Towards an Effective Calculus for Object Query Languages

Leonidas Fegaras David Maier

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
email: {*fegaras,maier*}*@cse.ogi.edu*

Abstract

We define a standard of *effectiveness* for a database calculus relative to a query language. Effectiveness judges suitability to serve as a processing framework for the query language, and comprises aspects of coverage, manipulability and efficient evaluation. We present the monoid calculus, and argue its effectiveness for object-oriented query languages, exemplified by OQL of ODMG-93. The monoid calculus readily captures such features as multiple collection types, aggregations, arbitrary composition of type constructors and nested query expressions. We also show how to extend the monoid calculus to deal with vectors and arrays in more expressive ways than current query languages do, and illustrate how it can handle identity and updates.

1 Introduction

A much-touted advantage of the relational data model is the existence of a formal calculus and algebra to model database queries. In practice, these formalisms fail to model many of the features present in commercial query languages (e.g., SQL): grouping, aggregation, duplicate values and sort orders, to name a few. Such features end up being handled in an ad hoc manner by query processors. The gap only widens as one moves to object-oriented query languages, such as OQL of ODMG-93 [9], which must deal with multiple bulk (collection) types, arbitrary nesting of type constructors, methods and embedded query expressions permitted whenever a collection may appear (not just in the where clause). (Such features are also making their way into relational query languages.) We seek a more effective formalism to model object query languages — one that represents the constructs actually found in those languages and that provides a suitable framework for the query processing component of a DBMS. We present a calculus, based on monoid comprehensions, that we believe meets those needs.

To appear in the ACM SIGMOD International Conference on Management of Data, San Jose, California, May 1995.

1.1 What is an Effective Calculus?

There is already a sizable body of proposals for database calculi and algebras. Why add another to the pile? In answering that question, it helps to separate the proposals by their purpose. Some have been used to study issues of computational complexity and relative expressive power of different language features [19, 1]. Other have been proposed to extend the relational model to handle one extension or another, such as duplicates [14], aggregate computation [24], nested structures [25, 12], temporal expressions [18], and path expressions [20]. Our interest, however, is finding a query model that provides an adequate basis for defining, translating and evaluating actual DBMS query languages. To that end, we describe what we mean by a database calculus C being *effective* for a query language L. Effectiveness has several aspects:

- *Coverage*: The calculus should have sufficient expressive power to represent all the language constructs of L. We want the calculus to serve as the initial internal representation of queries in L. To the extent that C does not cover L, the query processor will deal with the gap on an ad hoc basis. In addition to being an initial translation target, we would like to use the calculus to give a precise semantics to the query language. Having a precise semantics is a precondition if we ever want to demonstrate that certain query transformations do not alter the meaning of a query. Typical relational query formalisms fail to cover the features of current commercial query languages.
- *Manipulable*: Expressions in C should lend themselves to easy manipulation by programs for purposes of type checking, in-lining views, simplification, determining interactions with constraints and triggers, and translation into query plans.
- *Evaluable*: Here we mean not just that there is some means to evaluate an expression in C over a

database, but there is a wide space of alternative query plans accessible from a given calculus expression. Typically, we would imagine that a calculus expression could be translated into a companion logical algebra, that the algebra can be automatically transformed into equivalent expressions, and that those algebra expressions can be rendered into terms of specific physical operators or evaluation code. Note that a query formalism that expresses queries at a low level of abstraction can be a barrier to efficient evaluation. For example, a model that required nesting and combination orders of iterators to be explicitly listed could obscure the high-level intent of the query.

Given that the requirements above can be met, on what basis might one choose between alternative calculi for a language L? For the languages we are interested in, *uniformity* is a prime consideration. Object query languages typically deal with three or four different collections types (such as sets, bags, lists and arrays). A query model that exploits the similarities of these types and permits them to be operated upon in uniform ways can aid in the optimization process by cutting down on the number of different transformation rules or techniques needed to deal with the repertoire of operators. Another form of uniformity is how well the query model can integrate operations on database structures with general computation. Object databases provide powerful behavioral encapsulation with methods, but many prohibit method invocation in queries (or permit them but switch off optimization on queries with methods).

Our claim is that a query model based on monoid comprehensions meets the requirements of an effective calculus for real object query languages, and exhibits more uniformity than other current proposals.

1.2 Case Study: ODMG-93

To support our claims, it is useful to focus on a particular object query language. We have chosen to concentrate on the OQL language of the ODMG-93 standard proposal, which closely resembles the query language of the O2 OODBMS [15]. We chose OQL for two main reasons. One is that essentially all U.S. OODB companies have committed to supporting an OQL interface to their systems in the near term. The second is that it is a small language, hence easier to comprehend, but contains most of the language features that are showing up in other object query languages and proposed relational extensions, such as SQL3 [2]. Those features include multiple collection types, arbitrary nesting of type constructors, method invocation, complex object state, path expressions, object identity, subqueries at arbitrary points in query expressions, and a subtype hierarchy. If we are able to handle OQL, we believe our work will be widely applicable to other query languages.

1.3 Our Contribution

Our calculus is based on monoids, a general template for a data type, that can capture most collection and aggregate operators currently in use for relational and object-oriented databases. Monoid comprehensions — similar to set former notation but applicable to types other than sets — give us a uniform way to express queries that simultaneously deal with more than one collection type and also naturally compose in a way that mirrors the allowable query nesting in OQL. Further, comprehension expressions permit easy integration of functional subexpressions. We demonstrate coverage by showing how to map the major features of OQL into monoid calculus. In order to make expressions easier to operate upon, we use a formalism where type constructors are independent (similarly to the approach of Vandenberg and DeWitt [28]), rather than providing indivisible combinations of constructors, as in nested relational models. We give evidence of manipulability by exhibiting a simple normalization system for putting expressions into a canonical form that maximizes opportunities for pipelining. We believe the monoid calculus is amenable to efficient evaluation. We sketch a translation into a logical algebra and a companion paper [17] presents a framework for mapping to physical operators.

2 The Monoid Comprehension Calculus

Several recent proposals for object-oriented database languages, including OQL, support multiple collection types, such as sets, bags, lists, and arrays. These approaches define a language syntax, but they frequently fail to provide a concrete semantics. For example, is the join of a list with a set meaningful? If so, what is the result type of this join? More generally, what are the precise semantics of queries over multiple collection types? To answer these questions we need to form a theory that generalizes all collection types and their operations in a natural way. This theory must capture the differences between the collection types in such a way that no inconsistencies are introduced, and, more importantly, must abstract their similarities. By abstracting common semantic features, we derive a framework that treats collections uniformly in a simple and extensible language. The primary focus of such a framework is the bulk manipulation of collection types. Bulk operations are both the source of expressiveness and the basis for efficient execution.

Consider lists and sets. What are the semantic properties that make lists different from sets? Intuitively, one may exchange two elements of a set or insert a set element twice into a set without changing the set. These properties do not hold for lists. To formalize these observations, we need to see how sets and lists are constructed and then impose these properties on the set and list constructors. One way of constructing sets is to union together a number of singleton set elements, e.g., $\{1\} \cup \{2\} \cup \{3\}$ constructs the set $\{1, 2, 3\}$. Similarly, one way of constructing lists is to append singleton list elements, e.g., $[1]{+\!\!+}[2]{+\!\!+}[3]$ constructs the list $[1, 2, 3]$ (where ${+\!\!+}$ is the list append function). Both $\cup$ and ${+\!\!+}$ are associative operations, but only $\cup$ is commutative and idempotent (i.e., $\forall x : x \cup x = x$). It is the commutativity and idempotence properties of $\cup$ that make sets different from lists. As to their similarities, both the set and list constructors have an identity. The empty set $\{\}$ is the identity of $\cup$, while the empty list $[\,]$ is the identity of ${+\!\!+}$. Using terminology from abstract algebra, we say that both $(set, \{\}, \cup)$ and $(list, [\,], {+\!\!+})$ are *monoids*, and, in particular, $(set, \{\}, \cup)$ is a commutative and idempotent monoid.

Primitive types, such as integers and booleans, can be represented as monoids too, but with possibly a choice of monoid. For example, both $(int, 0, +)$ and $(int, 1, *)$ are integer monoids and both $(bool, \text{false}, \vee)$ and $(bool, \text{true}, \wedge)$ are boolean monoids. We call the monoids for collection types *collection monoids*, and the monoids for primitive types *primitive monoids*.

Each collection monoid has a unit function that takes an element of some type as input and constructs a singleton value of the collection type. For example, the list unit function takes an element a and constructs the singleton list $[a]$. Any list can be generated from the three list monoid primitives: the empty list $[\,]$, the list unit function, and the list append ${+\!\!+}$.

Since all types are represented as monoids, a query in our framework is a map from some monoids to a particular monoid. These maps are called *monoid homomorphisms*. For example, a monoid homomorphism from lists to sets in our framework is captured by an operation of the form

$$\text{hom}^{list \to set}(f)\, A$$

where A is a list and f is a function that takes an element x of A and returns the set $f(x)$. Basically, this monoid homomorphism performs the following computation:

```
result := {};
foreach $x$ in $A$ do
    result := result $\cup$ $f(x)$;
return result;
```

In other words, $\text{hom}^{list \to set}(f)\, A$ replaces $[\,]$ in A by $\{\}$, ${+\!\!+}$ by $\cup$, and the singleton list $[x]$ by $f(x)$. That is, if A is the list $[a_1, \ldots, a_n]$, which is generated by $[a_1]{+\!\!+} \cdots {+\!\!+}[a_n]$, then the result is the set $f(a_1) \cup \cdots \cup f(a_n)$. A monoid homomorphism captures a divide-and-conquer computation since any list $A = [a_1, \ldots, a_n]$ can be divided into two lists A_1 and A_2 such that $A = A_1{+\!\!+}A_2$. In that case, the operation $\text{hom}^{list \to set}(f)\, A$ is equal to $(\text{hom}^{list \to set}(f)\, A_1) \cup (\text{hom}^{list \to set}(f)\, A_2)$.

Unfortunately, not all monoid homomorphisms are well-formed. For example, sets cannot be converted into lists as this would introduce nondeterminism. (However, sets can be converted into sorted lists.) This semantic restriction is purely syntactic in our framework (i.e., it depends on the static properties of the monoids involved in a homomorphism, such as the commutativity and the idempotence properties).

The monoid homomorphism is the only form of bulk manipulation of collection types supported in our algebra. But, as we will demonstrate, monoid homomorphisms are very expressive. In fact, a small subset of these functions, namely the monoid homomorphisms from sets to sets, captures precisely the nested relational algebra (since they are equivalent to the set extension operator $ext(f)$, which has been shown to capture the nested relational algebra [5]). But monoid homomorphisms go beyond that to capture operations over multiple collection types, such as the join of a list with a bag that returns a set, plus predicates and aggregates. For example, an existential predicate over a set is a monoid homomorphism from the set monoid to the monoid $(bool, \text{false}, \vee)$, while an aggregation, such as summing all elements of a list, is a monoid homomorphism from the list monoid to the monoid $(int, 0, +)$.

We will also define a new calculus for this algebra, called the *monoid comprehension calculus*, that captures operations involving multiple collection types in declarative form. Monoid comprehensions are defined in terms of monoid homomorphisms, but any monoid homomorphism can be expressed in terms of a monoid comprehension. Programs expressed in our calculus are far easier to understand and manipulate than the equivalent algebraic forms. In a way, monoid comprehensions resemble the tuple relational calculus, but here query variables may range over multiple collection types while the output of the comprehension may yet be of a different collection type.

For example, the following monoid comprehension

$$set\{\, (a, b) \mid a \leftarrow [1, 2, 3],\, b \leftarrow \{\!\{4, 5\}\!\} \,\}$$

joins the list $[1, 2, 3]$ with the bag $\{\!\{4, 5\}\!\}$ and returns the following set (it is a set because the comprehension is tagged by the word *set*):

$$\{(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)\}$$

Another example is

$$sum\{\, a \mid a \leftarrow [1, 2, 3],\, a \geq 2 \,\}$$

monoid	type T	zero	unit(a)	merge	C/I
list	list(α)	$[\,]$	$[a]$	$+\!\!+$	
set	set(α)	$\{\}$	$\{a\}$	$\cup$	CI
bag	bag(α)	$\{\!\{\}\!\}$	$\{\!\{a\}\!\}$	$\uplus$	C
oset	list(α)	$[\,]$	$[a]$	$\cup$	I
string	list(char)	""	"a"	concat	
sorted$[f]$	list(α)	$[\,]$	$[a]$	merge$[f]$	CI

Table 1: Examples of Collection Monoids

monoid	type T	zero	unit(a)	merge	C/I
sum	int	0	a	$+$	C
prod	int	1	a	$*$	C
max	int	0	a	max	CI
some	bool	false	a	$\vee$	CI
all	bool	true	a	$\wedge$	CI

Table 2: Examples of Primitive Monoids

where *sum* is the monoid $(int, 0, +)$. This expression returns 5, the sum of all list elements greater than or equal to 2.

The rest of this section gives a formal definition of the monoids and the monoid operations.

2.1 The Formal Framework

Definition 1 (Monoid) *The triple* $\mathcal{M} = (T, \text{zero}, \text{merge})$ *is a monoid of type* T *if* merge *(of type* $T \times T \to T$*) is associative with identity* zero *(i.e.,* $\forall x : \text{merge}(\text{zero}, x) = x = \text{merge}(x, \text{zero})$*).*

In addition, a monoid $\mathcal{M} = (T, \text{zero}, \text{merge})$ may be a *commutative* monoid (i.e., when merge is commutative) and/or an *idempotent* monoid (i.e., when $\forall x : \text{merge}(x, x) = x$). For example, *sum* $= (\text{int}, 0, +)$ is a commutative monoid, while *set* $= (\text{set}(\alpha), \{\}, \cup)$ is a commutative and an idempotent monoid.

Definition 2 (Collection Monoid) *Let* $T(\alpha)$ *be a type determined by the type parameter* α *(i.e.,* T *is a type constructor) and* $\mathcal{M} = (T(\alpha), \text{zero}, \text{merge})$ *be a monoid. The quadruple* $(T(\alpha), \text{zero}, \text{unit}, \text{merge})$*, where* unit *is a function of type* $\alpha \to T(\alpha)$*, is a collection monoid.*

Later, we will be defining operations that involve multiple monoids. When necessary to distinguish the components of a particular monoid $\mathcal{M}$ we qualify them as $\text{zero}^{\mathcal{M}}$, $\text{unit}^{\mathcal{M}}$, and $\text{merge}^{\mathcal{M}}$.

Table 1 presents some examples of collection monoids. The C/I column indicates whether the monoid is a commutative or idempotent monoid. The monoids *list*, *bag*, and *set* capture the well-known collection types for linear lists, bags, and sets (where $\uplus$ is the additive union for bags). The monoid *oset* captures lists with no duplicates. The operator $\cup$ is defined as follows: $x \cup y = x +\!\!+ (y - x)$, where $y - x$ is the list y without any elements from x, e.g., $[2,5,3,1] \cup [3,2,6] = [2,5,3,1,6]$. The monoid *string* captures character strings. The monoid *sorted*$[f]$ is parameterized by the function f whose range is associated with a partial order $\leq$. The merge function of this monoid merges two sorted lists into a sorted list. If x appears before y in a *sorted*$[f]$ list, then $f(x) \leq f(y)$. This monoid was chosen to be idempotent (i.e., duplicates are removed) so it would be isomorphic to the set monoid.

We will use the shorthand $\mathcal{M}\{\, e_1, \ldots, e_n \,\}$ to represent the following construction over the monoid $\mathcal{M}$:

$$\text{merge}(\text{unit}(e_1), \ldots, \text{merge}(\text{unit}(e_{n-1}), \text{unit}(e_n)))$$

In particular, we will use the following shorthands:

$$\begin{aligned} [e_1, \ldots, e_n] &= \mathit{list}\{\, e_1, \ldots, e_n \,\} \\ \{\!\{e_1, \ldots, e_n\}\!\} &= \mathit{bag}\{\, e_1, \ldots, e_n \,\} \\ \{e_1, \ldots, e_n\} &= \mathit{set}\{\, e_1, \ldots, e_n \,\} \end{aligned}$$

Definition 3 (Primitive Monoid) *The quadruple* $(T, \text{zero}, \text{unit}, \text{merge})$*, where* $\mathcal{M} = (T, \text{zero}, \text{merge})$ *is a monoid and* unit *is the identity function (i.e.,* $\forall a : \text{unit}(a) = a$*), is a primitive monoid.*

Table 2 presents some examples of primitive monoids. Note that instances of a primitive monoid $\mathcal{M}$ cannot be generated from the monoid operations alone; instead they need values of type T to generate new values.

We define the mapping ψ from monoids to the set $\{C, I\}$ as: $C \in \psi(\mathcal{M})$ iff $\mathcal{M}$ is commutative and $I \in \psi(\mathcal{M})$ iff $\mathcal{M}$ is idempotent. The partial order between monoid names $\preceq$ is defined as:

$$\mathcal{N} \preceq \mathcal{M} \;\equiv\; \psi(\mathcal{N}) \subseteq \psi(\mathcal{M})$$

For example, *list* $\preceq$ *bag* $\preceq$ *set* since *set* is commutative-idempotent, *bag* is commutative but not idempotent, and *list* is neither commutative nor idempotent. See Figure 1 for a representation of the partial order $\preceq$ for some of the monoids.

We now define our algebraic operator, which is parameterized by input and output monoids.

Definition 4 (Monoid Homomorphism) *A homomorphism* $\text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,A$ *from the collection monoid* $\mathcal{M} = (T(\alpha), \text{zero}^{\mathcal{M}}, \text{unit}^{\mathcal{M}}, \text{merge}^{\mathcal{M}})$ *to any monoid* $\mathcal{N} = (S, \text{zero}^{\mathcal{N}}, \text{merge}^{\mathcal{N}})$*, where* $\mathcal{M} \preceq \mathcal{N}$*, is defined by the following inductive equations:*

$$\begin{aligned} \text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,(\text{zero}^{\mathcal{M}}) &= \text{zero}^{\mathcal{N}} \\ \text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,(\text{unit}^{\mathcal{M}}(a)) &= f(a) \\ \text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,(\text{merge}^{\mathcal{M}}(x, y)) & \\ = \text{merge}^{\mathcal{N}}(\, \text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,x,\ & \text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,y\,) \end{aligned}$$

$$\begin{tikzcd}
 & \mathbf{CI}\ \textit{(set)} & \\
\textit{(bag, sum)}\ \mathbf{C} \arrow[ur] & & \mathbf{I}\ \textit{(oset)} \arrow[ul] \\
 & \underset{\textit{(list)}}{\cdot} \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Figure 1: Restriction Lattice for Homomorphisms

Basically, the expression $\hom^{\mathcal{M}\rightarrow\mathcal{N}}(f)\,A$ replaces the $\text{zero}^{\mathcal{M}}$ in A by $\text{zero}^{\mathcal{N}}$, the $\text{merge}^{\mathcal{M}}$ by $\text{merge}^{\mathcal{N}}$, and the $\text{unit}^{\mathcal{M}}$ by f. That is, function f must have the same type as $\text{unit}^{\mathcal{N}}$. For example, if A is the list $[a_1, \ldots, a_n]$, then $\hom^{list\rightarrow set}(f)\,A$ computes the set $f(a_1) \cup \cdots \cup f(a_n)$.

The condition $\mathcal{M} \preceq \mathcal{N}$ in Definition 4 is important. If the collection monoid $\mathcal{M}$ is a commutative or idempotent monoid, then $\mathcal{N}$ must be too. For example, bag cardinality[1] $\hom^{bag\rightarrow sum}(\lambda x.\,1)\,A$ is a well-formed homomorphism, while set cardinality $\hom^{set\rightarrow sum}(\lambda x.\,1)\,A$ is not (since $+$ is commutative but not idempotent). Without this restriction we would have (see also [6]):

$$\begin{aligned} 1 &= \hom^{set\rightarrow sum}(\lambda x.\,1)\,\{a\} \\ &= \hom^{set\rightarrow sum}(\lambda x.\,1)\,(\{a\} \cup \{a\}) \;=\; 1+1 \;=\; 2 \end{aligned}$$

This restriction also prohibits the conversion of sets into lists (since *set* $\not\preceq$ *list*), but allows the conversion of sets into sorted lists.

The following are examples of well-formed monoid homomorphisms:

$$\begin{array}{lll} \text{image}(f)\,x & = & \hom^{set\rightarrow set}(\lambda a.\,\{f(a)\})\,x \\ x \times y & = & \hom^{set\rightarrow set}(\lambda a.\,\hom^{set\rightarrow set}(\lambda b.\,\{(a,b)\})\,y)\,x \\ e \in x & = & \hom^{set\rightarrow some}(\lambda a.\,(a=e))\,x \\ \text{filter}(p)\,x & = & \hom^{set\rightarrow set}(\lambda a.\,\textbf{if}\,p(a)\,\textbf{then}\,\{a\}\,\textbf{else}\,\{\})\,x \\ \text{length}(x) & = & \hom^{list\rightarrow sum}(\lambda a.\,1)\,x \end{array}$$

where image$(f)\,x$ maps the function f over all elements of the set x, $x \times y$ computes the cartesian product of the sets x and y, and filter$(p)\,x$ selects all elements a of the set x that satisfy the predicate $p(a)$.

Queries in our calculus are expressed in terms of monoid comprehensions. Informally, a monoid comprehension over the monoid $\mathcal{M}$ takes the form $\mathcal{M}\{\, e \mid \overline{q}\,\}$. Expression e is called the *head* of the comprehension. Each term q_i in the term sequence $\overline{q} = q_1, \ldots, q_n$, $n \geq 0$, is called a *qualifier*, and is either

- a *generator* of the form $v \leftarrow e'$, where v is a variable and e' is an expression, or
- a *filter* pred, where pred is a predicate.

Formally, monoid comprehensions are defined in terms of monoid homomorphisms. In particular, all qualifiers in the comprehensions are eliminated from left to right until no qualifier is left:

Definition 5 (Monoid Comprehension) *A monoid comprehension over a primitive or collection monoid $\mathcal{M}$ is defined by the following inductive equations:*

$$\begin{array}{lll} \mathcal{M}\{\, e \mid \,\} & = & \text{unit}^{\mathcal{M}}(e) \\ \mathcal{M}\{\, e \mid x \leftarrow u, \overline{q}\,\} & = & \hom^{\mathcal{N}\rightarrow\mathcal{M}}(\lambda x.\,\mathcal{M}\{\, e \mid \overline{q}\,\})\,u \\ \mathcal{M}\{\, e \mid \text{pred}, \overline{q}\,\} & = & \textbf{if}\,\text{pred}\,\textbf{then}\,\mathcal{M}\{\, e \mid \overline{q}\,\} \\ & & \qquad \textbf{else}\,\text{zero}^{\mathcal{M}} \end{array}$$

where u in $x \leftarrow u$ is an expression that computes an instance of the collection monoid $\mathcal{N}$, where $\mathcal{N} \preceq \mathcal{M}$.

While the monoid $\mathcal{M}$ of the output is specified explicitly, the collection monoid $\mathcal{N}$ associated with the expression u in $x \leftarrow u$ is inferred. (Details of this type inference process can be found elsewhere [16].)

We will use the following convention to represent variable bindings in a comprehension:

$$\mathcal{M}\{\, e \mid \overline{q},\, x \equiv u,\, \overline{s}\,\} \;=\; \mathcal{M}\{\, e[u/x] \mid \overline{q},\, \overline{s}[u/x]\,\}$$

where $e[u/x]$ is the expression e with u substituted for all free occurrences of x. That is, we substitute u for x in all qualifiers q (assuming that there is no other generator over x in q) and in term e. Terms of the form $x \equiv u$ are called *bindings* since they bind the variable x to the expression u. For example, *set*{ b.D | a ← x, b ≡ y, a.B=b.C } is equal to *set*{ y.D | a ← x, a.B=y.C }.

Note that *list*{ x | x ← {1,2} } is not a well-formed comprehension, since it is translated into a homomorphism from *set* to *list*, while *sum*{ x | x ← {{1,2}} } is. That is, if one of the generators in a monoid comprehension is over a commutative or idempotent monoid, then the comprehension must be over a commutative or idempotent monoid. This condition can be checked statically, since the commutativity and idempotence properties of a monoid are specified explicitly when this monoid is defined.

Relational joins can be represented directly as comprehensions. The join of two sets x and y is:

set{ f(a,b) | a ← x, b ← y, p(a,b) }

where p is the join predicate and function f constructs an output set element given two elements from x and y. For example, if p(a,b) = (a.C=b.C) ∧ (a.D > 10) and f(a,b) = ⟨ C=a.C, D=b.D ⟩ (i.e., a record construction), then this comprehension becomes:

set{ ⟨ C=a.C, D=b.D ⟩ | a ← x, b ← y,
a.C=b.C, a.D > 10 }

[1]Expression $\lambda x.e$ is the function f such that $f(x) = e$.

If we use the rules in Definition 5, this comprehension is translated into algebraic form as

$$\text{hom}^{set\to set}(\lambda\text{a. hom}^{set\to set}(\lambda\text{b. }\textbf{if}\ (\text{a.C=b.C}) \wedge (\text{a.D} > 10)\ \textbf{then}\ \{\langle\ \text{C=a.C, D=b.D}\ \rangle\}\ \textbf{else}\ \{\})\ \text{y})\ \text{x}$$

But comprehensions can be used to join different collection types. For example, $set\{\,(x,y) \mid x \leftarrow [1,2],\, y \leftarrow \{\!\{3,4,3\}\!\}\,\}$ is equal to $\{(1,3),(1,4),(2,3),(2,4)\}$. Another example is $\text{nest}(k)\,x$ equal to $set\{\,\langle\ \text{KEY} = k(e),\ \text{P} = set\{\,a \mid a \leftarrow x,\ k(e) = k(a)\,\}\ \rangle \mid e \leftarrow x\,\}$, which is the nesting operator for nested relations. Similarly, $\text{unnest}(x) = set\{\,e \mid s \leftarrow x,\ e \leftarrow s.\text{P}\,\}$. The last comprehension is an example of a *dependent join* in which the value of the second collection s.P depends on the value of s, an element of the first relation x. Dependent joins are a convenient way of traversing nested collections.

Other examples:

$$\begin{array}{lcl}
\text{filter}(p)\,x & = & set\{\,e \mid e \leftarrow x,\ p(e)\,\} \\
\text{flatten}(x) & = & set\{\,e \mid s \leftarrow x,\ e \leftarrow s\,\} \\
x \cap y & = & set\{\,e \mid e \leftarrow x,\ e \in y\,\} \\
\text{length}(x) & = & sum\{\,1 \mid e \leftarrow x\,\} \\
\text{sum}(x) & = & sum\{\,e \mid e \leftarrow x\,\} \\
\text{count}(x,a) & = & sum\{\,1 \mid e \leftarrow x,\ e = a\,\} \\
\exists a \in x : e & = & some\{\,e \mid a \leftarrow x\,\} \\
\forall a \in x : e & = & all\{\,e \mid a \leftarrow x\,\} \\
a \in x & = & some\{\,a = e \mid e \leftarrow x\,\}
\end{array}$$

The expression $\text{sum}(x)$ adds the elements of any non-idempotent monoid x, e.g., $\text{sum}([1,2,3]) = 6$. The expression $\text{count}(x,a)$ counts the number of occurrences of a in the bag x, e.g., $\text{count}(\{\!\{1,2,1\}\!\},1) = 2$.

From Definition 5 and from the equations

$$\text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,A \quad = \quad \mathcal{N}\{\,y \mid x \leftarrow A,\ y \leftarrow f(x)\,\}$$

if $\mathcal{N}$ is a collection monoid and

$$\text{hom}^{\mathcal{M}\to\mathcal{N}}(f)\,A \quad = \quad \mathcal{N}\{\,f(x) \mid x \leftarrow A\,\}$$

if $\mathcal{N}$ is a primitive monoid, we conclude that all monoid homomorphisms can be expressed as monoid comprehensions and vice versa.

2.2 The Monoid Comprehension Calculus

In our treatment of queries we will consider only the following types to be valid:

Definition 6 (Monoid Type) *A* monoid type *has one of the following forms:*

class_name	*(a reference to a class)*
T	*(T is a primitive type)*
$T(type)$	*(T is a collection type)*
$\langle\ A_1 : t_1, \ldots, A_n : t_n\ \rangle$	*(a record type)*

where type and $t_1, \ldots, t_n$ are monoid types.

That is, collection types can be freely nested.

An OODB schema is a set of potentially mutually recursive class definitions. A class definition takes the form:

class class_name $=$ *type* **extent** : extent_name

where *type* is a monoid type. The extent_name is an optional collection (a set) of all instances of this class. There are two ways of accessing instances of a class: via the class extent, and via the persistent variables, which are also part of the OODB schema.

For example, the following is an OODB schema in our language

```
class person = ⟨ name: string, address: string, spouse: person ⟩
                   extent: persons;

class hotel = ⟨ name: string, address: string,
                  facilities: set(string), employees: set(person),
                  rooms: set(⟨ bed#: int, price: int ⟩) ⟩
                extent: hotels;

class city = ⟨ name: string, hotels: bag(hotel),
                places_to_visit: list(⟨ name: string,
                                         address: string ⟩) ⟩
              extent: cities;

var my_city: city = ⟨ name="Portland", ... ⟩;
```

where my_city is a persistent variable.

Definition 7 (Monoid Comprehension Calculus) *The monoid calculus consists of the following syntactic forms:*

v	variable
c	constant
$e.A$	projection
$\langle\ A_1 = e_1, \ldots, A_n = e_n\ \rangle$	record construction
$e_1 \otimes e_2$	$\otimes \in \{=,<,>,\leq,\geq,\neq\}$
$\text{zero}^{\mathcal{M}}$	
$\text{unit}^{\mathcal{M}}(e)$	
$\text{merge}^{\mathcal{M}}(e_1,e_2)$	
$\mathcal{M}\{\,e \mid q_1, \ldots, q_n\,\}$	comprehension

where e, $e_1, \ldots, e_n$ are terms in the monoid calculus, v is a variable, and $q_1, \ldots, q_n$ are qualifiers of the form $v \leftarrow e$ or e.

For example, the following is a valid expression in the calculus:

$$bag\{\ \text{h.name} \mid \text{hl} \leftarrow bag\{\ \text{c.hotels} \mid \text{c} \leftarrow \text{Cities},\ \text{c.name="Portland"}\ \},\ \text{h} \leftarrow \text{hl},\ some\{\ \text{r.bed\#=3} \mid \text{r} \leftarrow \text{h.rooms}\ \}\ \}$$

2.3 Translating ODMG-93 OQL to the Monoid Calculus

One of our main claims for the effectiveness of the monoid calculus is that it gives better coverage of language features of real query languages. Nearly all OQL expressions have a direct translation into the monoid calculus, with the exception of indexed OQL collections. (In Section 4 we present a monoid for indexed collections as well as a comprehension syntax to capture complex vector and array operations.) Here we illustrate how to translate the main constructs of OQL in the monoid calculus.

A select-from-where OQL statement of the form:

$$\begin{array}{l}\textbf{select}\ e\\ \quad\textbf{from}\ x_1\ \textbf{in}\ e_1,\ \ldots,\ x_n\ \textbf{in}\ e_n\\ \quad\textbf{where}\ \mathit{pred}\end{array}$$

is translated into:

$$\mathit{bag}\{\, e \mid x_1 \leftarrow e_1,\ \ldots,\ x_n \leftarrow e_n,\ \mathit{pred}\,\}$$

Note that e, the e_i and *pred* could all contain nested comprehension expressions, which supports the capability in OQL to have nested queries in the *select*, *where*, and *from* clauses. For example, the following nested OQL query:

```
select h.address
   from hl in (select c.hotels
                  from c in Cities
                  where c.name="Portland"),
        h in hl
   where h.name="Hilton"
```

is expressed in the comprehension syntax as follows:

```
bag{ h.address | hl ← bag{ c.hotels | c ← Cities,
                                      c.name="Portland" },
                 h ← hl, h.name="Hilton" }
```

The **select-distinct** OQL statement is translated into a set comprehension in a similar way. For example, consider the following query against the previous OODB schema that finds all hotels in Portland that are also interesting places to visit:

```
select distinct h.name
       from c in Cities,
            h in c.hotels,
            p in c.places_to_visit
where c.name="Portland" and h.name=p.name
```

This query is translated into the following comprehension:

```
set{ h.name | c ← db.cities, h ← c.hotels, p ← c.places_to_visit,
              c.name="Portland", h.name=p.name }
```

The OQL **group-by** operator takes two forms. The simplest form is:

$$\textbf{group}\ x\ \textbf{in}\ e\ \textbf{by}\ (A_1 : e_1(x), \ldots, A_n : e_n(x))$$

This operation partitions the collection e according to the partition functions $e_1, \ldots, e_n$ in the *by* clause. For each different combination of values $e_1(x), \ldots, e_n(x)$, it creates a partition that contains all elements of e that are mapped to these values. This form is translated into:

$$\begin{array}{l}\mathit{set}\{\,\langle\, A_1 = e_1(x), \ldots, A_n = e_n(x),\\ \quad\quad \text{partition} = \mathit{set}\{\, a \mid a \leftarrow e,\ e_1(a) = e_1(x), \ldots,\\ \quad\quad\quad\quad e_n(a) = e_n(x)\,\}\,\rangle \mid x \leftarrow e\,\}\end{array}$$

The more general form is:

$$\begin{array}{l}\textbf{group}\ x\ \textbf{in}\ e\ \textbf{by}\ (A_1 : e_1(x), \ldots, A_n : e_n(x))\\ \textbf{with}\ (B_1 : u_1(x, \text{partition}), \ldots, B_m : u_m(x, \text{partition}))\end{array}$$

which is translated into:

$$\begin{array}{l}\mathit{set}\{\,\langle\, A_1 = e_1(x), \ldots, A_n = e_n(x),\\ \quad B_1 = u_1(x, \text{partition}), \ldots, B_m = u_m(x, \text{partition})\,\rangle\\ \quad \mid x \leftarrow e,\ \text{partition} \equiv \mathit{set}\{\, a \mid a \leftarrow e,\ e_1(a) = e_1(x), \ldots,\\ \quad\quad\quad e_n(a) = e_n(x)\,\}\,\}\end{array}$$

The following table gives the translation of other OQL expressions into the monoid calculus:

e_1 **intersect** e_2	$\longrightarrow$	$\mathit{set}\{\, x \mid x \leftarrow e_1,\ x\ \textbf{in}\ e_2\,\}$
for all x **in** e : *pred*	$\longrightarrow$	$\mathit{all}\{\,\mathit{pred} \mid x \leftarrow e\,\}$
exists x **in** e : *pred*	$\longrightarrow$	$\mathit{some}\{\,\mathit{pred} \mid x \leftarrow e\,\}$
e_1 **in** e_2	$\longrightarrow$	$\mathit{some}\{\, x = e_1 \mid x \leftarrow e_2\,\}$
count(e)	$\longrightarrow$	$\mathit{sum}\{\, 1 \mid x \leftarrow e\,\}$
sum(e)	$\longrightarrow$	$\mathit{sum}\{\, x \mid x \leftarrow e\,\}$
flatten(e)	$\longrightarrow$	$\mathit{set}\{\, x \mid s \leftarrow e,\ x \leftarrow s\,\}$
sort x **in** e_1 **by** e_2	$\longrightarrow$	$\mathit{sorted}[f]\{\, x \mid x \leftarrow e_1\,\}$

where in the last example, $f(x) = e_2$. For example, the following OQL query finds all hotels in Portland that have at least one room with three beds:

```
select h.name
  from hl in (select c.hotels
                 from c in Cities
                 where c.name="Portland"),
       h in hl
  where exists r in h.rooms: (r.bed#=3)
```

This query is expressed in the comprehension syntax as follows:

```
bag{ h.name | hl ← bag{ c.hotels | c ← Cities,
                                   c.name="Portland" },
              h ← hl, some{ r.bed#=3 | r ← h.rooms } }
```

The restriction on monoid comprehensions relating idempotence and commutativity of the monoids involved turns out not to be a limitation in translation. OQL *select* statements always return sets or bags, and the only explicit conversion function on collections is **listtoset**, which the calculus allows.

$$\mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow \text{zero}^{\mathcal{N}},\, \overline{s}\,\} \longrightarrow \text{zero}^{\mathcal{M}} \quad (1)$$

$$\mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow \text{unit}^{\mathcal{N}}(e'),\, \overline{s}\,\} \longrightarrow \mathcal{M}\{\, e \mid \overline{q},\, v \equiv e',\, \overline{s}\,\} \quad (2)$$

$$\mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow \text{merge}^{\mathcal{N}}(e_1, e_2),\, \overline{s}\,\} \longrightarrow \text{merge}^{\mathcal{M}}(\mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow e_1,\, \overline{s}\,\},\, \mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow e_2,\, \overline{s}\,\}) \quad (3)$$

(if $\mathcal{M}$ is commutative)

$$\langle\, A_1 = e_1, \ldots, A_n = e_n \,\rangle.A_i \longrightarrow e_i \quad (4)$$

$$\mathcal{M}\{\, e \mid \overline{q},\, v \leftarrow \mathcal{N}\{\, e' \mid \overline{r}\,\},\, \overline{s}\,\} \longrightarrow \mathcal{M}\{\, e \mid \overline{q},\, \overline{r},\, v \equiv e',\, \overline{s}\,\} \quad (5)$$

$$\mathcal{M}\{\, e \mid \overline{q},\, \mathit{some}\{\, \mathit{pred} \mid \overline{r}\,\},\, \overline{s}\,\} \longrightarrow \mathcal{M}\{\, e \mid \overline{q},\, \overline{r},\, \mathit{pred},\, \overline{s}\,\} \quad (6)$$

Table 3: The Normalization Algorithm

3 Program Normalization

Another claim for the monoid calculus is that it supports easy manipulation of query expressions. It is amenable to pattern-based rewriting. Here we illustrate one such transformation on the monoid calculus, which we have implemented on an earlier version of the calculus.

The monoid calculus can be put into a canonical form by an efficient rewrite algorithm, called the *normalization algorithm*. The evaluation of these canonical forms generally produces fewer intermediate data structures than the initial unnormalized programs. Moreover, the normalization algorithm improves program performance in many cases. It generalizes many optimization techniques already used in relational algebra, such as pushing a selection before a join.

Table 3 gives the normalization rules. Rule 5 is the most important: it flattens a nested comprehension (i.e., a comprehension that contains a generator whose domain is another comprehension). Rule 6 unnests an existential quantification. There are other cases of query unnesting that are not covered here, but for which we are currently extending the normalization algorithm.

One advantage of the normalization algorithm, or any algorithm on calculus expressions expressed via pattern-based rewrite, is that it can be shown to correctly preserve meaning by proving each rewrite transformation is correct. Proofs of correctness for the rules in Table 3 are given elsewhere [16].

Rules 5 and 6 may require some variable renaming to avoid name conflicts. If there is a generator $v' \leftarrow e_1$ in $\overline{q}$ and a generator $v' \leftarrow e_2$ in $\overline{r}$ then variable v' in $\overline{r}$ should be renamed. For example, filter(p)(filter(q) x) is

$$\begin{aligned}&\mathit{set}\{\ \mathsf{a} \mid \mathsf{a} \leftarrow \mathit{set}\{\ \mathsf{a} \mid \mathsf{a} \leftarrow \mathsf{x},\ \mathsf{q(a)}\ \},\ \mathsf{p(a)}\ \}\\ &= \mathit{set}\{\ \mathsf{a} \mid \mathsf{a} \leftarrow \mathit{set}\{\ \mathsf{b} \mid \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)}\ \},\ \mathsf{p(a)}\ \}\end{aligned}$$

(by renaming the inner variable a to b) and it is normalized into:

$$\begin{aligned}&\mathit{set}\{\ \mathsf{a} \mid \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)},\ \mathsf{a} \equiv \mathsf{b},\ \mathsf{p(a)}\ \}\\ &= \mathit{set}\{\ \mathsf{b} \mid \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)},\ \mathsf{p(b)}\ \}\end{aligned}$$

(by Rule 5 and by definition of $\equiv$) which is a filter whose predicate is the conjunction of p and q. As another example of normalization, consider the following nested OQL query:

```
select distinct r
  from r in R
  where r.B in (select distinct s.D
                  from s in S
                  where r.C=s.C)
```

which is expressed in the monoid calculus as follows:

$$\begin{aligned}&\mathit{set}\{\ \mathsf{r} \mid \mathsf{r} \leftarrow \mathsf{R},\\ &\qquad \mathit{some}\{\ \mathsf{x{=}r.B} \mid \mathsf{x} \leftarrow \mathit{set}\{\ \mathsf{s.D} \mid \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C}\ \}\ \}\ \}\end{aligned}$$

and it is normalized into:

$$\begin{aligned}&\mathit{set}\{\ \mathsf{r} \mid \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{x} \leftarrow \mathit{set}\{\ \mathsf{s.D} \mid \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C}\ \},\ \mathsf{x{=}r.B}\ \}\\ &= \mathit{set}\{\ \mathsf{r} \mid \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C},\ \mathsf{x} \equiv \mathsf{s.D},\ \mathsf{x{=}r.B}\ \}\\ &= \mathit{set}\{\ \mathsf{r} \mid \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C},\ \mathsf{s.D{=}r.B}\ \}\end{aligned}$$

(by rules 6 and 5 and by definition of $\equiv$.) Consider now the query, presented in Section 2.3, that finds all hotels in Portland that have at least one room with three beds:

$$\begin{aligned}&\mathit{bag}\{\ \mathsf{h.name} \mid \mathsf{hl} \leftarrow \mathit{bag}\{\ \mathsf{c.hotels} \mid \mathsf{c} \leftarrow \mathsf{Cities},\\ &\qquad\qquad \mathsf{c.name{=}\text{“Portland”}}\ \},\\ &\qquad \mathsf{h} \leftarrow \mathsf{hl},\ \mathit{some}\{\ \mathsf{r.bed\#{=}3} \mid \mathsf{r} \leftarrow \mathsf{h.rooms}\ \}\ \}\end{aligned}$$

This query is normalized into:

$$\begin{aligned}&\mathit{bag}\{\ \mathsf{h.name} \mid \mathsf{c} \leftarrow \mathsf{Cities},\ \mathsf{c.name{=}\text{“Portland”}},\\ &\qquad \mathsf{hl} \equiv \mathsf{c.hotels},\\ &\qquad \mathsf{h} \leftarrow \mathsf{hl},\ \mathsf{r} \leftarrow \mathsf{h.rooms},\ \mathsf{r.bed\#{=}3}\ \}\\ &= \mathit{bag}\{\ \mathsf{h.name} \mid \mathsf{c} \leftarrow \mathsf{Cities},\ \mathsf{c.name{=}\text{“Portland”}},\\ &\qquad \mathsf{h} \leftarrow \mathsf{c.hotels},\ \mathsf{r} \leftarrow \mathsf{h.rooms},\ \mathsf{r.bed\#{=}3}\ \}\end{aligned}$$

(by rules 4 and 5 and by definition of $\equiv$.)

The normalization algorithm can also be used for handling the inefficiencies introduced when new programming language constructs are incorporated into relational languages, such as in SQL3 [2]. (SQL3 contains many new proposed relational extensions such as user-defined types, multiple collections types, routines, and triggers.) In particular, the begin-end statement in SQL3 may introduce inefficiencies when it contains bindings to large intermediate results. For example,

```
begin declare hs SQL_set(hotel); addr string;
        select hotels into hs from cities where name='C';
        select address into addr from hs where name='H';
        return addr;
end;
```

Notice that hs (a set of hotels) is generated by the first select statement only to be used in the second select statement. This query is translated in comprehension form as follows:

$$\begin{aligned}&set\{\ \mathsf{addr}\ \mid\ \mathsf{hs} \leftarrow set\{\ \mathsf{c.hotels}\ \mid\ \mathsf{c} \leftarrow \mathsf{cities},\ \mathsf{c.name}=\text{“C”}\ \},\\ &\qquad\qquad \mathsf{addr} \leftarrow set\{\ \mathsf{h.address}\ \mid\ \mathsf{h} \leftarrow \mathsf{hs},\ \mathsf{h.name}=\text{“H”}\ \}\ \}\end{aligned}$$

If we normalize this expression, we derive a program that does not materialize any intermediate result.

A path *path* is a *name* (the identifier of a bound variable, or the identifier of a persistent variable, or the name of a class extent) or an expression *path'.name* (where *name* is an attribute name of a record and *path'* is a path). If the generator domains in a comprehension do not contain any non-commutative merges (such as the list append), then these domains can be normalized into paths. That is, monoid comprehensions can be put into the following canonical form. (We assume that all predicates are pushed to the end of the comprehension, forming a conjunction *pred* of predicates.):

$$\mathcal{M}\{\, e \mid v_1 \leftarrow path_1, \ldots, v_n \leftarrow path_n, pred \,\}$$

where each $path_i$ is a path. The proof of this statement is easy: if the domain of a generator in a monoid comprehension is a form other than a path, then this domain is reduced to a simpler form by the normalization algorithm.

In addition to the normalization rules, there are other important program transformations that explore the commutativity properties of monoids. In particular, if $\mathcal{M}$ is a commutative monoid, then we have the following join commutativity rule:

$$\begin{aligned}&\mathcal{M}\{\, e \mid \overline{q},\ v_1 \leftarrow e_1,\ v_2 \leftarrow e_2,\ \overline{r}\,\}\\ &\qquad \longrightarrow\ \mathcal{M}\{\, e \mid \overline{q},\ v_2 \leftarrow e_2,\ v_1 \leftarrow e_1,\ \overline{r}\,\}\end{aligned}$$

which holds only when term e_2 does not depend on v_1.

The following transformation, which is valid for any monoid $\mathcal{M}$, pushes a selection before a join if predicate p does not depend on v:

$$\mathcal{M}\{\, e \mid \overline{q},\ v \leftarrow e_1,\ p,\ \overline{r}\,\}\ \longrightarrow\ \mathcal{M}\{\, e \mid \overline{q},\ p,\ v \leftarrow e_1,\ \overline{r}\,\}$$

4 Language Extensions

While the monoid calculus is capable of directly representing the main features of a language such as OQL, it is in fact capable of expressing queries that are not expressible in OQL. For example, comprehensions can yield values for any monoid type, whereas the OQL *select* statement always yields sets or bags. We would like to see OQL extended with *select* statements that return collections other than sets or bags, such as:

```
select list e.B
from e in (sort s in x by s.A)
```

which sorts x by A and then projects over B preserving the order.

Here we examine a monoid for vectors, which could be a basis for extending the expressiveness of OQL. We also show how to handle identity and updates in the calculus.

4.1 Vectors and Arrays

Vectors and arrays are important collection types for scientific and other applications [22]. In contrast to other collection types, there is no obvious monoid that captures vectors effectively. Vector operations should provide random access through indexing as well as bulk manipulation. We will first propose an effective form of vector comprehensions and then describe a monoid that captures these comprehensions. An example of a vector manipulation is vector reverse, which computes $x[n-i-1]$ for $i = 0, 1, \ldots, n-1$ from a vector x of size n. This function can be computed by

$$vec[n]\{\, a[n-i-1] \mid a[i] \leftarrow x \,\}$$

where $vec[n]$ is the anticipated monoid for vectors of size n [7]. Note that we want to access both the value a and the associated index i from the vector x, but we do not want to impose any order on the way $a[i]$ is accessed. The generator $a[i] \leftarrow x$ accesses the pairs (a, i) in some unspecified order, much as elements of a set or a list x are retrieved in a comprehension by the generator $a \leftarrow x$. But the constructed element of the vector comprehension above is $a[n-i-1]$, which means that we should be able to store an element at any position in the resulting vector. That is, even though vector elements are accessed in bulk fashion (in pairs of value-index values), vectors are constructed in random fashion by specifying which value is stored at what place. But there is a problem here: what if we have two different values stored in the same place in a vector? We need to perform a merge operation on vectors. One solution is to merge the vector elements individually. That is, when two elements are stored at the same place in a vector, the resulting vector value is computed by merging these two elements. Another solution, which is not considered here, is to choose the last of the two values if this value is not zero, otherwise to choose the first. That way, the last non-zero value overwrites the previous values.

We will now formalize these observations. We introduce a new collection monoid $\mathcal{M}[n]$, for some monoid $\mathcal{M}$ and some constant integer n, to denote all vectors of size n with elements of type $\mathcal{M}$. This monoid has the following primitives: (The unit function here is binary, but in functional languages any n-ary function can be considered as a unary applied to a tuple.)

$$\text{zero}^{\mathcal{M}[n]} = (\!|\text{zero}^{\mathcal{M}}, \ldots, \text{zero}^{\mathcal{M}}|\!)$$

$$\text{unit}^{\mathcal{M}[n]}(a,k) = (\!|e_0, \ldots, e_{n-1}|\!) \quad \textit{where } e_i = \begin{cases} \text{unit}^{\mathcal{M}}(a) & \text{if } i = (k \bmod n) \\ \text{zero}^{\mathcal{M}} & \text{otherwise} \end{cases}$$

$$\text{merge}^{\mathcal{M}[n]}((\!|a_0, \ldots, a_{n-1}|\!), (\!|b_0, \ldots, b_{n-1}|\!)) \quad (\!|\text{merge}^{\mathcal{M}}(a_0, b_0), \ldots, \text{merge}^{\mathcal{M}}(a_{n-1}, b_{n-1})|\!)$$

where $(\!|e_0, \ldots, e_{n-1}|\!)$ constructs a vector of n elements. Thus, the zero element of $\mathcal{M}[n]$ is a vector of n zeros; the unit takes two values $a : \mathcal{M}$ and $k : int$ and constructs a vector of n zeros, except the k^{th} element is set to a; the merge function uses $\text{merge}^{\mathcal{M}}$ to merge the two input vectors element-wise. For example,

$$\text{zero}^{sum[4]} = (\!|0,0,0,0|\!)$$

$$\text{unit}^{sum[4]}(8,2) = (\!|0,0,8,0|\!)$$

$$\text{merge}^{sum[4]}((\!|0,1,2,0|\!), (\!|3,0,2,1|\!)) = (\!|0+3, 1+0, 2+2, 0+1|\!) = (\!|3,1,4,1|\!)$$

Note that the monoid $\mathcal{M}[n]$ is not freely generated from $\mathcal{M}$, as is the case for the other collection monoids. Instead, the $\mathcal{M}[n]$ operations depend on the $\mathcal{M}$ operations.

When there is no ambiguity, we will use $\text{merge}^{\mathcal{M}}$ for merging vectors instead of $\text{merge}^{\mathcal{M}[n]}$. For example,

$$\begin{array}{lcl} (\!|1,2|\!) + (\!|3,4|\!) & = & (\!|4,6|\!) \\ (\!|[1,2],[3]|\!) +\!\!+ (\!|[4],[5,6]|\!) & = & (\!|[1,2,4],[3,5,6]|\!) \end{array}$$

The matrix sum of two matrices x and y of type $mat = \text{sum}[n][m]$ is then $x+y$, since merge^{mat} is $\text{merge}^{sum[n]}$, which is merge^{sum}, namely $+$.

The following are examples of vector manipulations. To make the comprehensions more readable, we represent a pair of the form (a,i) as $a[i]$.

$$\begin{array}{lcl} \text{sum\_all}(x) & = & sum\{\, a \mid a[i] \leftarrow x \,\} \\ \text{subseq}(x,n,l) & = & sum[l]\{\, a[i-n] \mid a[i] \leftarrow x,\ i \geq n \,\} \\ \text{permute}(x,p) & = & sum[n]\{\, a[b] \mid a[i] \leftarrow x,\ b[j] \leftarrow p,\ i = j \,\} \\ \text{concat}(x,y) & = & (sum[n+m]\{\, a[i] \mid a[i] \leftarrow x \,\}) \\ & & +(sum[n+m]\{\, b[n+i] \mid b[i] \leftarrow y \,\}) \end{array}$$

that is, $\text{subseq}(x,n,l)$ returns the vector $x[i]$, $i = n, \ldots, l+n-1$ and $\text{permute}(x,p)$ returns the vector $x[p[i]]$, $i = 1, \ldots, n$. If $mat = sum[n][m]$, i.e., mat is the type of $n \times m$ integer matrices, then the following are examples of matrix operations:

$$\begin{array}{lcl} \text{map}(f)\,x & = & mat\{\, (f(b))[j][i] \mid a[i] \leftarrow x,\ b[j] \leftarrow a \,\} \\ \text{transpose}(x) & = & mat\{\, b[i][j] \mid a[i] \leftarrow x,\ b[j] \leftarrow a \,\} \\ \text{inner}(x,y) & = & sum\{\, a \mid a[i] \leftarrow x * y \,\} \\ \text{multiply}(x,y) & = & mat\{\, v[j][i] \mid a[i] \leftarrow x,\ b[j] \leftarrow \text{transpose}(y), \\ & & \qquad v \equiv \text{inner}(a,b) \,\} \end{array}$$

To understand map, notice that the resulting matrix is formed by merging the values of the form $\text{unit}^{mat}((f(b))[j][i]) = \text{unit}^{sum[n][m]}((f(b),j),i)$, which is all zeros, except the ij^{th} element, which is $f(b)$. In an OQL-like syntax, $\text{map}(f)\,x$ could be expressed as:

```
select sum[n][m] $(f(b))[j][i]$
from $a[i]$ in $x$,
     $b[j]$ in $a$
```

4.2 Object Identity and Database Updates

The monoid calculus can be extended to capture object identity. We introduce a new type constructor $obj(\alpha)$ that represents all objects with states represented by values of type α. In addition, we extend the monoid calculus with the following operations [23]:

- $\text{new}(s)$ that creates a new object with state s;
- $!e$ that dereferences the object e (returns the state of e);
- $e := s$ that changes the state of the object e to s and returns true.

For example, one valid object-oriented comprehension is

$$list\{\, !x \mid x \leftarrow [\text{new}(1), \text{new}(2)],\ x := !x + 1 \,\}$$

which first creates a list of type $list(obj(int))$ containing two new objects ($\text{new}(1)$ and $\text{new}(2)$). Variable x ranges over this list (i.e., x is of type $obj(int)$) and the state of x is incremented by one (by $x := !x + 1$). The result of this computation is the list $[2,3]$. Other examples are:

$$\begin{array}{lcl} some\{\, x = y \mid x \equiv \text{new}(1),\ y \equiv \text{new}(1) \,\} & \rightarrow & \text{false} \\ some\{\, !x = !y \mid x \equiv \text{new}(1),\ y \equiv \text{new}(1) \,\} & \rightarrow & \text{true} \\ some\{\, x = y \mid x \equiv \text{new}(1),\ y \equiv x,\ y := 2 \,\} & \rightarrow & \text{true} \\ sum\{\, !x \mid x \equiv \text{new}(1),\ y \equiv x,\ y := 2 \,\} & \rightarrow & 2 \\ set\{\, e \mid x \equiv \text{new}([\,]),\ x := [1,2],\ e \leftarrow !x \,\} & \rightarrow & \{1,2\} \\ list\{\, !x \mid x \equiv \text{new}(0),\ e \leftarrow [1,2,3,4],\ x := !x + e \,\} & & \\ & \rightarrow & [1,3,6,10] \end{array}$$

The first example indicates that different objects are distinct while the second indicates that objects can have equal states.

We have defined a monoid with higher-order primitives that captures these object-oriented computations as state transformers [16]. These state transformers propagate the object heap (which contains bindings from OIDs to object states) through all operations in an expression, and change it in response to any operation that creates a new object or modifies an existing object. This translation using state transformers captures precisely the semantics of object identity without the need of extending the base model. It also provides an equational theory that allows us to do valid optimizations for object-oriented queries.

Database updates can be captured by extending Definition 5 with the following comprehension qualifiers: Qualifier $path$ **:=** u destructively replaces the value stored at $path$ with u, qualifier $path$ **+=** u merges the singleton u with $path$, and qualifier $path$ **-=** u deletes all elements of $path$ equal to u. For example, the SQL3 program that inserts a new hotel is:

```
select * into c from cities where name=city_name;
insert into c..hotels values(hotel_name, hotel_address,
                             set(), set(), set());
set c..hotel# = c..hotel#+1;
```

This program has the following comprehension form:

```
set{ c | c ← set{ c | c ← db.cities, c.name=city_name },
         c.hotels += ⟨ name=hotel_name,
                       address=hotel_address, facilities={},
                       employees={}, rooms={} ⟩,
         c.hotel# += 1 }
```

5 Related Work

There are many proposals for object query algebras (see for example [21, 13, 11, 3, 25]). In contrast to our algebra, these algebras support multiple bulk operators. But, as we have demonstrated in this paper, we get enough expressive power with just one operator, namely the monoid homomorphism. Supporting a small number of operators is highly desirable, since the more bulk operations an algebra supports, the more transformation rules it needs and, therefore, the harder the optimization task becomes.

Our framework is based on monoid homomorphisms, which were first introduced as an effective way to capture database queries by V. Tannen, et al. [4, 6, 5]. Their form of monoid homomorphism (also called structural recursion over the union presentation — SRU) is more expressive than ours. Operations of the SRU form, though, require the validation of the associativity, commutativity, and idempotence properties of the monoid associated with the output of this operation. These properties are hard to check by a compiler [6], which makes the SRU operation impractical. They first recognized that there are some special cases where these conditions are automatically satisfied, such as for the $\text{ext}(f)$ operation (which is equivalent to $\text{hom}^{\mathcal{M}\rightarrow\mathcal{M}}(f)$ for a monoid $\mathcal{M}$). In our view, SRU is too expressive, since inconsistent programs cannot always be detected in that form. Moreover, the SRU operator can capture non-polynomial operations, such as the powerset, which complicate query optimization. In fact, to our knowledge, there is no normalization algorithm for SRU forms in general (i.e., SRU forms cannot be put in canonical form). On the other hand, $\text{ext}(f)$ is not expressive enough, since it does not capture operations that involve different collection types and it cannot express predicates and aggregates. We believe that our monoid homomorphism algebra is the most expressive subset of SRU where inconsistencies can always be detected at compile time, and, more importantly, where all programs can be put in canonical form.

Monad comprehensions were first introduced by P. Wadler [29] as a generalization of list comprehensions (which already exist in some functional languages). Monoid comprehensions are related to monad comprehensions, but they are considerably more expressive. In particular, monoid comprehensions can mix inputs from different collection types and may return output of a different type. This is not possible for monad comprehensions, since they restrict the inputs and the output of a comprehension to be of the same type. Monad comprehensions were first proposed as a convenient and practical database language by P. Trinder [27, 26, 10], who also presented many algebraic transformations over these forms as well as methods for converting comprehensions into joins. The monad comprehension syntax was also adopted by P. Buneman, et al. [8] as an alternative syntax to monoid homomorphisms. The comprehension syntax was used for capturing operations that involve collections of the same type while structural recursion was used for expressing the rest of the operations (such as converting one collection type to another, predicates, and aggregates).

Our normalization algorithm is highly influenced by L. Wong's work on normalization of monad comprehensions [30]. He presented some powerful rules for flattening nested comprehensions into canonical comprehensions whose generators are over simple paths. These canonical forms are equivalent to our canonical forms for monoid homomorphisms.

6 Conclusion

We believe the monoid calculus provides an effective framework for processing object-oriented query languages such as OQL. The only significant area of coverage left to deal with is method invocation, and our initial studies indicate no insurmountable problems there. We would also like to turn this work around, and use the monoid calculus to improve query languages. One area is in ensuring languages such as OQL have a well-founded semantics. Another is to put features into such languages to match the expressive capabilities of the monoid calculus, such as *select*s that return lists and more comprehensive manipulations on arrays.

Acknowledgements: The authors are grateful to Len Shapiro and Bennet Vance for helpful comments on the paper. This work is supported by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518, and by NSF Grant IRI 91 18360.

References

[1] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. In *International Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt*, 1987.

[2] D. Beech. Collections of Objects in SQL3. In *VLDB'93*, pp 244–255.

[3] C. Beeri and Y. Kornatzky. Algebraic Optimization of Object-Oriented Query Languages. In *International Conference on Database Theory, Paris, France*, pp 72–88. Springer-Verlag, December 1990. LNCS 470.

[4] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages*, pp 9–19. August 1991.

[5] V. Breazu-Tannen, P. Buneman, and L. Wong. Naturally Embedded Query Languages. In *4th International Conference on Database Theory, Berlin, Germany*, pp 140–154. Springer-Verlag, October 1992. LNCS 646.

[6] V. Breazu-Tannen and R. Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *18th International Colloquium on Automata, Languages and Programming, Madrid, Spain*, pp 60–75, July 1991. LNCS 510.

[7] P. Buneman. The Fast Fourier Transform as a Database Query. Technical report, University of Pennsylvania, March 1993. MS-CIS-93-37/L&C 60.

[8] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

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

[10] D. Chan and P. Trinder. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Twelfth British Conference on Databases*, pp 55–72, July 1994. LNCS 826.

[11] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *SIGMOD'92*, pp 383–392.

[12] P. Dadam, et al. A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies. In *SIGMOD'86*, pp 356–367.

[13] S. Danforth and P. Valduriez. A FAD for Data Intensive Applications. *Transactions on Knowledge and Data Engineering*, 4(1):34–51, February 1992.

[14] U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control Over Duplicate Elimination. In *PODS'82*, pp 117–123.

[15] O. Deux, et al. The Story of O2. *Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[16] L. Fegaras. A Uniform Calculus for Collection Types. Oregon Graduate Institute Technical Report 94-030. Available by anonymous ftp from `cse.ogi.edu:/pub/crml/tapos.ps.Z`.

[17] L. Fegaras and D. Maier. An Algebraic Framework for Physical OODB Design. Available by anonymous ftp from `cse.ogi.edu:/pub/crml/oodb-design.ps.Z`.

[18] S. K. Gadia. A Homogeneous Relational Model and Query Languages for Temporal Databases. *Transactions on Database Systems*, 13(4):418–448, December 1988.

[19] N. Immerman, S. Patnaik, and D. Stemple. The Expressiveness of a Family of Finite Set Languages. In *PODS'91*, pp 37–52.

[20] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *VLDB'90*, pp 290–301.

[21] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik. The AQUA Data Model and Algebra. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City*, pp 157–175, August 1993.

[22] D. Maier and B. Vance. A Call to Order. In *PODS'93*, pp 1–16.

[23] A. Ohori. Representing Object Identity in a Pure Functional Language. In *International Conference on Database Theory, Paris, France*, pp 41–55. Springer-Verlag, December 1990. LNCS 470.

[24] G. Özsoyoğlu, Z. Özsoyoğlu, and V. Matos. Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. *ACM Transactions on Database Systems*, 12(4):566–592, December 1987.

[25] P. Pistor and R. Traunmueller. A Database Language for Sets, Lists, and Tables. *Information Systems*, 11(4):323–336, 1986.

[26] P. Trinder. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages*, pp 55–68, August 1991.

[27] P. Trinder and P. Wadler. Improving List Comprehension Database Queries. In *in Proceedings of TENCON'89, Bombay, India*, pp 186–192, November 1989.

[28] S. Vandenberg and D. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. In *SIGMOD'91*, pp 158–167.

[29] P. Wadler. Comprehending Monads. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France*, pp 61–78, June 1990.

[30] L. Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *PODS'93*, pp 26–36.