_{o_2sql}[\![\mathbf{TM}_{o_2sql}[\![E]\!]]\!][\, xs \leftarrow \mathbf{TE}_{o_2sql}[\![E]\!];\ x \leftarrow xs \mid x \,] \tag{o2sql.9}$$

$$\mathbf{TE}_{o_2sql}[\![\text{ listtoset(E)}]\!] \Rightarrow \mathit{Set}[\, x \leftarrow \mathbf{TE}_{o_2sql}[\![E]\!] \mid x \,] \tag{o2sql.10}$$

$$\mathbf{TE}_{o_2sql}[\![\text{ magic(E)}]\!] \Rightarrow \mathit{List}[\, x \leftarrow \mathbf{TE}_{o_2sql}[\![E]\!] \mid x \,] \tag{o2sql.11}$$

Logical connectives are translated as follow.

$$\mathbf{TE}_{o_2sql}[\![E_1 \text{ and } E_2]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![E_1]\!];\ \mathbf{TE}_{o_2sql}[\![E_2]\!] \tag{o2sql.12}$$

$$\mathbf{TE}_{o_2sql}[\![E_1 \text{ or } E_2]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![E_1]\!]\ \mathit{or}\ \mathbf{TE}_{o_2sql}[\![E_2]\!] \tag{o2sql.13}$$

$$\mathbf{TE}_{o_2sql}[\![\text{ not (E)}]\!] \Rightarrow \mathit{not}\ (\ \mathbf{TE}_{o_2sql}[\![E]\!]\ ) \tag{o2sql.14}$$

O_2SQL supports a more general form of quantifiers. To express the equivalent form in object comprehensions involves the use of a query function. The next two rules translate the universal and existential quantifiers.

$$\begin{aligned}&\mathbf{TE}_{o_2sql}[\![\text{ for all I in E : (} E_1 \text{)}]\!]\\ &\quad\Rightarrow\ \mathit{let}\ f(\, xs : \mathbf{TT}_{o_2sql}[\![E]\!]\,)\ \mathit{be}\\ &\qquad\quad \mathbf{TC}_{o_2sql}[\![E]\!][\, I \leftarrow xs \mid \mathbf{TE}_{o_2sql}[\![E_1]\!]\,]\\ &\qquad \mathit{in}\ \mathit{every}\ f(\, \mathbf{TE}_{o_2sql}[\![E]\!]\,) = \mathit{true}\end{aligned} \tag{o2sql.15}$$

$$\begin{aligned}&\mathbf{TE}_{o_2sql}[\![\text{ there exists I in E : (} E_1 \text{)}]\!]\\ &\quad\Rightarrow\ \mathit{let}\ f(\, xs : \mathbf{TT}_{o_2sql}[\![E]\!]\,)\ \mathit{be}\\ &\qquad\quad \mathbf{TC}_{o_2sql}[\![E]\!][\, I \leftarrow xs \mid \mathbf{TE}_{o_2sql}[\![E_1]\!]\,]\\ &\qquad \mathit{in}\ \mathit{some}\ f(\, \mathbf{TE}_{o_2sql}[\![E]\!]\,) = \mathit{true}\end{aligned} \tag{o2sql.16}$$

Relational and arithmetic operations are translated as follows.

$$\mathbf{TE}_{o_2sql}[\![E_1\ \phi\ E_2]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![E_1]\!]\ \mathbf{TO}_{o_2sql}[\![\, \phi]\!]\ \mathbf{TE}_{o_2sql}[\![E_2]\!] \tag{o2sql.17}$$

Method calls, identifiers, constants, and brackets are translated as shown below.

$$\mathbf{TE}_{o_2sql}[\![\ E_1\ .\ E_2\ ]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ .\ \mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!] \quad \text{(o2sql.18)}$$

$$\mathbf{TE}_{o_2sql}[\![\ I\ (\ E\ )\ ]\!] \Rightarrow I\ (\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ ) \quad \text{(o2sql.19)}$$

$$\mathbf{TE}_{o_2sql}[\![\ E\ ]\!] \Rightarrow E \quad \text{(o2sql.20)}$$

$$\mathbf{TE}_{o_2sql}[\![\ (\ E\ )\ ]\!] \Rightarrow (\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ ) \quad \text{(o2sql.21)}$$

Collection literals, list operations, choosing an element from a collection, and an example aggregate function are translated as follows.

$$\mathbf{TE}_{o_2sql}[\![\ \text{set}(\ E\ )\ ]\!] \Rightarrow \mathit{Set}\{\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ \} \quad \text{(o2sql.22)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{list}(\ E\ )\ ]\!] \Rightarrow \mathit{List}\{\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ \} \quad \text{(o2sql.23)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{list}(\ E_1..E_2\ )\ ]\!] \Rightarrow \mathit{List}\{\ \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]..\mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!]\ \} \quad \text{(o2sql.24)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{concat}(\ E_1\ ,\ E_2\ )\ ]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ \mathit{union}\ \mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!] \quad \text{(o2sql.25)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{sublist}(\ E_1\ ,\ E_2\ ,\ E_3\ )\ ]\!] \Rightarrow \mathit{List}[\ I \leftarrow \mathit{List}\{\ \mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!]..\mathbf{TE}_{o_2sql}[\![\ E_3\ ]\!]\ \} \mid \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!].[\ i\ ]\ ] \quad \text{(o2sql.26)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{head}(\ E_1\ ,\ E_2\ )\ ]\!] \Rightarrow \mathit{List}[\ I \leftarrow \mathit{List}\{\ 1..\mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!]\ \} \mid \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ .\ [\ i\ ]\ ] \quad \text{(o2sql.27)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{tail}(\ E_1\ ,\ E_2\ )\ ]\!] \Rightarrow \mathit{List}[\ I \leftarrow \mathit{List}\{\ E_2..(\mathit{size}\ \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!])\ \} \mid \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ .\ [\ i\ ]\ ] \quad \text{(o2sql.28)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{ith}(\ E_1\ ,\ E_2\ )\ ]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ .\ [\mathbf{TE}_{o_2sql}[\![\ E_2\ ]\!]] \quad \text{(o2sql.29)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{first}(\ E\ )\ ]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ .\ [1] \quad \text{(o2sql.30)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{last}(\ E\ )\ ]\!] \Rightarrow \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]\ .\ [\mathit{size}\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!]] \quad \text{(o2sql.31)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{element}(\ E\ )\ ]\!] \Rightarrow \mathit{List}[\ x \leftarrow \mathbf{TE}_{o_2sql}[\![\ E_1\ ]\!]\ \mid\ x\ ]\ .\ [\ 1\ ] \quad \text{(o2sql.32)}$$

$$\mathbf{TE}_{o_2sql}[\![\ \text{count}(\ E\ )\ ]\!] \Rightarrow \mathit{size}\ \mathbf{TE}_{o_2sql}[\![\ E\ ]\!] \quad \text{(o2sql.33)}$$

Translating Domains

$$\mathbf{TD}_{o_2sql}[\![\ D_1\ ,\ D_2\ ]\!] \Rightarrow \mathbf{TD}_{o_2sql}[\![\ D_1\ ]\!]\ ;\ \mathbf{TD}_{o_2sql}[\![\ D_2\ ]\!] \quad \text{(o2sql.34)}$$

$$\mathbf{TD}_{o_2sql}[\![\ I\ \text{in}\ E\ ]\!] \Rightarrow I \leftarrow \mathbf{TE}_{o_2sql}[\![\ E\ ]\!] \quad \text{(o2sql.35)}$$

Bibliography

[AA92] R. Alhajj and M.E. Arkun. Queries in Object-Oriented Database Systems. In *Proceedings of the International Conference on Information and Knowledge Management*, volume 752 of *Lecture Notes in Computer Science*, pages 36–52. Springer-Verlag, 1992.

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247. ACM Press, 1991.

[AB93] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Technical report, INRIA, France, January 1993.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 40–57. Elsevier, 1989.

[ABD+90] S. Abiteboul, P. Buneman, C. Delobel, R. Hull, P. Kanellakis, and V. Vianu. New Hope on Data Models and Types: Report of an NSF-INRIA Workshop. *ACM SIGMOD Record*, 19(4):41–48, December 1990.

[ABGvG89] S. Abiteboul, C. Beeri, M. Gyssens, and D. van Gucht. An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. In Abiteboul et al. [AFS89], pages 117–138.

[AD94] R. Agrawal and L.G. DeMichiel. Type Derivation Using the Projection Operation. *Information Systems*, 19(1):55–68, 1994.

[ADL91] R. Agrawal, L.G. DeMichiel, and B.G. Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 113–128. ACM Press, 1991.

[AFS89] S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[Alh92] R. Alhajj. *A Query Model and an Object Algebra for Object-Oriented Databases*. PhD thesis, Bilkent University, Turkey, February 1992.

[Alt89] Altaïr, France. *The O_2 Query Language User's Manual*, December 1989.

[Ban89] F. Bancilhon. Query Languages for Object-Oriented Database Systems: Analysis and a Proposal. In *Proceedings of the GI Conference on Database Systems for Office, Engineering, and Scientific Applications*, pages 1–18. Springer-Verlag, 1989.

[Bar93] R.S.M. Barros. Formal Specification of Relational Database Applications: A Method and an Example. Technical Report GE-93-02, University of Glasgow, U.K., September 1993.

[BB89] E.J. Borowski and J.M. Borwein. *Dictionary of Mathematics*. Collins, 1989.

[BBN91] V. Breazu-Tannen, P. Buneman, and S. Naqiv. Structural Recursion as a Query Language. In *Proceedings of the International Workshop on Database Programming Languages*, pages 9–19. Morgan Kaufmann, 1991.

[BCD90] F. Bancilhon, S. Cluet, and C. Delobel. The O_2 Query Language Syntax and Semantics. Technical Report 45-90, Altaïr, France, May 1990.

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene and G. Kiczales. Common Lisp Object System Specification X3JI3 Document 88-002R. *ACM SIGPLAN Notices*, 23(special issue), September 1988.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building An Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[Bee88] D. Beech. A Foundation for Evolution from Relational to Object Databases. In *Proceedings of the International Conference on Extending Database Technology*, volume 303 of *Lecture Notes in Computer Science*, pages 251–270. Springer-Verlag, 1988.

[Ber92] E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proceedings of the International Conference on Extending Database Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 136–151. Springer-Verlag, 1992.

[BH91] R.S.M. Barros and D.J. Harper. A Method for the Specification of Relational Database Applications. In *Proceedings of the Sixth Z User Meeting*, Workshops in Computing Series, pages 261–286. Springer-Verlag, 1991.

[BJNS94] M. Buchheit, M.A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between Queries to Object-Oriented Databases. *Information Systems*, 19(1):33–54, 1994.

[BK93] P.J. Barclay and J.B. Kennedy. Viewing Objects. In *Proceedings of the British National Conference on Databases*, volume 696 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, 1993.

[BM89] O. Boucelma and J.L. Maitre. Querying Complex-Object Databases: the LIFOO Functional Language. Technical report, Université de Provence, France, 1989.

[BMS80] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an Experimental Applicative Language. In *Proceedings of the 1st ACM Lisp Conference*, pages 136–143. ACM Press, 1980.

[BNPS92] E. Bertino, M. Nagri, G. Pelagatti, and L. Sbattella. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[Bra92] S.E. Bratsberg. Unified Class Evolution by Object-Oriented Views. In *11th International Conference on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, pages 423–439. Springer-Verlag, 1992.

[BTA90] J.A. Blakeley, C.W. Thompson, and A.M. Alashqur. OQL[X]: Extending a Programming Language X with a Query Capability. Technical Report 90-07-01, Texas Instruments Incorporated, U.S.A., November 1990.

[BZ87] T. Bloom and S.B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In *Proceedings of thc Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 441–451. ACM Press, October 1987.

[Car84] L. Cardelli. Semantics of Multiple Inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[CDLR89] S. Cluet, C. Delobel, C. Lécluse, and P. Richard. Reloop, An Algebra Based Query Language for an Object-Oriented Database System. Technical Report 36-89, Altaïr, France, October 1989.

[CDV88] M.J. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–422. ACM Press, 1988.

[Cha92] C. Chambers. Object-Oriented Multi-Methods in Cecil. In *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, 1992.

[CHT92a] D.K.C. Chan, D.J. Harper, and P.W. Trinder. A Reference Object-Oriented Data Model Specification. Technical Report DB-92-2, University of Glasgow, U.K., November 1992.

[CHT92b] D.K.C. Chan, D.J. Harper, and P.W. Trinder. Object-Oriented Query Languages: Data Model Issues, Survey, Comparison, and Analysis. Technical Report DB-92-1, University of Glasgow, U.K., November 1992.

[CHT93a] D.K.C. Chan, D.J. Harper, and P.W. Trinder. A Case Study of Object-Oriented Query Languages. In *Proceedings of the International Conference on Information Systems and Management of Data*, pages 63–86. Indian National Scientific Documentation Centre (INSDOC), 1993.

[CHT93b] D.K.C. Chan, D.J. Harper, and P.W. Trinder. An Object-Oriented Data Model Specification. In *Proceedings of the 5th International Conference on Computing and Information*, pages 453–457. IEEE Press, 1993.

[CK94] D.K.C. Chan and D.A. Kerr. Improving One's Views of Object-Oriented Databases. In *Proceedings of the Colloquium on Object-Orientation in Databases and Software Engineering*. Elsevier, 1994.

[Cod72] E.F. Codd. Relational Completeness of Database Sublanguages. In *Data Base Systems*. Prentice-Hall, 1972.

[CT93] D.K.C. Chan and P.W. Trinder. An Evaluation Framework for Object-Oriented Query Languages. Technical Report DB-93-3, University of Glasgow, U.K., April 1993.

[CT94a] D.K.C. Chan and P.W. Trinder. An Object-Oriented Data Model Supporting Multi-methods, Multiple Inheritance, and Static Type Checking: A Specification in Z. In *Proceedings of the 8th Z User Meeting*, Workshops in Computing Series, pages 297–315. Springer-Verlag, 1994.

[CT94b] D.K.C. Chan and P.W. Trinder. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Proceedings of the British National Conference on Databases*, volume 826 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 1994.

[CTW95] D.K.C. Chan, P.W. Trinder, and R.C. Welland. Evaluating Object-Oriented Query Languages. *The Computer Journal*, 38(2), February 1995.

[DAF86] DAFTG. Database Architecture Framework Task Group (DAFTG) of the ANSI/X3/SPARC Database System Study Group: Reference Model for DBMS Standardization. *ACM SIGMOD Records*, 15(1):19–58, 1986.

[Dat84] C.J. Date. Some Principles of Good Language Design. *ACM SIGMOD Record*, pages 1–7, January 1984.

[Dat87] C.J. Date. *A Guide to INGRES.* Addison-Wesley, 1987.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proceedings of the International Workshop on Database Programming Languages*, pages 80–102. Morgan Kaufmann, 1989.

[DBCRW92] S.W. Dietrich, M. Brown, E. Cortes-Rello, and S. Wunderlin. A Partitioner's Introduction to Database Performance Benchmarks and Measurements. *The Computer Journal*, 35(4):322–331, August 1992.

[DD91] K.C. Davis and L.M.L. Delcambre. A Denotational Approach to Object-Oriented Query Language Definition. In *Proceedings of the International Workshop on Specifications of Database Systems*, Workshops in Computing Series, pages 88–105. Springer-Verlag, 1991.

[DGJ92] S. Dar, N.H. Gehani, and H.V. Jagadish. CQL++: A SQL for the ODE Object-Oriented DBMS. In *Proceedings of the International Conference on Extending Database Tec..nology*, volume 580 of *Lecture Notes in Computer Science*, pages 201–216. Springer-Verlag, 1992.

[DH87] S.A. Demurjian and D.K. Hsiao. The Multi-lingual Database System. In *Proceedings of the IEEE Data Engineering Conference*, pages 44–51. IEEE Press, 1987.

[Dit91] K.R. Dittrich. Object-Oriented Database Systems: The Notion and the Issues. In *On Object-Oriented Database Systems*, pages 3–10. Springer-Verlag, 1991.

[DLR88] C. Delobel, C. Lécluse, and P. Richard. LOOQ: A Query Language for Object-Oriented Databases, Informal Presentation. Technical Report 22-88, Altaïr, France, October 1988.

[DT88] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[EN89] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[FAC+89] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W.Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. risch, M.C. Shan, and W.K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. ACM Press, 1989.

[GD87] G. Graefe and D. DeWitt. The EXODUS Optimizer Generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 160–172. ACM Press, 1987.

[GM93] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proceedings of the Principles of Database Systems*, pages 49–58. ACM Press, 1993.

[GOPT92] G. Ghelli, R. Orsini, A. Pereira Paz, and P.W. Trinder. Design of an Integrated Query and Manipulation Notation for Database Languages. Technical Report FIDE/92/41, University of Glasgow, U.K., 1992.

[GW87] R.A. Ganski and H.K.T. Wong. Optimization of Nested SQL Queries Revisited. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–33, 1987.

[Ham90] K. Hammond. Definitional List Comprehensions. Technical Report 90/R3, University of Glasgow, U.K., January 1990.

[HFLP89] L. Hass, J. Freytag, G. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–388. ACM Press, June 1989.

[HS88] R. Hull and J. Su. On the Expressive Power of Database Queries with Intermediate Types. *Proceedings of the Principles of Database Systems*, pages 39–51, 1988.

[HW90] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. Technical Report 89/R5, University of Glasgow, U.K., February 1990.

[HZ90] S. Heiler and S. Zdonik. Object Views: Extending the Vision. In *Proceedings of the IEEE Data Engineering Conference*, pages 86–93. IEEE Press, 1990.

[JK84] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[Kim89] W. Kim. Object-Oriented Databases: Definition and Research Directions,. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):327–341, September 1989.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.

[Kim92] W. Kim. On Unifying Relational and Object-Oriented Database Systems. In *European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1992.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402. ACM Press, 1992.

[Klu82] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *Journal of the Association of Computing Machinery*, 29(3):699–717, 1982.

[KMK90] K. Kato, K. Masuda, and Y. Kiyoki. A Comprehension-based Database Language and its Distributed Execution. In *Proceedings of Distributed Computer System*, 1990.

[Kul93] K.G. Kulkarni. Object-Orientation and the SQL-Standard. *Computer Standards & Interfaces*, 15:287–300, 1993.

[LK86] P. Lyngbaek and W. Kent. A Data Modelling Methodology for the Design and Implementation of Information Systems. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 6–17. IEEE Press, 1986.

[LW93] L. Libkin and L. Wong. Some Properties of Query Languages for Bags. In *Proceedings of the International Workshop on Database Programming Languages*, Workshops in Computing Series, pages 97–114. Springer-Verlag, 1993.

[Lyn91] P. Lyngbaek. OSQL: A Language for Object Databases. Technical Report HPL-DTD-91-4, Hewlett-Packard Company, U.S.A., January 1991.

[Man91] F. Manola. Object Data Language Facilities for Multimedia Data Types. Technical Report TR-0169-12-91-165, GTE Laboratories Incorporated, U.S.A., December 1991.

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The Napier88 Reference Manual. Technical Report PPRR-77-89, University of Glasgow & University of St. Andrews, U.K., 1989.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MG93] J. Murphy and J. Grimson. Formal Specification of a Persistent Object Management System. *Information and Software Technology*, 35(5):277–286, 1993.

[MH87] C. Minkowitz and P. Henderson. A Formal Description of Object-Oriented Programming using VDM. In *VDM'87 - A Formal Method at Work*, volume 252 of *Lecture Notes in Computer Science*, pages 237–259. Springer-Verlag, 1987.

[MHH91] W. Mugridge, J. Hamer, and J. Hosking. Multi-Methods in a Statically-Typed Programming Languages. In *European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 307–324. Springer-Verlag, 1991.

[Mit93] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database.* PhD thesis, Brown University, U.S.A., May 1993.

[MS89] M. Missikoff and M. Scholl. An Algorithm for Insertion into a Lattice: Application to Type Classification. In *Proceedings of the 3rd International Conference on Foundations of Data Organisation and Algorithms*, volume 367 of *Lecture Notes in Computer Science*, pages 64–82. Springer-Verlag, 1989.

[NH91] R.S. Nikhil and M.L. Heytens. List Comprehensions in AGNA, a Parallel Persistent Object System. In *Proceedings of the Conference on Functional Programming and Computer Architecture.* ACM Press, 1991.

[Nor92] M.C. Norrie. *A Collection Model for Data Management in Object-Oriented Systems.* PhD thesis, University of Glasgow, United Kingdom, 1992.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 46–57. ACM Press, 1989.

[Ont91a] Ontologic Inc., U.S.A. *ONTOS Developer's Guide*, 1991.

[Ont91b] Ontologic Inc., U.S.A. *ONTOS Reference Manual*, 1991.

[Ont91c] Ontologic Inc., U.S.A. *ONTOS SQL Guide*, 1991.

[Osb88] S.L. Osborn. Identity, Equality and Query Optimization. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 346–351. Springer-Verlag, 1988.

[PD89] P. Pistor and P. Dadam. The Advanced Information Management Prototype. In Abiteboul et al. [AFS89], pages 3–26.

[PG90] N.W. Paton and P.M.D. Gray. Optimising and Executing DAPLEX Queries using Prolog. *The Computer Journal*, 33(6):547–555, 1990.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages*, chapter 7, pages 127–138. Prentice-Hall, 1987.

[PMSL94] H. Pirahesh, B. Mitschang, N. Südkamp, and B. Lindsay. Composite-Object Views in Relational DBMS: An Implementation Perspective. *Information Systems*, 19(1):69–88, 1994.

[Pou89] A. Poulovassilis. *The Design and Implementation of FDL, a Functional Database Language.* PhD thesis, Birkbeck College, University of London, 1989.

[PS94] A. Poulovassilis and Carol Small. Investigation of Algebraic Query Optimisation for Database Programming Languages. In *Proceedings of the Conference on Very Large Data Bases*, 1994.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, 1991.

[RKB87] M. Roth, H. Korth, and D. Batory. SQL/NF: A Query Language for ¬NF Relational Databases. *Information Systems*, 12(1):99–114, 1987.

[SAB+89] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Veroust. VERSO: A Database Machine Based on Nested Relations. In Abiteboul et al. [AFS89], pages 27–49.

[SAD94] C.S. Santos, S. Abiteboul, and C. Delobel. Virtual Schemas and Bases. In *Proceedings of the International Conference on Extending Database Technology*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 1994.

[Ser87] Servio Logic Development Corporation, U.S.A. *Programming in OPAL, Version 1.3*, 1987.

[SH85] B. Surfin and J. Hughes. A Tutorial Introduction to Relational Algebra. Technical report, Oxford University Programming Research Group, July 1985.

[SJ92] G. Saake and R. Jungclaus. Views and Formal Implementation in a Three-level Schema Architecture for Dynamic Objects. In *Proceedings of the British National Conference on Databases*, volume 618 of *Lecture Notes in Computer Science*, pages 78–95. Springer-Verlag, 1992.

[SLT91] M. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 189–207. Springer-Verlag, 1991.

[SO90] D. Straube and T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Office Information Systems*, 8(4):387–430, 1990.

[SP91] C. Small and A. Poulovassilis. An Overview of PFL. In *Proceedings of the International Workshop on Database Programming Languages*, pages 89–103. Morgan Kaufmann, 1991.

[Spi88] J.M. Spivey. *Understanding Z - A Specification Language and its Formal Semantics.* Cambridge University Press, 1988.

[Spi92] J.M. Spivey. *The Z Notation - A Reference Manual.* Prentice-Hall, second edition, 1992.

[SS89] J.J. Shilling and P.F. Sweeney. Three Steps to Views: Extending the Object-Oriented Paradigm. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, pages 353–361, 1989.

[SS90] M.H. Scholl and H.J. Schek. A Relational Object Model. In *Proceedings of the International Conference on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 1990.

[Sto77] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[Str90] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, University of Alberta, Canada, 1990.

[SZ89] G.M. Shaw and S.B. Zdonik. Object-Oriented Queries: Equivalence and Optimization. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 264–278. Elsevier, 1989.

[SZ90] G.M. Shaw and S.B. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proceedings of the IEEE Data Engineering Conference*. IEEE Press, 1990.

[TCH90] P.W. Trinder, D.K.C. Chan, and D.J. Harper. Improving Comprehension Queries in PS-algol. In *Proceedings of the 1990 Glasgow Database Workshop*, pages 103–119, U.K., 1990. University of Glasgow.

[Tri89] P.W. Trinder. *A Functional Database*. D.Phil thesis, Oxford University, December 1989.

[Tri91] P.W. Trinder. Comprehensions: a Query Notation for DBPLs. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, pages 55–70. Morgan Kaufmann, 1991.

[Tur81] D.A. Turner. Recursion Equations as a Programming Language. In Darlington, Henderson, and Turner, editors, *Functional Programming and its Application*. Cambridge University Press, 1981.

[Tur85] D.A. Turner. Miranda: a Non-strict Functional Language with Polymorphic Types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architectures*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[TW89] P.W. Trinder and P.L. Wadler. Improving List Comprehensions Database Queries. In *Proceedings of the TENCON'89*, pages 186–192. IEEE Press, 1989.

[US92] R. Unland and G. Schlageter. Object-Oriented Database Systems: State of the Art and Research Problems. In K. Jeffery, editor, *Expert Database Systems*, chapter 5, pages 117–222. Academic Press, 1992.

[Van92] B. Vance. Towards an Object-Oriented Query Algebra. Technical Report CS/E 91-008, Oregon Graduate Institute, U.S.A., January 1992.

[VD90] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. Technical Report 987, University of Wisconsin - Madison, U.S.A., December 1990.

[Wol87] M. Wolczko. Semantics of Smalltalk-80. In *European Conference on Object-Oriented Programming*, volume 276 of *Lecture Notes in Computer Science*, pages 108–120. Springer-Verlag, 1987.

[WT91] D. Watt and P.W. Trinder. Towards a Theory of Bulk Types. Technical Report FIDE/91/26, University of Glasgow, U.K., July 1991.

[YO91] L. Yu and S.L. Osborn. An Evaluation Framework for Algebraic Object-Oriented Data Models. In *Proceedings of the IEEE Data Engineering Conference*, pages 670–677. IEEE Press, 1991.