A Processing Framework for Object Comprehensions

Daniel K.C. Chan and Philip W. Trinder

Department of Computing Science, University of Glasgow
Glasgow G12 8QQ, United Kingdom

In order to execute database queries efficiently, they are usually translated into an algebra: a small set of operations together with some equivalences. The operations are implemented very efficiently by the database engine, and the equivalences are used to optimise the query. Since query languages for object-oriented databases are more complex than their relational counterparts, the algebra and translation, or *processing framework*, is more elaborate. This paper describes the processing framework for object comprehensions, a powerful object-oriented query language. Both operations and equivalences of the algebra are defined, together with a translation into it. The framework is illustrated by translating and optimising an example object comprehension query.

Key words: object-oriented database, query language, query optimisation, collection type, comprehensions

1 Introduction

Novel applications, such as computer-aided design, require data models that support complex relationships and a rich set of types, including function or method types. Object-oriented data models supporting complex objects and operations have been developed to cope with these requirements. To manipulate the rich structures found in these data models a query language will require more constructs. Often this is resolved by extending a relational query language with ad hoc features. Object comprehensions were specifically designed as a query language for object-oriented databases [14] and shown to be at least as powerful as several other prominent query languages [10].

It is essential that a query can be converted into an efficient execution plan, and this typically entails translating it into a small set of primitive operations. The operations form an algebra because there are equivalences between expressions containing the operations. The equivalences are used to transform a naive expression of the query into a more efficient form. The primitive operations are also very efficiently implemented by the database engine.

Preprint submitted to Elsevier Preprint *27 March 1997*

For simple data models, like the relational model, the algebra and translation are straightforward. There are several reasons why it is harder to provide algebraic support for richer query languages, i.e. those with more data structures and computational power. The target algebra must be more complex to represent sophisticated computation over a larger set of types, and more equivalences are required over the larger set of operations. The translation is more complex as both source language and target language are more complex.

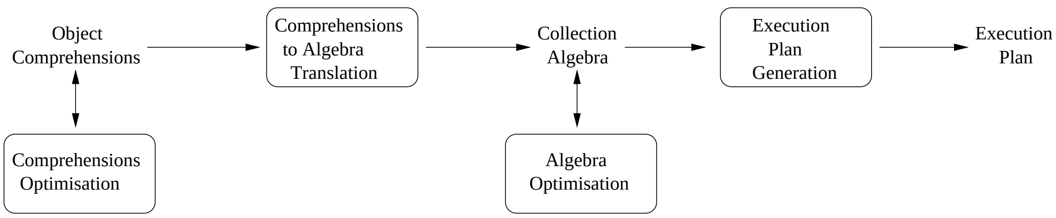

Fig. 1. A Complete Query Processing Framework

This paper describes an algebra-based processing framework for object comprehensions which constitutes an essential part of the complete query processing framework depicted in Figure 1. We formally specify a collection algebra, together with equivalence rules which can be used for optimisation. We describe some optimising transformations for object comprehensions and a translation of object comprehensions into the collection algebra. The translation is straightforward because both source and target languages are compositional, and uniform across multiple collection kinds. The framework is illustrated by translating a realistic object comprehension into the algebra and optimising the result. Automating the framework is the subject of ongoing work.

The paper is organised as follows. Section 2 provides a summary of the features of the reference data model on which object comprehensions and the collection algebra are based. A running example including essential features of the model is introduced. Section 3 describes object comprehensions. It begins with a brief review of the development of comprehensions as a query notation, follows with an examination of the new features in object comprehensions, then discusses the expressive power of object comprehensions and optimisation of some new features. Section 4 introduces the collection algebra, discussing the the design rationale, giving an informal semantics and a brief comparison with other algebras. Section 5 studies the translation from object comprehensions to the collection algebra. The translation enables automatic conversion between different kinds of collections. Section 6 lists some optimising equivalence rules of the collection algebra. Section 7 concludes and comments on future work.

2 Reference Data Model

```
Class Person isa Entity
methods
 name:→ string.

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 VisitingStaff isa Staff.

Class Department isa Entity
methods
 address:→ Address.

Class Course isa Entity
methods
 code:→ string,
 runBy:→ set of Department,
 prerequisites:→ set of Course,
 credits:→ integer,
 assessments:→ bag of integer.

Class Address isa Entity
methods
 street:→ string.

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

Fig. 2. Simplified Schema Definition

The reference data model supports classes each of which has a number of attributes and a set of methods. A class does *not* have an extent that contains all instances of the class. Hence the term class and type are freely interchangeable in the context of this paper – domains of integers, strings, and so forth are referred to as types. Classes can be related by the ISA relationship, with the possibility of multiple inheritance, to form a class graph. A subclass inherits all attributes and methods from its superclasses. It is possible to have overloaded methods – multiple methods with the same name – as long as their signatures are different. A class instance is identified by an object identifier which is unique among all instances. Communication and computation are performed by sending messages to objects. On receiving a message, the object chooses the most appropriate method to carry out the requested task. In other words, methods are dynamically bound. The reference data model supports three kinds of collections, namely set, bag, and list. Collections can be compared using their object identifiers or the elements they contain. A formal definition of the reference data model can be found in [13].

The running example database is a simplified university administration system that records information about students and staff members of a university, its academic departments and courses. Its simplified schema definition is given in Figure 2. For the sake of simplicity, only the relevant method signatures are given, attributes and method implementations are omitted. `Entity` is the root class in the class graph. 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. In the rest of the paper, a reflexive, asymmetric, and transitive relationship $\lll$ is used to capture paths in the class graph. For example, `Tutor` $\lll$ `Person` asserts that `Tutor` is a subclass of `Person` – an indirect subclass in this case.

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`. A student can have a principal supervisor, a second supervisor, and so forth. This relationship is modelled by the method `supervisedBy` as a list of staff members. Every staff member and student is 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` which records the number of credits (`credits`) each course gives and the percentage weights of assessments given in each course. The latter is represented as a bag of integers using the method `assessments`. During the discussion of object comprehensions and the collection algebra, the function *type* is used to return the (most specific) class of a given object.

Since classes have no extents, the schema defines five set collections: `Persons`, `Departments`, and `Courses` containing instances of their corresponding classes that are members of the university; whereas `StaffMembers` and `Students` containing instances of their corresponding classes that are members of the Science Faculty. Two more functions are used in the ensuing discussion. The function *size* returns the number of elements in a collection while *kind* distinguishes if a collection is a set, a bag, or a list (i.e. the class of a collection without its element class).

3 Comprehensions

3.1 Development

Object comprehensions are a generalisation and extension of *list comprehensions*, which are a language construct inspired by set abstraction in mathematics. List comprehensions are widely available in functional programming languages and are heavily used to manipulate collections of data. A full description can be found in [31].

List comprehensions were first used in functional database programming languages [1,32,36]. It is argued in [37] that list comprehensions are a good query notation for complex databases because of their brevity, clarity, expressive power, ease of optimisation, and smooth integration with programming languages. A study of collections in [40] shows that collection comprehensions can provide a uniform query notation over properly defined collections.

During the course of development of comprehensions, different dialects of comprehensions have found their way into various programming and query languages. For example, they have been applied to imperative languages such as Napier88 [27]. Not surprisingly, comprehensions are popular among systems that adopt the functional data model, such as [30,33]. The algebra underlying comprehensions has been the starting point for the theoretical studies of query languages [7,17]. Its role as an internal representation language for the ODMG-93 OQL [9] is investigated in [19].

Object comprehensions differ from other dialects of comprehensions in that they are designed to be a user query language for object-oriented databases [14] as opposed to just a vehicle for optimisation as in [30,19] or a notation for studying the theoretical foundation of query languages as in [7,17]. The design criteria adopted are thoroughly discussed in [15]. A comparison of object comprehensions with ONTOS SQL [29], OSQL [24], O_2SQL [3], ORION [21], EXCESS [8], OQL[C++] [6], CQL++ [16], and XSQL [20] can be found in [12].

3.2 Object Comprehensions

Q0. Return names of staff earning less than £20000.

```
set[ s ← StaffMembers; s.salary < 20000 | s.name ]
```

An example of an object comprehension query is given above. The result of evaluating this query is a new collection, precisely a set, computed from the existing collection `StaffMembers` of class `set of Staff`. The elements of the new collection are determined by repeatedly evaluating `s.name`, as controlled by the qualifier `s.salary` < `20000`. Since the result of `s.name` is of type `string` the elements in the resultant set are therefore of type `string`.

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 `s.salary` < `20000` above, ensuring that only staff members earning less than £20000 are used in computing the result. A generator of the form `I` ← `C` makes the variable `I` range over the elements of the collection `C`. An example of a generator was `s` ← `StaffMembers` above, making `s` range over the elements of the set `StaffMembers`. A local definition of the form `I as E`, introduces a name `I` for the value of the expression `E`. The abstract syntax of object comprehensions (without query function) is given in Appendix A. Some features of object comprehensions are demonstrated next using example queries.

Q1. Return all visiting staff members in the university.

```
set[ p ← Persons; p hasClass VisitingStaff | p ]
```

No collection in the database contains only objects of class `VisitingStaff` – recall that a class does not have an extent. Since `StaffMembers` contains only members in the Science Faculty, the only collection that contains all visiting staff members is `Persons`. This is the reason why the `Persons` collection is used in Q1. Since a collection can contain heterogeneous elements of 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 Q1, `hasClass` returns true if person object `p` is indeed of class `VisitingStaff`. This operation is essential for data models not supporting class extents as is the case with the reference data model.

Q2. Return all visiting staff in the university earning below £20000.

```
set[ p ← Persons;
     p hasClass VisitingStaff with p.salary < 20000 | 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 and with construct can be used. The role of with is similar to that of conjunction. The second condition (e.g. p.salary < 20000) is evaluated only if the first condition (e.g. p hasClass VisitingStaff) is true.

Q3. Return students whose major departments are in either Hillhead or Gibson Streets.

```
set[ s ← Students; a as s.major.address.street;
     a = “Hillhead Street” or a = “Gibson Street” | s ]
```

Local definitions simplify queries by providing names to expressions. In Q3, the path expression s.major.address.street would have been written twice if local definitions were not supported. A similar, but more restricted construct is found in OQL.

Q4. Return courses of one to three credits co-run by Computing Science.

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

Q5. Return students taking two or more of Johnson’s courses.

```
set[ l ← StaffMembers; l.name = “Johnson”; s ← Students;
     some s.takes = atleast 2 l.teaches | s ]
```

The quantification supported in object comprehensions is of a restricted form; however, query functions can be used to express complex quantification. Quantifiers, such as some, can appear on either side of an operator. In Q4, the second filter succeeds if the department object d is one of the elements in the set c.runBy. So some serves as the existential quantifier. In Q5, the last filter becomes true if there are at least two elements that are common between s.takes and l.teaches. The use of *numerical quantifiers* simplifies retrieval relying on occurrences. Object comprehensions support three numerical quantifiers: atleast, just, and atmost.

Q6. Return the first supervisor of all students. The result is used to assess the work load of supervisors and hence duplicates should be kept.

```
bag[ s ← Students | s.supervisedBy.[1] ]
```

This query is based on a set of `Student` objects and therefore the result is naturally a set of `Staff` objects. 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. Explicit specification of the resultant collection kind significantly simplifies the provision of algebraic support. OQL also allows such specification but with "full" type information.

3.3 Expressive Power

Object comprehensions are powerful because queries that can be expressed in ONTOS SQL [29], OSQL [24], O_2SQL [3], and ORION [21] over the reference data model can also be expressed in object comprehensions. Translation rules from these query languages to object comprehensions can be found in [10] and those for ONTOS SQL are presented in [11].

3.4 Transforming Object Comprehensions

It is possible to transform an object comprehension into another, more efficient comprehension. Optimisation developed for list comprehensions include most important relational optimisations, and are also applicable to object comprehensions. Details of which are not repeated here but can be found in [36], instead we examine here the optimisation of some new features that exist in object comprehensions: namely the `hasClass` predicate and the quantifiers.

A	*Aggregate Function*	O	*Object-valued Expression*
B	*Boolean-valued Expression*	Q	*Collection Algebra Operation*
C	*Collection-valued Expression*	T	*Class/Type Name*
D	*Generator*	X	*Qualifier*
E	*Expression*	Y	*Quantifier*
F	*Function*	α	*Logical Operation*
I	*Identifier*	ξ	*Collection Kind*
K	*Constant*	ψ	*Arithmetic Operation*
L	*Local Definition*	ω	*Relational Operation*
N	*Integer-valued Expression*	Λ	*Empty String*
nil	*Only Instance of Bottom Type*		

Fig. 3. Notations for Syntactic Categories

To facilitate the description of the transformation and translation rules a consistent notation is used, as listed in Figure 3, to represent the syntactic categories of object comprehensions and the collection algebra to be presented in the Section 4. A syntactic category is represented by an uppercase letter or a Greek letter which may be subscripted to represent different instances of the same category. An operation of the collection algebra may be sub-scripted or super-scripted by collection kind.

The `hasClass` predicate is defined in terms of $\lll$ by rule C1 below, and we give some simplifying transformations for it next.

$$O \;\texttt{hasClass}\; T \equiv \texttt{true} \quad \text{if } \; type(O) \lll T \tag{C1}$$
$$O \;\texttt{hasClass Entity} \equiv \texttt{true} \tag{C2}$$

$$(\; O \;\texttt{hasClass}\; T_1 \;) \;\texttt{and}\; (\; O \;\texttt{hasClass}\; T_2 \;)$$
$$\equiv O \;\texttt{hasClass}\; T \quad \text{if } \; \exists_1 \, T : ClassName \bullet T \lll T_1 \wedge T \lll T_2 \tag{C3}$$
$$\equiv \texttt{false} \quad \text{if } \; \neg\exists \, T : ClassName \bullet T \lll T_1 \wedge T \lll T_2 \tag{C4}$$

$$(\; O \;\texttt{hasClass}\; T_1 \;) \;\texttt{or}\; (\; O \;\texttt{hasClass}\; T_2 \;)$$
$$\equiv O \;\texttt{hasClass}\; T_1 \quad \text{if } \; T_2 \lll T_1 \tag{C5}$$

$$\texttt{not} \; (\; O \;\texttt{hasClass}\; T_1 \;) \;\texttt{and}\; (\; O \;\texttt{hasClass}\; T_2 \;)$$
$$\equiv \texttt{false} \quad \text{if } \; T_2 \lll T_1 \tag{C6}$$
$$\equiv O \;\texttt{hasClass}\; T_2 \quad \text{if } \; \neg\exists \, T : ClassName \bullet T \lll T_1 \wedge T \lll T_2 \tag{C7}$$

Rule C2 holds since all classes are subclasses of the root class `Entity`. A way to understand rules C3 and C4 is to study the paths starting from class T_1 and T_2 in the class graph. Rule C3 holds if extensions of the two paths meet at some class. Rule C4 holds when the extensions of the two paths do not meet at all. The same reasoning can be applied to prove the equivalence of rules C5, C6 and C7. A useful corollary to rule C3 is (O `hasClass` T_1) `and` (O `hasClass` T_2) $\equiv$ O `hasClass` T_2, if $T_2 \lll T_1$. Rule C1 to C7 can optimise queries by turning a predicate over objects into a predicate over the schema which needs to be evaluated only once at compile-time instead of once for each object at run-time. Similar, but less general transformation rules for semantic optimisation are proposed in [35].

Quantifiers can be combined freely around a relational operator. The following transformation rules can be used to simplify some usage patterns. Many equivalences depend on the size of a collection, and hence the rules can only be applied immediately prior to execution when the sizes are known.

$$\texttt{every} \; C \; \omega \; E \equiv \texttt{true} \quad \text{if } \; size(C) = 0 \tag{C8}$$

$$
\begin{aligned}
Y\ N\ C\ \omega\ E &\equiv \texttt{false} \ \text{ if } \ N < 0 && \text{(C9)}\\
\texttt{atmost}\ N\ C\ \omega\ E &\equiv \texttt{true} \quad \text{if } \ size(C) = 0 && \text{(C10)}\\
&\equiv \texttt{true} \quad \text{if } \ size(C) \leq N && \text{(C11)}\\
\texttt{just}\ N\ C\ \omega\ E &\equiv \texttt{true} \quad \text{if } \ size(C) = 0 \ \wedge \ N = 0 && \text{(C12)}\\
&\equiv \texttt{false} \ \text{ if } \ size(C) < N && \text{(C13)}\\
\texttt{atleast}\ N\ C\ \omega\ E &\equiv \texttt{true} \quad \text{if } \ N = 0 && \text{(C14)}\\
&\equiv \texttt{false} \ \text{ if } \ size(C) < N && \text{(C15)}\\
\texttt{some}\ C\ \omega\ E &\equiv \texttt{false} \ \text{ if } \ size(C) = 0 && \text{(C16)}
\end{aligned}
$$

If the collections are sets, the lack of duplicates makes more transformations possible:

$$
\begin{aligned}
\texttt{every}\ C = E &\equiv \texttt{false} \text{ if } \ size(C) > 1 && \text{(C17)}\\
\texttt{every}\ C_1 = \texttt{every}\ C_2 &\equiv \texttt{false} \text{ if } \ size(C_1) > 1 \vee size(C_2) > 1 && \text{(C18)}\\
\texttt{atmost}\ N\ C_1 = \texttt{some}\ C_2 &\equiv \texttt{true} \quad \text{if } \ size(C_2) < N && \text{(C19)}\\
\texttt{just}\ N\ C = E &\equiv \texttt{false} \text{ if } \ N > 1 && \text{(C20)}\\
\texttt{just}\ N\ C_1 = \texttt{some}\ C_2 &\equiv \texttt{false} \text{ if } \ size(C_2) < N && \text{(C21)}\\
\texttt{atleast}\ N\ C = E &\equiv \texttt{false} \text{ if } \ N > 1 && \text{(C22)}\\
\texttt{atleast}\ N\ C_1 = \texttt{some}\ C_2 &\equiv \texttt{false} \text{ if } \ size(C_2) < N && \text{(C23)}
\end{aligned}
$$

If an object-oriented database system explicitly records collection sizes, rule C8 to C23 can significantly improve the evaluation of quantifiers by "lifting" the predicate to the collection level hence avoiding any computation over the elements of the collection. If a system does not explicitly record collection sizes, the rules may still be useful as evaluating the size of a collection may very well be less expensive than actually performing computation over the elements.

4 Collection Algebra

4.1 Design Rationale

The *collection algebra* presented here is designed with the aim of being an expressive and uniform language. It must have enough expressive power to capture queries that can be expressed in object comprehensions. This implies the ability to express a wide variety of operations over complex data including various kinds of collections. Uniformity reduces the conceptual complexity of the language, simplifies optimisation, and renders the language more extensible. One way to achieve uniformity is to separate the concerns of collection kind from element type. In other words, collections are treated as containers independent of the elements contained in them. For instance, a relation should be seen as a set of tuples which can be operated on using a set algebra and a tuple algebra. Another means is to exploit as far as possible the regularity between operations for different kinds of collections. The collection algebra demonstrates how uniformity can be established from these two dimensions.

4.2 Algebra Operations

The collection algebra consists of a small number of operations. Some of the operations are parameterised with functional arguments. This treatment makes the operations more uniform as variations can be captured in the functional arguments. Unlike methods in the reference data model, these functions are known to the query processor and hence are amenable to reasoning and therefore optimisation. It should be noted that the collection algebra is not a minimal set of operations, some operations can be defined by others. For instance, *select* is introduced to capture well-known evaluation strategies even if it can be expressed in terms of *iter*. The algebra operations are described informally here, formal definitions can be found in [10].

$empty_\xi(\ E\ )$	$single_\xi(\ E\ )$	$union_\xi(\ C_1,\ C_2\ )$
$iter_\xi(\ F,\ C\ )$	$select_\xi(\ F,\ C\ )$	$reduce_\xi(\ E_0,\ F_1,\ F_{aggregate},\ C\ )$
$differ_\xi(\ C_1,\ C_2\ )$	$equal_\xi(\ C_1,\ C_2\ )$	$range_\xi(\ N_1,\ N_2\ )$
$index(\ C,\ N\ )$	$make^{\xi_2}_{\xi_1}(\ C\ )$	
$and(\ B_1,\ B_2\ )$	$being(\ O,\ T\ )$	$if(\ B_{condition},\ E_{true},\ E_{false}\ )$

Fig. 4. Collection Algebra Operations.

The $empty_\xi$ function takes an expression as argument and returns an empty collection of kind ξ. Whereas *single* returns a singleton collection instead. The $union_\xi$ function takes two collections of kind ξ and returns a collection of kind ξ containing all the elements of the argument collections. These three operations are called *collection constructors* as any collection can be constructed using one or a combination of these operations.

Simple functions can be applied to the elements of a collection by *iter* (called *ext* in [7]). The abstract syntax of the functions is given in Appendix B, and the semantics of $iter_\xi$ is given by rules iter.1 to iter.3.

$$iter_\xi(\ F,\ empty_\xi(\ nil\ )) \equiv empty_\xi(\ nil\ ) \tag{iter.1}$$
$$iter_\xi(\ F,\ single_\xi(\ E\ )) \equiv F\ E \tag{iter.2}$$
$$iter_\xi(\ F,\ union_\xi(\ C_1,\ C_2\ )) \equiv union_\xi(\ iter_\xi(\ F,\ C_1\ ),\ iter_\xi(\ F,\ C_2\ )) \tag{iter.3}$$

Select behaves in a similar fashion to *iter* but differs in the treatment to the elements in the argument collection. If applying the argument function to an element returns true a singleton collection containing that element is returned; otherwise, an empty collection is returned (rule select.2). Actually, *select* can be defined in terms of *iter* and *if* (rule select.4).

$$select_\xi(\ B,\ empty_\xi(\ nil\ )) \equiv empty_\xi(\ nil\ ) \tag{select.1}$$
$$select_\xi(\ B,\ single_\xi(\ E\ )) \equiv if(\ B\ E,\ single_\xi(\ E\ ),\ empty_\xi(\ E\ )) \tag{select.2}$$
$$select_\xi(\ B,\ union_\xi(\ C_1,\ C_2\ )) \equiv union_\xi(\ select_\xi(\ B,\ C_1\ ),\ select_\xi(\ B,\ C_2\ )) \tag{select.3}$$

$$select_\xi(\ B,\ C\ ) \equiv iter_\xi(\ \lambda i.if(\ B\ i,\ single_\xi(\ i\ ),\ empty_\xi(\ i\ )),\ C\ ) \tag{select.4}$$

Reduce is used to combine elements in a collection. If the argument collection C is empty, E_0 is returned (rule reduce.1). When the argument collection is a singleton collection, the result of F_1 on the singleton element E is returned (rule reduce.2). Otherwise, F_1 is applied to each element of C and the results are supplied pairwise to F_a which accumulates the results to give a single value (rule reduce.3). *Reduce* is a very powerful operation able to express many other operations including *iter* as shown in rule reduce.4.

$$reduce_\xi(\ E_0,\ F_1,\ F_a,\ empty_\xi(\ nil\ )) \equiv E_0 \tag{reduce.1}$$
$$reduce_\xi(\ E_0,\ F_1,\ F_a,\ single_\xi(\ E\ )) \equiv F_1\ E \tag{reduce.2}$$
$$reduce_\xi(\ E_0,\ F_1,\ F_a,\ union_\xi(\ C_1,\ C_2\ )) \equiv F_a\ reduce_\xi(\ E_0,\ F_1,\ F_a,\ C_1\ )\ reduce_\xi(\ E_0,\ F_1,\ F_a,\ C_2\ ) \tag{reduce.3}$$

$$reduce_\xi(\ empty_\xi(\ nil\ ),\ F,\ union_\xi,\ C\ ) \equiv iter_\xi(\ F,\ C\ ) \tag{reduce.4}$$

Differ removes elements in the second argument collection from the first argument collection. *Equal* returns true if the two argument collections contain the same elements. *Range* generates a collection containing integers within the range defined by the two argument expressions. An empty collection is returned if the first argument is less than the second. *Index* takes a list C and returns an element of the list at position N. *Make* converts the argument collection from its original collection kind ξ_1 to another collection kind ξ_2. Conversion from bag or set to list is non-deterministic as an arbitrary order will be assigned to the elements. *And* is non-commutative conjunction, used to confirm the class of an object before applying a method of that class. *Being* checks if the argument O is of class T or of a subclass of T. *If* returns either the second or the third argument depending on the boolean value of the first argument.

4.3 Related Work

Using the analysis reported in [7], the collection algebra is provably at least as powerful as the relational algebra and the nested relational algebras. The collection algebra can be seen as a pragmatic counterpart of the pioneering theoretical work on the application of *monads* [40,39]. Monads are a structure originating in category theory, but for query language purposes we view them as an algebra that relates *empty*, *single* and *iter* with a few simple laws.

The extensive analysis of query algebras carried out in [7] reveals that uniformity does not compromise expressive power at all. Interaction between different kinds of collections has been studied using *monoids* and *monoid homomorphism* [17], a slightly different algebra over query operations. In this work each collection kind is captured as a monoid and conversion between different collection kinds is captured as a homomorphism from one collection kind to another. The collection monoids used in the approach are essentially monads and homomorphism between collection monoids is effectively governed by the *Boom hierarchy* of data structures [25]. A collection kind can be converted into another collection kind lower in the hierarchy but not the other way round, e.g. a list can be converted into a set but not a set to a list. Part of the collection algebra, the monoid homomorphism described above, and the *monad morphism* described in [39] bear a striking resemblance even though they were developed independently.

FAD [2] is a functional database language designed for a parallel database machine that provides, among others, two collection operations *filter* and *pump*. *Filter* can be defined in terms of *iter* of the collection algebra by composing the argument functions with *single*. *Pump* behaves like *reduce* except that it is undefined for empty argument collections. FAD supports sets but not other kinds of collections. Machiavelli [28], a functional database programming language, includes three collection operations: *iter*, *select*, and *hom*. The first two operations can be defined in terms of *iter* and *select* of the collection algebra by composing their argument functions with *single*. *Hom* differs from *reduce* in the treatment of singleton collection. Instead of returning $F_1\ E$ as defined in rule reduce.2, *hom* returns $F_a\ (F_1\ E)\ E_0$. Machiavelli supports sets but not other kinds of collections. EQUAL [26] is an object algebra which allows the manipulation of sets. The *select* operation is the same as in the collection algebra. The *imagine* operation is the same as *iter* in Machiavelli. The *flatten* operation can be expressed as applying *iter* of the collection algebra with an *identity function*. The EXCESS algebra [38] is an object algebra supporting bags and arrays. The *set_collapse* and *set_apply* operations are the bag equivalent of *flatten* and *iter* in EQUAL. While *arr_apply* is the array version of EQUAL's *iter*. However, no operation is provided for conversion between collection kinds. Properties of bag algebras are studied in [18,23].

5 Translating Object Comprehensions

5.1 Translation Rules

The following rules enable an object comprehension query to be translated into a term in the collection algebra. The translation rules are presented in denotational style [34] using three translation functions: **TE** for translating expressions, **TO** for translating operations, and **TC** for extracting collection kinds.

The two infix collection operations – *union* and *differ* – can be translated as follows. The subscript to the algebraic operation is obtained using **TC** $[\![\ E_1\ ]\!]$.

$$\mathbf{TE}\ [\![\ E_1\ \texttt{union}\ E_2\ ]\!] \rightarrow union_{\mathbf{TC}\ [\![\ E_1\ ]\!]}(\ \mathbf{TE}\ [\![\ E_1\ ]\!],\ \mathbf{TE}\ [\![\ E_2\ ]\!]\ ) \qquad \text{(T1)}$$
$$\mathbf{TE}\ [\![\ E_1\ \texttt{differ}\ E_2\ ]\!] \rightarrow differ_{\mathbf{TC}\ [\![\ E_1\ ]\!]}(\ \mathbf{TE}\ [\![\ E_1\ ]\!],\ \mathbf{TE}\ [\![\ E_2\ ]\!]\ ) \qquad \text{(T2)}$$

Comprehensions are translated into the algebra using standard rules:

$$\mathbf{TE}\ [\![\ \xi[\ |\ E\ ]\ ]\!] \rightarrow single_{\xi}(\ \mathbf{TE}\ [\![\ E\ ]\!]\ ) \qquad \text{(T3)}$$
$$\mathbf{TE}\ [\![\ \xi[\ I \leftarrow E_1;\ Q\ |\ E\ ]\ ]\!] \rightarrow iter_{\xi}\ (\ \lambda I.\mathbf{TE}\ [\![\ \xi[\ Q\ |\ E\ ]\ ]\!],$$
$$make^{\xi}_{\mathbf{TC}\ [\![\ E_1\ ]\!]}(\ \mathbf{TE}\ [\![\ E_1\ ]\!]\ )) \qquad \text{(T4)}$$
$$\mathbf{TE}\ [\![\ \xi[\ I\ \texttt{as}\ E_1;\ Q\ |\ E\ ]\ ]\!] \rightarrow \mathbf{TE}\ [\![\ \xi[\ I \leftarrow \xi\{\ E_1\ \};\ Q\ |\ E\ ]\ ]\!] \qquad \text{(T5)}$$
$$\mathbf{TE}\ [\![\ \xi[\ E_1;\ Q\ |\ E\ ]\ ]\!] \rightarrow if(\ \mathbf{TE}\ [\![\ E_1\ ]\!],\ \ \mathbf{TE}\ [\![\ \xi[\ Q\ |\ E\ ]\ ]\!],$$
$$empty_{\xi}(\ nil\ )) \qquad \text{(T6)}$$

Logical connectives are mapped to their equivalents in the algebra.

$$\mathbf{TE} [\![E_1 \texttt{ and } E_2]\!] \rightarrow \mathbf{TE} [\![E_1]\!] \wedge \mathbf{TE} [\![E_2]\!] \quad \text{(T7)}$$
$$\mathbf{TE} [\![E_1 \texttt{ or } E_2]\!] \rightarrow \mathbf{TE} [\![E_1]\!] \vee \mathbf{TE} [\![E_2]\!] \quad \text{(T8)}$$
$$\mathbf{TE} [\![\texttt{not } E]\!] \rightarrow \neg \mathbf{TE} [\![E]\!] \quad \text{(T9)}$$

Class checking is performed using *being*. The non-commutative *and* in the second translation rule ensures that methods of E_3 are only invoked if the object is a member of the class.

$$\mathbf{TE} [\![E_1 \texttt{ hasClass } E_2]\!] \rightarrow$$
$$being(\mathbf{TE} [\![E_1]\!], \mathbf{TE} [\![E_2]\!]) \quad \text{(T10)}$$
$$\mathbf{TE} [\![E_1 \texttt{ hasClass } E_2 \texttt{ with } E_3]\!] \rightarrow$$
$$and(being(\mathbf{TE} [\![E_1]\!], \mathbf{TE} [\![E_2]\!]), \mathbf{TE} [\![E_3]\!]) \quad \text{(T11)}$$

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 of an operator, the binding order – universal then numerical followed by existential quantifier – determines the meaning of the filter. Note that the argument collection is converted to a bag before counting takes place in the rules for numerical quantifiers like T13. The translation of the other quantifiers can be found in [10].

$$\mathbf{TE} [\![\texttt{some } E_1 \ \omega \ E_2]\!]$$
$$\rightarrow reduce_{\mathbf{TC} [\![E_1]\!]}(false, \ \lambda i.\mathbf{TE} [\![i \ \omega \ E_2]\!], \ \vee, \ \mathbf{TE} [\![E_1]\!]) \quad \text{(T12)}$$
$$\mathbf{TE} [\![\texttt{atleast } E \ E_1 \ \omega \ E_2]\!]$$
$$\rightarrow reduce_{bag}(0, \ \lambda i.if \ (\mathbf{TE} [\![i \ \omega \ E_2]\!], \ 1, \ 0), \ +,$$
$$make^{bag}_{\mathbf{TC} [\![E_1]\!]}(\mathbf{TE} [\![E_1]\!])) >= \mathbf{TE} [\![E]\!] \quad \text{(T13)}$$

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 using **TO**. Method calls, identifiers, and constants are translated in the obvious way.

$$\mathbf{TE} [\![E_1 == E_2]\!] \rightarrow equal_{\mathbf{TC} [\![E_1]\!]}(\mathbf{TE} [\![E_1]\!], \ \mathbf{TE} [\![E_2]\!]) \quad \text{(T14)}$$
$$\mathbf{TE} [\![E_1 \sim== E_2]\!] \rightarrow \neg \ equal_{\mathbf{TC} [\![E_1]\!]}(\mathbf{TE} [\![E_1]\!], \ \mathbf{TE} [\![E_2]\!]) \quad \text{(T15)}$$

$$\mathbf{TE} [\![E_1 \ \omega \ E_2]\!] \rightarrow \mathbf{TE} [\![E_1]\!] \ \mathbf{TO} [\![\omega]\!] \ \mathbf{TE} [\![E_2]\!] \quad \text{(T16)}$$

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

$$\mathbf{TE}\ [\![\ \xi\{\}\ ]\!] \rightarrow empty_{\xi}(\ nil\ ) \tag{T17}$$
$$\mathbf{TE}\ [\![\ \xi\{\ E\ \}\ ]\!] \rightarrow single_{\xi}(\ \mathbf{TE}\ [\![\ E\ ]\!]\ ) \tag{T18}$$
$$\mathbf{TE}\ [\![\ \xi\{\ E_1\ ,\ E_2\ \}\ ]\!] \rightarrow union_{\xi}(\ \mathbf{TE}\ [\![\ \xi\{\ E_1\ \}\ ]\!],\ \mathbf{TE}\ [\![\ \xi\{\ E_2\ \}\ ]\!]\ ) \tag{T19}$$
$$\mathbf{TE}\ [\![\ \xi\{\ E_1..E_2\ \}\ ]\!] \rightarrow range_{\xi}(\ \mathbf{TE}\ [\![\ E_1\ ]\!],\ \ \mathbf{TE}\ [\![\ E_2\ ]\!]\ ) \tag{T20}$$
$$\mathbf{TE}\ [\![\ E_1\ ,\ E_2\ ]\!] \rightarrow \mathbf{TE}\ [\![\ E_1\ ]\!]\ ,\ \mathbf{TE}\ [\![\ E_2\ ]\!] \tag{T21}$$

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

$$\mathbf{TE}\ [\![\ E_1\ .\ [\ E_2\ ]\ ]\!] \rightarrow index(\ \mathbf{TE}\ [\![\ E_1\ ]\!],\ \mathbf{TE}\ [\![\ E_2\ ]\!]\ ) \tag{T22}$$

As an example of aggregate functions, the size of a collection can be computed using *reduce*. The accumulation function is addition (+). The argument collection is converted to a bag because addition is not idempotent and hence cannot be used on a set.

$$\mathbf{TE}\ [\![\ \texttt{size}\ E\ ]\!] \rightarrow reduce_{bag}(\ 0,\ \lambda i.1,\ +,\ make^{bag}_{\mathbf{TC}\ [\![\ E\ ]\!]}(\ \mathbf{TE}\ [\![\ E\ ]\!]\ )) \tag{T23}$$

The translation rules from object comprehensions to the collection algebra can be seen as a definition of object comprehensions in terms of the collection algebra. Note that *select* does not appear as a target operation in any of the translation rules. It only comes into being through the transformation of *iter* which is described in the Section 6. An example of the application of the translation rules is given in the next subsection.

5.2 Example Translation

The translation of example query Q4,

```
TE ⟦  set[ c ← Courses; d ← Departments;
           d.name = “Computing Science”; d = some c.runBy;
           1 <= c.credits; c.credits <= 3  |  c ]  ⟧
```

into the collection algebra is straightforward, and results in the following naive algebraic query. The following section introduces algebraic optimisations that greatly improve naive queries using techniques such as eliminating redundant operations, combining operations and substituting cheaper operations for more expensive operations where possible.

$$
\begin{array}{l}
iter_{set}(\\
\quad \lambda c.iter_{set}(\\
\quad\quad \lambda d.if(\ d.name = \text{“}Computing\ Science\text{”}, \\
\quad\quad\quad if(\ reduce_{set}(\ false,\ \lambda i.d = i,\ \vee,\ c.runBy\ ), \\
\quad\quad\quad\quad if(\ 1 <= c.credits, \\
\quad\quad\quad\quad\quad if(\ c.credits <= 3, \\
\quad\quad\quad\quad\quad\quad single_{set}(\ c\ ), \\
\quad\quad\quad\quad\quad empty_{set}(\ nil\ )), \\
\quad\quad\quad\quad empty_{set}(\ nil\ )), \\
\quad\quad\quad empty_{set}(\ nil\ )), \\
\quad\quad empty_{set}(\ nil\ )), \\
\quad make^{set}_{set}(\ Departments\ )), \\
make^{set}_{set}(\ Courses\ ))
\end{array}
$$

6 Collection Algebra Equivalences

6.1 Transformation Rules

This subsection gives equivalence rules between terms in the collection algebra. The rules aim to highlight properties of the “non-conventional” operations in the collection algebra. To use the rules in a real optimiser many other transformations are also necessary, some sources for additional transformations are [4,26,38].

Two notions of equivalence are used in the rules given below. For operations returning a collection, equivalence is defined over the elements of the collections as is supported by *equal* in the collection algebra. For operations returning a value that is not a collection, equivalence is defined as equality over either object identifiers or base values. The common restriction is made that no method has side-effects. Without this assumption the result of a method application may not be reproducible and hence the rule may not hold.

$$select_{\xi}(\ B,\ make^{\xi}_{\xi'}(\ C\ )) \equiv make^{\xi}_{\xi'}(\ select_{\xi'}(\ B,\ C\ )) \tag{O1}$$

$$select_{\xi}(\ B,\ differ_{\xi}(\ C_1,\ C_2\ )) \equiv differ_{\xi}(\ select_{\xi}(\ B,\ C_1\ ),\ C_2\ ) \tag{O2}$$

Rule O1 and O2 have to be applied carefully since it is not necessarily true that one expression always performs better the other. Which expression to use depends on how collections are implemented as well as if indexing can be employed for the selection predicate in question. In rule O1, it is true that pushing the conversion outside of the selection may not always improve performance but at the same time it is very unlikely to hurt performance. However, the transformation will pay off when the implementation of one collection kind is significantly different from that of the others, hence requiring significant computation for the conversion. While for rule O2 the trade off is between the evaluation cost of B and the size of C_1. In brief, selection should be pushed inside if appropriate indexes are available for evaluating B resulting in the difference being taken over a small collection.

$$select_\xi(\ B_1,\ select_\xi(\ B_2,\ C\ )) \equiv select_\xi(\ B_1\ \wedge\ B_2,\ C\ ) \tag{O3}$$
$$select_\xi(\ B,\ iter_\xi(\ F,\ C\ )) \equiv select_\xi(\ B \circ F,\ C\ ) \tag{O4}$$
$$select_\xi(\ B_1\ \wedge\ B_2,\ C\ ) \equiv if(\ B_1,\ select_\xi(\ B_2,\ C\ ),\ empty_\xi(\ nil\ )) \tag{O5}$$

Rule O3 and O4 simplify an expression from having two scans over a collection to only one scan over the same collection. Rule O5 extracts a constant predicate B_1 out of a selection; hence can potentially avoid having any computation over the given collection.

$$iter_\xi(\ F_1,\ iter_\xi(\ F_2,\ C\ )) \equiv iter_\xi((\ iter_\xi\ F_1\ ) \circ F_2,\ C\ ) \tag{O6}$$
$$iter_\xi(\ F,\ select_\xi(\ B,\ C\ )) \equiv iter_\xi(\ \lambda i.if(\ B\ i,\ F\ i,\ empty_\xi(\ i\ )),\ C\ ) \tag{O7}$$
$$iter_\xi(\ if(\ B,\ E,\ empty_\xi(\ nil\ )),\ C\ ) \equiv if(\ B,\ iter_\xi(\ E,\ C\ ),\ empty_\xi(\ nil\ )) \tag{O8}$$

Rule O6 to O8 are often used in optimising nested scans. Rule O6, used together with the definition of *iter* (rule iter.2), can simplifies scanning a collection twice to just once. Similarly, rule O7 merges a scan after a selection into just one scan. Rule O8 extracts predicates that are independent of the values of the collection elements.

$$iter_{set}(\ iter_{set}(\ F,\ C_2\ ),\ C_1\ ) \equiv iter_{set}(\ iter_{set}(\ F,\ C_1\ ),\ C_2\ ) \tag{O9}$$
$$iter_{bag}(\ iter_{bag}(\ F,\ C_2\ ),\ C_1\ ) \equiv iter_{bag}(\ iter_{bag}(\ F,\ C_1\ ),\ C_2\ ) \tag{O10}$$

Rule O9 and O10 assert that consecutive scans over sets and bags can be swapped. It is not true with lists because swapping two consecutive scans will affect the order of the elements in the resultant list.

$$index(\ union_{list}(\ C_1,\ C_2\ ),\ N\ ) \equiv if(\ size(\ C_1\ ) >= N, index(\ C_1,\ N\ ),\ index(\ C_2,\ N - size(\ C_1\ ))) \quad (O11)$$

$$index(\ range_{list}(\ N_1,\ N_2\ ),\ N\ ) \equiv N_1 + N - 1 \quad (O12)$$

$$if(\ B_1,\ if(\ B_2,\ E,\ E_1\ ),\ E_1\ ) \equiv if(\ B_1\ \wedge\ B_2,\ E,\ E_1\ ) \quad (O13)$$

$$make_{\xi}^{\xi}(\ C\ ) \equiv C \quad (O14)$$

The laws governing *index*, *if* and *make* given above are straightforward. An example of the application of the collection algebra optimisations is given in the next subsection.

6.2 Example Optimisation

The collection algebra query obtained in the Section 5 can be optimised using identities in the algebra. For brevity only the laws, and motivation are given below,

$\rightarrow$ *Eliminate redundant makes using O14*
$\rightarrow$ *Reorder iters using O9*
$\rightarrow$ *Promote filter using O8*
$\rightarrow$ *Flatten nested ifs using O13*
$\rightarrow$ *Convert iter to select using select.4*

$$\begin{array}{l}
iter_{set}(\\
\quad \lambda d.if(\ d.name = \text{``}Computing\ Science\text{''}, \\
\quad\quad select_{set}(\\
\quad\quad\quad \lambda c.\ reduce_{set}(\ false,\ \lambda i.d = i,\ \vee,\ c.runBy\ ) \\
\quad\quad\quad\quad \wedge\ 1 <= c.credits \\
\quad\quad\quad\quad \wedge\ c.credits <= 3, \\
\quad\quad Courses\ ), \\
\quad empty_{set}(\ nil\ )), \\
Departments\ )
\end{array}$$

7 Conclusions

7.1 Summary

The research reported in this paper began with identifying the functional requirements of object-oriented query languages. The 24 requirements presented in [15] 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 reported in this paper are an example.

Object comprehensions are both powerful and easily optimised. They are powerful because multiple collection classes can be dealt with, and queries expressible in other prominent query languages including the relational algebra and nested relational algebras can be expressed. They are easily optimised, first using rules to transform comprehensions, and then using equivalences in the collection algebra.

The collection algebra is regular and retains the power and ease of optimisation of comprehensions. The algebra is powerful because all object comprehension queries can be translated into it. It is regular because different collections can be manipulated in similar fashion using a small group of operations. It is optimisable because transformation rules are available and the regularity of the operations reduces significantly the optimisation search space.

The translation of object comprehensions into the collection algebra is simple, reflecting the uniformity of the source and target languages. The uniformity is principally due to the compositional nature of the two languages. In brief, the research described in this paper represents a step towards a better understanding of the processing framework required by object-oriented query languages.

7.2 Future Work

One pressing issue concerning object comprehensions is the support of arrays. A few proposals have been put forward for array query languages. The OQL proposal in [9] treats one-dimensional arrays as lists and multi-dimensional arrays as lists of lists. The monoid comprehensions suggested in [17] are problematic because different monoids are connected via implicit conversion which produces the undesirable effects of changing some constants in an expression, for instance all the 0's into 1's. A recent prototype reported in [22] incorporated arrays as functions instead of as collections. The integration of arrays into object comprehensions is being investigated along the direction proposed in [5]. The result of the integration would allow array operations to be expressed as naturally as for other collection classes. More importantly, it would overcome the undesirable effects and irregularity found in existing proposals.

Acknowledgement

We would like to express our gratitude to Ray Welland, Malcolm Atkinson, and David Watt of Glasgow University, David Harper of Robert Gordon University, and Norman Paton of University of Manchester for their inputs. We should also thank Catriel Beeri, Paula Ta-Shma, and David Michaeli of the Hebrew University of Jerusalem for the many inspiring discussions that indirectly influenced this piece of work.

A Abstract Syntax of Object Comprehensions

$$
\begin{array}{lcl}
Es & ::= & E \mid E\ \texttt{,}\ Es \\
E & ::= & E\ \texttt{union}\ E \mid E\ \texttt{differ}\ E \\
 & \mid & \xi\ \texttt{[}\ Xs\ \texttt{|}\ E\ \texttt{]} \\
 & \mid & E\ \texttt{and}\ E \mid E\ \texttt{or}\ E \mid \texttt{not}\ E \\
 & \mid & E\ \texttt{hasClass}\ E \mid E\ \texttt{hasClass}\ E\ \texttt{with}\ E \\
 & \mid & Y\ E\ \omega\ Y\ E \mid E\ \psi\ E \\
 & \mid & E\ \texttt{.}\ E \mid I\texttt{(}\ Es\ \texttt{)} \mid I \\
 & \mid & K \mid \xi\{\ Es\ \} \mid \xi\{\ E \ldots E\ \} \\
 & \mid & E\ \texttt{.}\ \texttt{[}\ E\ \texttt{]} \mid A\ E \mid \texttt{(}\ E\ \texttt{)} \\
Xs & ::= & \Lambda \mid X \mid X\ \texttt{;}\ Xs \\
X & ::= & D \mid L \mid E \\
D & ::= & I \leftarrow E \\
L & ::= & I\ \texttt{as}\ E \\
Y & ::= & \Lambda \mid \texttt{some} \mid \texttt{atleast}\ E \mid \texttt{just}\ E \mid \texttt{atmost}\ E \mid \texttt{every} \\
A & ::= & \texttt{size} \\
\xi & ::= & \texttt{set} \mid \texttt{bag} \mid \texttt{list} \\
\omega & ::= & \texttt{=} \mid \sim\texttt{=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \sim\texttt{==} \\
\psi & ::= & \texttt{*} \mid \texttt{/} \mid \texttt{+} \mid \texttt{-}
\end{array}
$$

B Abstract Syntax of Function Arguments

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

References

[1] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures - Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.

[2] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Powerful and Simple Database Language. In *Proceedings of the International Conference on Very Large Data Bases*, pages 97–105. Morgan Kaufmann, 1987.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building An Object-Oriented Database System - The Story of O_2*. Morgan Kaufmann, 1992.

[4] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *Proceedings of the International Conference on Database Theory*, volume 470 of *Lecture Notes in Computer Science*, pages 72–88. Springer-Verlag, 1990.

[5] C. Berri and D.K.C. Chan. Bound Arrays: a Bulk Type Perspective. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, October 1995.

[6] 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.

[7] P. Buneman, S. Naqiv, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[8] 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.

[9] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[10] D.K.C. Chan. *Object-Oriented Query Language Design and Processing*. PhD thesis, Department of Computing Science, University of Glasgow, U.K., September 1994. Available as Technical Report TR-1995-2.

[11] D.K.C. Chan. Translating Queries to Object Comprehensions. In *Industrial and Poster Paper Proceedings of the 14th International Conference on Object-Oriented & Entity Relationship Modelling*, pages 97–99. Queensland University of Technology Press, 1995.

[12] D.K.C. Chan and P.W. Trinder. An Evaluation Framework for Object-Oriented Query Languages. Technical Report DB-93-3, Department of Computing Science, University of Glasgow, U.K., April 1993.

[13] 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 Workshop*, Workshops in Computing Series, pages 297–315. Springer-Verlag, 1994.

[14] 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.

[15] D.K.C. Chan, P.W. Trinder, and R.C. Welland. Evaluating Object-Oriented Query Languages. *The Computer Journal*, 37(10):858–872, 1994.

[16] 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 Technology*, volume 580 of *Lecture Notes in Computer Science*, pages 201–216. Springer-Verlag, 1992.

[17] L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–58. ACM Press, 1995.

[18] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–58. ACM Press, 1993.

[19] T. Grust and M.H. Scholl. Translating OQL into Monoid Comprehensions - Stuck with Nested Loops? Technical Report 03/1996, Department of Mathematics and Computer Science, University of Konstanz, Germany, March 1996.

[20] 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.

[21] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, 1990.

[22] L. Libkin, R. Machlin, and L. Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 228–239. ACM Press, 1996.

[23] L. Libkin and L. Wong. Some Properties of Query Languages for Bags. In *Proceedings of the International Workshop on Database Programming Languages*, pages 97–114. Morgan Kaufmann, 1993.

[24] P. Lyngbaek. OSQL: A Language for Object Databases. Technical Report HPL-DTD-91-4, Hewlett-Packard Company, U.S.A., January 1991.

[25] L. Meertens. Algorithms - Towards Programming as a Mathematical Activity. In *Proceedings of CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[26] G. Mitchell. *Extensible Query Processing in an Object-Oriented Database*. PhD thesis, Brown University, U.S.A., 1993.

[27] 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.

[28] 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.

[29] Ontologic Inc., U.S.A. *ONTOS SQL Guide*, 1991.

[30] N.W. Paton and P.M.D. Gray. Optimising and Executing DAPLEX Queries using Prolog. *The Computer Journal*, 33(6):547–555, 1990.

[31] S. Peyton-Jones. *The Implementation of Functional Programming Languages*, chapter 7, pages 127–138. Prentice-Hall, 1987.

[32] A. Poulovassilis and P. King. Extending the Functional Data Model to Computational Completeness. In *Proceedings of the International Conference on Extending Database Technology*, volume 416 of *Lecture Notes in Computer Science*, pages 75–91. Springer-Verlag, 1990.

[33] 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.

[34] J.E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[35] 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.

[36] P.W. Trinder. *A Functional Database*. D.Phil thesis, Computing Laboratory, Oxford University, U.K., 1989.

[37] P.W. Trinder. Comprehensions: a Query Notation for DBPLs. In *Proceedings of the International Workshop on Database Programming Languages*, pages 55–70. Morgan Kaufmann, 1991.

[38] S.L. Vandenberg. *Algebras for Object-Oriented Query Languages*. PhD thesis, University of Wisconson - Madison, U.S.A., 1993.

[39] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[40] D. Watt and P.W. Trinder. Towards a Theory of Bulk Types. Technical Report FIDE/91/26, Department of Computing Science, University of Glasgow, U.K., July 1991.