Optimizing Object Queries Using an Effective Calculus

LEONIDAS FEGARAS
The University of Texas at Arlington
and
DAVID MAIER
Oregon Graduate Institute of Science & Technology

Object-oriented databases (OODBs) provide powerful data abstractions and modeling facilities but they generally lack a suitable framework for query processing and optimization. One of the key factors for OODB systems to successfully compete with relational systems as well as to meet the performance requirements of many non-traditional applications is the development of an effective query optimizer. We propose an effective framework with a solid theoretical basis for optimizing OODB query languages. Our calculus, called the monoid comprehension calculus, captures most features of ODMG OQL and is a good basis for expressing various optimization algorithms concisely. This paper concentrates on query unnesting (also known as query decorrelation), an optimization that, even though improves performance considerably, is not treated properly (if at all) by most OODB systems. Our framework generalizes many unnesting techniques proposed recently in the literature and is capable of removing any form of query nesting using a very simple and efficient algorithm. The simplicity of our method is due to the use of the monoid comprehension calculus as an intermediate form for OODB queries. The monoid comprehension calculus treats operations over multiple collection types, aggregates, and quantifiers in a similar way, resulting in a uniform way of unnesting queries, regardless of their type of nesting.

Categories and Subject Descriptors: H.2.3 [**Information Systems**]: Database Management

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: Object-oriented databases, query optimization

---

This work is supported in part by the National Science Foundation under the grants IIS-9811525 and IRI-9619977, and by Texas Higher Education Advanced Research Program under the grant 003656-0043-1999.
Name: Leonidas Fegaras
Address: Department of Computer Science and Engineering, The University of Texas at Arlington, 416 Yates Street, P.O. Box 19015, Arlington, TX 76019-19015, email: *fegaras@cse.uta.edu*
Name: David Maier
Address: Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 20000 N.W. Walker Road, Beaverton, OR 97006, email: *maier@cse.ogi.edu*

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

1. INTRODUCTION

One of the reasons for the commercial success of the relational database systems is that they offer good performance to many business applications, mostly due to the use of sophisticated query processing and optimization techniques. However, many non-traditional database applications require more advanced data structures and more expressive operations than those provided by the relational model. Currently, there are two major approaches for addressing the needs of these applications [Carey and DeWitt 1996]. One is object-relational databases (ORDBs), which extend the relational model with object-oriented features, and the other is object-oriented databases (OODBs), which add database capabilities to object-oriented programming languages.

OODBs provide powerful data abstractions and modeling facilities but they usually lack a suitable framework for query processing and optimization. In addition, many early OODB systems lacked a declarative language for associative access to data. Instead, they used simple pointer chasing to perform object traversals, which does not leave many opportunities for optimization. There is an increasing number of commercial OODB systems today, such Poet, GemStone, ObjectStore, Objectivity, Jasmine, and Versant, that support a higher-lever query language interface (often along the lines of SQL). In addition, there is now an object data standard, called ODMG 3.0 [Cattell 2000], which supports a declarative higher-level object query language, called OQL.

One of the key factors for OODB systems to successfully compete with relational systems as well as to meet the performance requirements of many non-traditional applications is the development of an effective query optimizer. There are many aspects to the OODB query optimization problem that can benefit from the already proven relational query-optimization technology. However many key features of OODB languages present new and difficult problems not adequately addressed by this technology. These features include object identity, methods, encapsulation, subtype hierarchy, user-defined type constructors, large multimedia objects, multiple collection types, arbitrary nesting of collections, and nesting of query expressions.

We propose an effective framework with a solid theoretical basis for optimizing OODB query languages. We have chosen to concentrate on the OQL language of the ODMG 3.0 standard, which closely resembles the query language of the O2 OODBMS [Deux 1990]. We chose OQL because even though it is a small language, and hence easier to comprehend, it contains most of the language features that are showing up in other object query languages and proposed relational extensions, such as SQL3 [Beech 1993], now called SQL:1999 [Eisenberg and Melton 1999]. If we are able to handle OQL in full generality, we believe our work will be widely applicable to other query languages.

OQL queries in our framework are translated into a calculus format that serves as an intermediate form, and then are translated into a version of the nested relational algebra. We use both a calculus and an algebra as intermediate forms because the calculus closely resembles current OODB languages and is easy to put into canonical form, while the algebra is lower-level and can be directly translated into the execution algorithms supported by database systems.

Our calculus is called the *monoid comprehension calculus* [Fegaras and Maier 1995]. As we will demonstrate in this paper, our calculus captures most features of OQL and is a good basis for expressing various optimization algorithms concisely. It is based on monoids, a general template for a data type, which 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 the expressive power of our calculus by showing how to map the major features of OQL into the monoid calculus.

We will show in this paper that the monoid calculus is amenable to efficient evaluation. We give initial evidence by exhibiting a simple normalization system for putting expressions into a canonical form. The main contribution of this paper is the description of a framework for optimizing monoid comprehensions. We focus on a very important optimization problem, query unnesting (sometimes called query decorrelation), and propose a practical, effective, and general solution [Fegaras 1998b]. Our method generalizes many unnesting techniques proposed recently in the literature. Our framework is capable of removing *any* form of query nesting in our calculus using a very simple and efficient algorithm. The simplicity of our method is due to the use of the monoid comprehension calculus as an intermediate form for OODB queries. The monoid comprehension calculus treats operations over multiple collection types, aggregates, and quantifiers in a similar way, resulting in a uniform way of unnesting queries, regardless of their type of nesting. Our unnesting algorithm is compositional, that is, the translation of an embedded query does not depend on the context in which it is embedded; instead, each subquery is translated independently, and all translations are composed to form the final unnested query. This property makes the soundness and completeness of the algorithm easy to prove. Our unnesting algorithm is complete, efficient, and easy to implement. It is the first decorrelation algorithm developed for a non-trivial database language *proven* to be complete. In addition, we believe and we plan to show in a future work, that it can be adapted to handle object-relational and relational queries.

This paper is organized as follows. Section 2 describes the monoid comprehension calculus. Section 3 gives the translation of some OQL query forms into monoid comprehensions. Section 4 describes a simple normalization algorithm that unnests most simple forms of nesting that do not require outer-joins, outer-unnests, or grouping. Section 5 proposes extensions to the basic framework for alleviating restrictions about idempotent operations and for capturing advanced list operations, vectors, and arrays. Section 6 describes a version of the nested-relational algebra that supports aggregation, quantification, and the handling of null values (using outer-joins and outer-unnests). The semantics of these operations is given in terms of the monoid calculus. Section 7 presents transformation rules for unnesting OODB queries. Section 8 reports on the implementation of an ODMG-based system based on our theoretical framework and Section 9 evaluates the performance of our implementation using sample queries. Finally, Section 10 compares our work with related work and Section 11 describes our current and future plans related to this project.

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 indicate 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 $(\cup, \{\})$ and $(+\!\!+, [\,])$ are *monoids*, and, more specifically, $(\cup, \{\})$ 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 $(+, 0)$ and $(*, 1)$ are integer monoids and both $(\vee, \text{false})$ and $(\wedge, \text{true})$ are boolean monoids. As it will be apparent when monoid operations are described, the choice of a monoid depends on a particular interpretation of a primitive type in a query. We call the monoids for primitive types *primitive monoids*.

In contrast to primitive types, collection types, such as sets, bags, and lists, are parametric types that capture homogeneous collections of objects. The types $\text{set}(\langle \text{name} : \text{string}, \text{age} : \text{int} \rangle)$ and $\text{set}(\text{set}(\text{int}))$ are examples of collection types. We call the monoids for collection types *collection monoids*. Each collection monoid must have 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 constructed 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

$$\mathrm{hom}[+\!\!+,\cup](f)\,A,$$

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

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

In other words, $\mathrm{hom}[+\!\!+,\cup](f)\,A$ replaces $[\,]$ in A by $\{\}$, $+\!\!+$ by $\cup$, and the singleton list $[x]$ by $f(x)$. Therefore, 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 $\mathrm{hom}[+\!\!+,\cup](f)\,A$ can be decomposed into $(\mathrm{hom}[+\!\!+,\cup](f)\,A_1)\cup(\mathrm{hom}[+\!\!+,\cup](f)\,A_2)$.

Unfortunately, not all monoid homomorphisms are well-formed. For example, sets cannot be converted unambiguously into lists because of nondeterminism. This semantic restriction is purely syntactic in our framework. That is, it depends on the static properties of the monoids involved in a homomorphism, such as the commutativity and the idempotence properties. Thus well-formed queries will not result in any ill-defined homomorphisms.

The monoid homomorphism is the only form of bulk manipulation of collection types supported in our framework. 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 the nested relational algebra (since these homomorphisms are equivalent to the extension operator $ext(f)$ for sets, which has been shown to be at least as expressive as the nested relational algebra [Buneman et al. 1995; Breazu-Tannen et al. 1992]). But monoid homomorphisms go beyond that algebra to capture operations over multiple collection types, such as the join of a list with a bag returning a set, as well as predicates and aggregates. For example, an existential predicate over a set is a monoid homomorphism from the set monoid to the boolean monoid $(\vee,\mathrm{false})$, while an aggregation, such as summing all elements of a list, is a monoid homomorphism from the list monoid to the numeric monoid $(+,0)$.

We will also define a 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 by some 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

type $T_\oplus$	$\oplus$	$\mathcal{Z}_\oplus$	C/I
int	$+$	0	C
int	$\times$	1	C
int	max	0	CI
bool	$\vee$	false	CI
bool	$\wedge$	true	CI

A. Primitive Monoids

type $T_\oplus$	$\oplus$	$\mathcal{Z}_\oplus$	$\mathcal{U}_\oplus(a)$	C/I
set(α)	$\cup$	$\{\}$	$\{a\}$	CI
bag(α)	$\uplus$	$\{\!\{\}\!\}$	$\{\!\{a\}\!\}$	C
list(α)	$+\!\!+$	$[\,]$	$[a]$	

B. Collection Monoids

Table 1. Examples of Monoids

the tuple relational calculus, but here query variables may range over multiple collection types, while the output of the comprehension may be of a yet different collection type.

For example, the following monoid comprehension:

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

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 $\cup$):

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

Another example is

$$+\{\, a \,\|\, a \leftarrow [1,2,3],\, a \geq 2 \,\},$$

which uses the merge function $+$ of the monoid $(+,0)$ to add integers. 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 monoids and monoid operations.

2.1 Monoids

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

The type $T \times T \rightarrow T$ in the above definition denotes the type of binary functions that take two values of type T as input and return a value of type T. Function $\oplus$ is called the *merge* function and value $\mathcal{Z}_\oplus$ is called the *zero* element of the monoid $(\oplus, \mathcal{Z}_\oplus)$. According to the definition, the zero element satisfies $\mathcal{Z}_\oplus \oplus x = x \oplus \mathcal{Z}_\oplus = x$ for every x. A monoid $(\oplus, \mathcal{Z}_\oplus)$ may be a *commutative* monoid (i.e., when $\oplus$ is commutative) or an *idempotent* monoid (i.e., when $\forall x: x \oplus x = x$), or both. In our framework, if a monoid is not idempotent, it must be *anti-idempotent*: $\forall x \neq \mathcal{Z}_\oplus : x \oplus x \neq x$, and if it is not commutative, it must be anti-commutative: $\forall x \neq y,\, x \neq \mathcal{Z}_\oplus,\, y \neq \mathcal{Z}_\oplus :$ $x \oplus y \neq y \oplus x$, for reasons that will be discussed in Section 2.2. For example, $(+,0)$ is a commutative and anti-idempotent monoid, while $(\cup, \{\,\})$ is a commutative and idempotent monoid.

Since the merge function uniquely identifies a monoid, we will often use the merge function name as the monoid name. In addition, we will use the notation $T_\oplus$ to represent the type of the monoid with merge function $\oplus$.

Table 1 presents some examples of monoids. The C/I column indicates whether the monoid is a commutative or idempotent monoid (if a monoid is not tagged by

I, it is anti-idempotent, and if it is not tagged by C, it is anti-commutative). An example of an idempotent and anti-commutative monoid, which captures vectors and arrays, is presented in Section 5.3.

The monoids in Table 1.A are called *primitive monoids* because they construct values of a primitive type. The monoids $\cup$, $\uplus$, and $+\!\!+$ in Table 1.B capture the well-known collection types for sets, bags, and linear lists (where $\uplus$ is the additive union for bags). They are called *collection monoids*. Each collection monoid $(\oplus, \mathcal{Z}_\oplus)$ needs the additional definition of a *unit function*, $\mathcal{U}_\oplus$, which, along with merge and zero, allows the construction of all possible values of the monoid type $T_\oplus$. For example, the unit function for the set monoid is $\lambda x.\,\{x\}$, where expression $\lambda x.e$ denotes the function f with $f(x) = e$. That is, the set unit function takes a value x as input and constructs the singleton set $\{x\}$ as output. Consequently, a collection monoid is associated with a triple $(\oplus, \mathcal{Z}_\oplus, \mathcal{U}_\oplus)$ and corresponds to a parametric type $T_\oplus(\alpha)$, where α is a type parameter that can be bound to any type with equality, including another collection type[1].

We will use the shorthand $\oplus\{e_1, \ldots, e_n\}$ to represent the construction $\mathcal{U}_\oplus(e_1) \oplus \cdots \oplus \mathcal{U}_\oplus(e_n)$ over the collection monoid $\oplus$. In particular, we will use the shorthands $\{e_1, \ldots, e_n\}$ for sets, $\{\!\{e_1, \ldots, e_n\}\!\}$ for bags, and $[e_1, \ldots, e_n]$ for lists.

We define the mapping ψ from monoids to the powerset of $\{C, I\}$ as: $C \in \psi(\oplus)$ if and only if $\oplus$ is commutative and $I \in \psi(\oplus)$ if and only if $\oplus$ is idempotent. The partial order $\preceq$ between monoids is defined as:

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

For example, $+\!\!+ \preceq \uplus \preceq \cup$ since $\cup$ is commutative and idempotent, $\uplus$ is commutative but not idempotent, and $+\!\!+$ is neither commutative nor idempotent.

2.2 Monoid Homomorphisms

We now define our algebraic operator, which is parameterized by input and output monoids. The main purpose of this operator is to give formal semantics to monoid comprehensions, to be discussed in Section 2.3. Our monoid calculus is defined in terms of monoid comprehensions exclusively.

Definition 2 (Monoid Homomorphism). *A homomorphism* $\mathrm{hom}[\oplus, \otimes](f)\, A$ *from the collection monoid* $(\oplus, \mathcal{Z}_\oplus, \mathcal{U}_\oplus)$ *to any monoid* $(\otimes, \mathcal{Z}_\otimes)$*, where* $\oplus \preceq \otimes$*, is defined by the following inductive equations:*

$$\begin{aligned}
\mathrm{hom}[\oplus, \otimes](f)\,(\mathcal{Z}_\oplus) &= \mathcal{Z}_\otimes \\
\mathrm{hom}[\oplus, \otimes](f)\,(\mathcal{U}_\oplus(a)) &= f(a) \\
\mathrm{hom}[\oplus, \otimes](f)\,(x \oplus y) &= (\mathrm{hom}[\oplus, \otimes](f)\,(x)) \otimes (\mathrm{hom}[\oplus, \otimes](f)\,(y)).
\end{aligned}$$

That is, A is a collection of type $T_\oplus(T')$ for some type T' and f is a function from T' to $T_\otimes$. Basically, the expression $\mathrm{hom}[\oplus, \otimes](f)\, A$ replaces $\mathcal{Z}_\oplus$ in A by $\mathcal{Z}_\otimes$, $\oplus$ by

[1] Using terminology from abstract algebra, $(\oplus, \mathcal{Z}_\oplus, \mathcal{U}_\oplus)$ is a free monoid and $\mathcal{U}_\oplus$ is a natural transformation from the identity to the $T_\oplus$ functor that satisfies the universal mapping property (also known as uniqueness property or adjunction) that for any $f : \alpha \to T_\oplus(\alpha)$ there is a unique function $\mathrm{hom}[\oplus, \otimes](f)$ (given in Definition 2) such that $\mathrm{hom}[\oplus, \otimes](f) \circ \mathcal{U}_\oplus = f$. Function $\mathrm{hom}[\oplus, \otimes](f)$ is called the homomorphic extension of f. The interested reader is referred to the book by B. Pierce [1991] for a more formal treatment of monoids and monoid homomorphisms.

$\otimes$, and $\mathcal{U}_\oplus$ by f. For example, for $H = \mathrm{hom}[\mathbin{++},\cup](f)$, Definition 2 is equivalent to

$$\begin{array}{lcl} H([\,]) & = & \{\,\} \\ H([a]) & = & f(a) \\ H(x \mathbin{++} y) & = & H(x) \cup H(y). \end{array}$$

If A is the list $[a_1, \ldots, a_n]$, then $H(A)$ computes the set $f(a_1) \cup \cdots \cup f(a_n)$. For instance, if $f(x) = \{x+1\}$, then $H(A) = \{a_1+1, \ldots, a_n+1\}$.

The condition $\oplus \preceq \otimes$ in Definition 2 is important. If the collection monoid $\oplus$ is a commutative or idempotent monoid, then $\otimes$ must be too. For example, the bag cardinality function can be expressed as $\mathrm{hom}[\uplus,+](\lambda x.\,1)\,A$, which is well-formed, while the similar function for sets $\mathrm{hom}[\cup,+](\lambda x.\,1)\,A$ is not (since $+$ is commutative but not idempotent). Without this restriction we would have [Breazu-Tannen and Subrahmanyam 1991]:

$$\begin{array}{rcl} 1 & = & \mathrm{hom}[\cup,+](\lambda x.\,1)\,(\{a\}) \\ & = & \mathrm{hom}[\cup,+](\lambda x.\,1)\,(\{a\} \cup \{a\}) \\ & = & (\mathrm{hom}[\cup,+](\lambda x.\,1)\,(\{a\})) + (\mathrm{hom}[\cup,+](\lambda x.\,1)\,(\{a\})) \\ & = & 1+1 \;=\; 2. \end{array}$$

This restriction also prohibits the conversion of sets into lists (since $\cup \not\preceq \mathbin{++}$). Furthermore, Definition 2 justifies the restriction that non-commutative monoids should be anti-commutative and non-idempotent monoids should be anti-idempotent. If we allow a non-idempotent monoid $\oplus$ to satisfy $x \oplus x = x$ for at least one $x = x_0$ (but not for all x, since $\oplus$ is not idempotent), then we will have $\mathrm{hom}[\oplus,\otimes](f)\,(x_0) = \mathrm{hom}[\oplus,\otimes](f)\,(x_0 \oplus x_0) = (\mathrm{hom}[\oplus,\otimes](f)\,(x_0)) \otimes (\mathrm{hom}[\oplus,\otimes](f)\,(x_0))$, which is not always true for an arbitrary non-idempotent monoid $\otimes$. The observations above are generalized in the following theorem.

Theorem 1. *A well-formed monoid homomorphism* $\mathrm{hom}[\oplus,\otimes](f)$ *preserves the properties of the monoid* $\otimes$.

Proof. Let $H = \mathrm{hom}[\oplus,\otimes](f)$; then $\oplus \preceq \otimes$, since H is well-formed. We will prove that for any valid instance, x, of the monoid $\oplus$, the value $H(x)$ is a valid instance of $\otimes$. We consider idempotence only; commutativity can be handled in a similar way. If $\oplus$ is idempotent, then $H(x \oplus x) = H(x)$. Thus, $H(x) \otimes H(x) = H(x)$, which means that $\otimes$ is not anti-idempotent. This is true since $\oplus \preceq \otimes$, which implies that $\otimes$ is idempotent. If $\oplus$ is anti-idempotent, then $\otimes$ can be either idempotent or anti-idempotent. If $\otimes$ is anti-idempotent, then for $H(x) \neq \mathcal{Z}_\otimes$: $H(x) \otimes H(x) \neq H(x) \;\Rightarrow\; H(x \oplus x) \neq H(x)$, which implies that $x \oplus x \neq x$. If $\otimes$ is idempotent, then $H(x) \otimes H(x) = H(x) \;\Rightarrow\; H(x \oplus x) = H(x)$, which does not necessarily conflict with $x \oplus x \neq x$. $\square$

The following are examples of well-formed monoid homomorphisms:

$$\begin{array}{lcl} \mathrm{map}(f)\,x & = & \mathrm{hom}[\cup,\cup](\lambda a.\,\{f(a)\})\,x \\ x \times y & = & \mathrm{hom}[\cup,\cup](\lambda a.\,\mathrm{hom}[\cup,\cup](\lambda b.\,\{(a,b)\})\,y)\,x \\ e \in x & = & \mathrm{hom}[\cup,\vee](\lambda a.\,(a=e))\,x \\ \mathrm{filter}(p)\,x & = & \mathrm{hom}[\cup,\cup](\lambda a.\,\mathbf{if}\;p(a)\;\mathbf{then}\;\{a\}\;\mathbf{else}\;\{\})\,x \\ \mathrm{length}(x) & = & \mathrm{hom}[\mathbin{++},+](\lambda a.\,1)\,x, \end{array}$$

where $\text{map}(f)\,x$ 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 $\text{filter}(p)\,x$ selects all elements a of the set x that satisfy the predicate $p(a)$.

2.3 Monoid Comprehensions

Queries in our calculus are expressed in terms of monoid comprehensions. Informally, a monoid comprehension over the monoid $\oplus$ takes the form $\oplus\{\, e \,\|\, \overline{q} \,\}$. The merge function $\oplus$ is called the *accumulator* of the comprehension and the 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 *range variable* and e' is an expression (the generator domain) that constructs a collection, or
- a *filter* pred, where pred is a predicate.

Formally, monoid comprehensions are defined in terms of monoid homomorphisms:

DEFINITION 3 (MONOID COMPREHENSION). *A monoid comprehension over a primitive or collection monoid $\oplus$ is defined by the following inductive equations:*

$$\oplus\{\, e \,\|\, \,\} \;=\; \begin{cases} \mathcal{U}_\oplus(e) & \textit{for a collection monoid } \oplus \\ e & \textit{for a primitive monoid } \oplus \end{cases} \tag{D1}$$

$$\oplus\{\, e \,\|\, x \leftarrow u, \overline{q} \,\} \;=\; \text{hom}[\otimes, \oplus](\lambda x.\, \oplus\{\, e \,\|\, \overline{q} \,\})\, u \tag{D2}$$

$$\oplus\{\, e \,\|\, \text{pred}, \overline{q} \,\} \;=\; \textbf{if } \text{pred } \textbf{then } \oplus\{\, e \,\|\, \overline{q} \,\} \textbf{ else } \mathcal{Z}_\oplus \tag{D3}$$

where $\otimes$ is the collection monoid associated with the expression u and is possibly different from $\oplus$.

While the monoid $\oplus$ of the output is specified explicitly, the collection monoid $\otimes$ associated with the expression u in $x \leftarrow u$ can be inferred (by the typing rules to be given in Section 2.5). The main purpose of the definition above is to give semantics to monoid comprehensions, not to suggest in any way how they are implemented. Monoid comprehensions can be effectively translated into efficient evaluation algorithms, as we will show in Section 7.

Note that $+\!\!+\{\, x \,\|\, x \leftarrow \{1, 2\} \,\}$ is not a well-formed comprehension, since it is translated into a homomorphism from $\cup$ to $+\!\!+$ (it maps sets into lists), while $+\{\, x \,\|\, x \leftarrow \{\!\{1, 2\}\!\} \,\}$ is well-formed and computes $1 + 2 = 3$. That is, if one of the generators in a monoid comprehension is over a commutative or idempotent monoid, then the comprehension accumulator must be a commutative or idempotent monoid, respectively. This condition can be checked statically, since the commutativity and idempotence properties of a monoid are specified explicitly when this monoid is defined (see Section 2.5).

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

$$\cup\{\; \mathsf{f(a,b)} \;\|\; \mathsf{a} \leftarrow \mathsf{x},\ \mathsf{b} \leftarrow \mathsf{y},\ \mathsf{p(a,b)} \;\},$$

where p is the join predicate and function f constructs an output set element given two elements from x and y. For example, if $\mathsf{p(a,b)} = (\mathsf{a.C{=}b.C}) \wedge (\mathsf{a.D} > 10)$ and $\mathsf{f(a,b)} = \langle\ \mathsf{C{=}a.C},\ \mathsf{D{=}b.D}\ \rangle$ (i.e., a record construction), then this comprehension becomes:

$$\cup\{\!\!\{ \; \langle \; \mathsf{C{=}a.C,\ D{=}b.D} \; \rangle \; [\!] \; \mathsf{a} \leftarrow \mathsf{x},\ \mathsf{b} \leftarrow \mathsf{y},\ \mathsf{a.C{=}b.C},\ \mathsf{a.D} > 10 \; \}\!\!\}.$$

If we use the rules in Definition 3, this comprehension is translated into algebraic form as

$$\begin{array}{ll} \mathsf{hom}[\cup,\cup](\lambda \mathsf{a}.\ \mathsf{hom}[\cup,\cup](\lambda \mathsf{b}. & \textbf{if } (\mathsf{a.C{=}b.C}) \wedge (\mathsf{a.D} > 10) \\ & \textbf{then } \{\langle \; \mathsf{C{=}a.C,\ D{=}b.D} \; \rangle\} \\ & \textbf{else } \{\})\ \mathsf{y})\ \mathsf{x}. \end{array}$$

This evaluation resembles a nested loop, but we will show in Section 7 that comprehensions similar to the one above can be effectively translated into joins.

Furthermore, comprehensions can be used to join different collection types. For example,

$$\cup\{\!\!\{ \, (x, y) \, [\!] \, x \leftarrow [1, 2],\, y \leftarrow \{\!\{3, 4, 3\}\!\} \, \}\!\!\} \; = \; \{(1, 3), (1, 4), (2, 3), (2, 4)\}.$$

Another example is $\mathrm{nest}(k)\, x$ equal to

$$\cup\{\!\!\{ \, \langle \mathrm{KEY} = k(e),\, \mathrm{INFO} = \cup\{\!\!\{ \, a \, [\!] \, a \leftarrow x,\, k(e) = k(a) \, \}\!\!\} \rangle \, [\!] \, e \leftarrow x \, \}\!\!\},$$

which is similar to the nesting operator for nested relations, but it is a bit more general and gives output in a different form. Similarly, the inverse function, unnest, is

$$\mathrm{unnest}(x) \; = \; \cup\{\!\!\{ \, e \, [\!] \, s \leftarrow x,\, e \leftarrow s.\mathrm{INFO} \, \}\!\!\}.$$

The last comprehension is an example of a *dependent join* in which the value of the collection s.INFO in the second generator 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 of comprehensions are the following:

$$\begin{array}{lll@{\qquad\qquad}lll} \mathrm{filter}(p)\, x & = & \cup\{\!\!\{ \, e \, [\!] \, e \leftarrow x,\, p(e) \, \}\!\!\} & \mathrm{count}(x, a) & = & +\{\!\!\{ \, 1 \, [\!] \, e \leftarrow x,\, e = a \, \}\!\!\} \\ \mathrm{flatten}(x) & = & \cup\{\!\!\{ \, e \, [\!] \, s \leftarrow x,\, e \leftarrow s \, \}\!\!\} & \mathrm{maximum}(x) & = & max\{\!\!\{ \, e \, [\!] \, e \leftarrow x \, \}\!\!\} \\ x \cap y & = & \cup\{\!\!\{ \, e \, [\!] \, e \leftarrow x,\, e \in y \, \}\!\!\} & \exists a \in x : e & = & \vee\{\!\!\{ \, e \, [\!] \, a \leftarrow x \, \}\!\!\} \\ \mathrm{length}(x) & = & +\{\!\!\{ \, 1 \, [\!] \, e \leftarrow x \, \}\!\!\} & \forall a \in x : e & = & \wedge\{\!\!\{ \, e \, [\!] \, a \leftarrow x \, \}\!\!\} \\ \mathrm{sum}(x) & = & +\{\!\!\{ \, e \, [\!] \, e \leftarrow x \, \}\!\!\} & a \in x & = & \vee\{\!\!\{ \, a = e \, [\!] \, e \leftarrow x \, \}\!\!\}. \end{array}$$

The expression $\mathrm{sum}(x)$ adds the elements of any non-idempotent monoid x, e.g., $\mathrm{sum}([1, 2, 3]) = 6$. The expression $\mathrm{count}(x, a)$ counts the number of occurrences of a in the bag x, e.g., $\mathrm{count}(\{\!\{1, 2, 1\}\!\}, 1) = 2$. Recall that the $(max, 0)$ monoid is both commutative and idempotent, and thus x in $\mathrm{maximum}(x)$ can be of any collection type.

Theorem 2. *Monoid comprehensions and monoid homomorphisms have the same expressive power.*

Proof. Definition 3 expresses comprehensions in terms of monoid homomorphisms. Thus, we need to prove that any monoid homomorphism can be expressed in terms of monoid comprehensions. In particular, we will prove the equations $\mathrm{hom}[\oplus, \otimes](f)\, (A) = \otimes\{\!\!\{ \, y \, [\!] \, x \leftarrow A,\, y \leftarrow f(x) \, \}\!\!\}$ for a collection monoid $\otimes$ and $\mathrm{hom}[\oplus, \otimes](f)\, (A) \; = \; \otimes\{\!\!\{ \, f(x) \, [\!] \, x \leftarrow A \, \}\!\!\}$ for a primitive monoid $\otimes$. The universal property for homomorphisms indicates that, for a collection monoid $\otimes$, $\forall x \; : \; \mathrm{hom}[\otimes, \otimes](\mathcal{U}_\otimes)\, x \; = \; x$. According to Definition 3, for a collection monoid

```
class Person ( extent Persons key ssn )
{   attribute long ssn;
    attribute string name;
    attribute struct( street: string, zipcode: string ) address;
};
class Instructor extends Person ( extent Instructors )
{   attribute long salary;
    attribute string rank;
    attribute set⟨string⟩ degrees;
    relationship Department dept inverse Department::instructors;
    relationship set⟨Course⟩ teaches inverse Course::taught_by;
};
class Department ( extent Departments keys dno, name )
{   attribute long dno;
    attribute string name;
    attribute Instructor head;
    relationship set⟨Instructor⟩ instructors inverse Instructor::dept;
    relationship set⟨Course⟩ courses_offered inverse Course::offered_by;
};
class Course ( extent Courses keys code, name )
{   attribute string code;
    attribute string name;
    relationship Department offered_by inverse Department::courses_offered;
    relationship Instructor taught_by inverse Instructor::teaches;
    relationship set⟨Course⟩ is_prerequisite_for inverse Course::has_prerequisites;
    relationship set⟨Course⟩ has_prerequisites inverse Course::is_prerequisite_for;
};
```

Fig. 1. The ODL Schema of the University Database

$\otimes$, $\otimes\{\, y \,\|\, x \leftarrow A,\, y \leftarrow f(x) \,\} \;=\; \mathrm{hom}[\oplus,\otimes](\lambda x.\,\mathrm{hom}[\otimes,\otimes](\lambda y.\,\mathcal{U}_{\otimes}(y))\,(f(x)))\,(A) \;=$ $\mathrm{hom}[\oplus,\otimes](f)\,(A)$, which proves the first equation. For a primitive monoid $\otimes$, $\otimes\{\, y \,\|\, x \leftarrow A,\, y \leftarrow f(x) \,\} \;=\; \mathrm{hom}[\oplus,\otimes](\lambda x.\,\mathrm{hom}[\otimes,\otimes](\lambda y.\,y)\,(f(x)))\,(A)$, which in turn is equal to $\mathrm{hom}[\oplus,\otimes](f)\,(A)$, which proves the second equation. $\square$

2.4 Monoid Types

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

Definition 4 (Monoid Type). *A* monoid type *takes one of the following forms:*

class_name	(a reference to a user-defined class)
T	(a primitive type, such as int or bool)
$T(t)$	(a collection type, such as list(int))
$\langle A_1 : t_1, \ldots, A_n : t_n \rangle$	(a record type)

where t, t_1, …, t_n *are monoid types.*

Thus, monoid types can be freely nested.

Our type system is rich enough to capture most ODMG ODL data types. We will use the University schema shown in Figure 1 as a running example for our queries. Note that the class Instructor is a subclass of the class Person. ODL relationships can be 1:1, 1:N, or N:M. When used in OQL queries, a binary relationship between

NULL	null value
c	constant
v	variable
$e.A$	record projection
$\langle A_1 = e_1, \ldots, A_n = e_n \rangle$	record construction
$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	if-then-else statement
$e_1\ \mathrm{op}\ e_2$	where op is a primitive binary function, such as $+, =, <, >$
$\lambda v : t.\, e$	function abstraction
$e_1(e_2)$	function application
$\mathcal{Z}_\oplus$	zero element
$\mathcal{U}_\oplus(e)$	singleton construction
$e_1 \oplus e_2$	merging
$\oplus\{\, e \,[\!]\, q_1, \ldots, q_n \,\}$	comprehension

Fig. 2. The Monoid Comprehension Calculus

the classes A and B offers two ways of relating A and B objects: given an instance of A, to retrieve the related instance(s) of B, and given an instance of B, to retrieve the related instance(s) of A. A class extent, such as the extent Instructors for the class Instructor, is a set of all persistent instances of this class. Extents are the only entry points to the database. We assume persistence by reachability, which indicates that, not only the members of an extent, but also all the objects that can be reached from these members, are persistent. In our treatment of types, relationships are handled as attributes and extents as sets of class references, such as Courses: set⟨Course⟩. A class itself is treated as an aggregation (a record) of its attributes.

2.5 The Monoid Comprehension Calculus

Our monoid comprehension calculus consists of a number of syntactic forms that operate over monoid types. These syntactic forms can be composed to construct yet more complicated terms. Not every combination of syntactic forms is well-formed though. Ill-formed expressions can be detected and terms can be assigned unique types with the help of a typing system, which is usually specified by a set of typing rules. We first present the monoid calculus and then the typing rules of the calculus.

Definition 5 (Monoid Comprehension Calculus). *The monoid calculus consists of the syntactic forms in Figure 2, where e, $e_1, \ldots, e_n$ are terms in the monoid calculus, v is a variable, t is a monoid type, and $q_1, \ldots, q_n$ are qualifiers of the form $v \leftarrow e$ or e.*

For example, the following expression:

$$\begin{aligned}\cup\{\ & \mathsf{e.name}\ [\!]\ \mathsf{el} \leftarrow \uplus\{\ \mathsf{d.instructors}\ [\!]\ \mathsf{d} \leftarrow \mathsf{Departments},\ \mathsf{d.name}{=}\text{“CSE”}\ \},\\ & \mathsf{e} \leftarrow \mathsf{el},\ \vee\{\ \mathsf{c.name}{=}\text{“cse5331”}\ [\!]\ \mathsf{c} \leftarrow \mathsf{e.teaches}\ \}\ \},\end{aligned}$$

which is based on our example OODB schema, is a valid expression in the calculus. It returns all CSE instructors who teach cse5331. For each department with name CSE, the inner bag comprehension returns all instructors in the department. Thus, the result of the bag comprehension is of type bag(bag(Instructor)). Since variable

$$\sigma \vdash c : \text{ type of the constant } c \qquad \text{(T1)}$$

$$\sigma \vdash \text{NULL} : \alpha \qquad \text{(T2)}$$

$$\sigma \vdash v : \sigma(v) \qquad \text{(T3)}$$

$$\frac{\sigma \vdash e : \langle A_1 : t_1, \ldots, A_n : t_n \rangle}{\sigma \vdash e.A_i : t_i} \qquad \text{(T4)}$$

$$\frac{\sigma \vdash e_1 : t_1, \ldots, \sigma \vdash e_n : t_n}{\sigma \vdash \langle a_1 = e_1, \ldots, a_n = e_n \rangle : \langle a_1 : t_1, \ldots, a_n : t_n \rangle} \qquad \text{(T5)}$$

$$\frac{\sigma \vdash e_1 : \text{bool},\ \sigma \vdash e_2 : t,\ \sigma \vdash e_3 : t}{\sigma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : t} \qquad \text{(T6)}$$

$$\frac{\sigma[t_1/v] \vdash e : t_2}{\sigma \vdash \lambda v \!:\! t_1 .\, e : t_1 \rightarrow t_2} \qquad \text{(T7)}$$

$$\frac{\sigma \vdash e_1 : t_1 \rightarrow t_2,\ \sigma \vdash e_2 : t_1}{\sigma \vdash e_1(e_2) : t_2} \qquad \text{(T8)}$$

$$\frac{\oplus \text{ is collection monoid}}{\sigma \vdash \mathcal{Z}_{\oplus} : T_{\oplus}(\alpha)} \qquad \text{(T9)}$$

$$\frac{\sigma \vdash e : t}{\sigma \vdash \mathcal{U}_{\oplus}(e) : T_{\oplus}(t)} \qquad \text{(T10)}$$

$$\frac{\oplus \text{ is primitive monoid} \quad \sigma \vdash e_1 : T_{\oplus},\ \sigma \vdash e_2 : T_{\oplus}}{\sigma \vdash e_1 \oplus e_2 : T_{\oplus}} \qquad \text{(T11)}$$

$$\frac{\oplus \text{ is collection monoid} \quad \sigma \vdash e_1 : T_{\oplus}(t),\ \sigma \vdash e_2 : T_{\oplus}(t)}{\sigma \vdash e_1 \oplus e_2 : T_{\oplus}(t)} \qquad \text{(T12)}$$

$$\frac{\sigma \vdash e : T_{\oplus},\ \oplus \text{ is primitive monoid}}{\sigma \vdash \oplus\{\, e \,[\!]\, \} : T_{\oplus}} \qquad \text{(T13)}$$

$$\frac{\sigma \vdash e : t,\ \oplus \text{ is collection monoid}}{\sigma \vdash \oplus\{\, e \,[\!]\, \} : T_{\oplus}(t)} \qquad \text{(T14)}$$

$$\frac{\sigma \vdash e_2 : T_{\otimes}(t_2),\ \otimes \preceq \oplus, \quad \sigma[t_2/v] \vdash \oplus\{\, e_1 \,[\!]\, \overline{r} \,\} : t_1}{\sigma \vdash \oplus\{\, e_1 \,[\!]\, v \leftarrow e_2, \overline{r} \,\} : t_1} \qquad \text{(T15)}$$

$$\frac{\sigma \vdash e_2 : \text{bool},\ \sigma \vdash \oplus\{\, e_1 \,[\!]\, \overline{r} \,\} : t}{\sigma \vdash \oplus\{\, e_1 \,[\!]\, e_2, \overline{r} \,\} : t} \qquad \text{(T16)}$$

Fig. 3. The Typing Rules of the Monoid Calculus

el ranges over the result of the bag comprehension, the type of el is bag(Instructor). Variable e ranges over el, which indicates that the type of e is Instructor. The $\vee$-comprehension captures an existential quantification; it returns true if there is at least one course c in e.teaches that has c.name="cse5331".

We use the shorthand $x \equiv u$ to represent the binding of the variable x with the value u. The meaning of this construct is given by the following reduction rule:

$$\oplus\{\, e \,[\!]\, \overline{r},\ x \equiv u,\ \overline{s} \,\} \longrightarrow \oplus\{\, e[u/x] \,[\!]\, \overline{r},\ \overline{s}[u/x] \,\}, \qquad \text{(N1)}$$

where $e[u/x]$ is the expression e with u substituted for all the free occurrences of x. Another shorthand is the pair (x, y), which is equivalent to the record $\langle \text{fst} = x, \text{snd} = y \rangle$. In addition, as syntactic sugar, we allow irrefutable patterns in place of lambda variables, range variables, and variables in bindings. In functional programming languages [Peyton Jones 1987], a pattern is irrefutable if it is either a variable or a tuple consisting of irrefutable patterns. Patterns such as these can be compiled away using standard pattern decomposition techniques [Peyton Jones 1987]. For example, $\cup\{\, x + y \,[\!]\, (x, (y, z)) \leftarrow A,\ z = 3 \,\}$ is equivalent to $\cup\{\, a.\text{fst} + a.\text{snd}.\text{fst} \,[\!]\, a \leftarrow A,\ a.\text{snd}.\text{snd} = 3 \,\}$.

Figure 3 gives the typing rules of the terms in the monoid calculus. The name v in Figure 3 indicates a variable name, names starting with A are attribute names, names starting with e represent terms in our calculus, α is a type variable, and names starting with t represent types. The notation $\sigma \vdash e : t$ indicates that the term e is assigned the type t under the substitution σ. If a type equation is a fraction, the numerator is the premise while the denominator is the consequence. For example, Equation (T4) indicates that if e is a tuple of type $\langle A_1 : t_1, \ldots, A_n : t_n \rangle$, then the type of $e.A_i$ is the ith component of the tuple type, t_i. The substitution list σ binds variable names to types ($\sigma(v)$ returns the binding of v in σ and $\sigma[t/v]$ extends σ with the binding from v to t). The λ-variable, v, in Equation (T7) is annotated with its type, t_1. This annotation is omitted in our examples because it can be inferred from context in most cases. This equation indicates that, if the type of e is t_2 under the assumption that the type of v is t_1, then the type of $\lambda v : t_1 . e$ is $t_1 \rightarrow t_2$ (a function from t_1 to t_2). Equation (T15) checks the validity of a comprehension by testing whether $\otimes \preceq \oplus$. Note that, according to these typing rules, the term $1 + 2 \times 3$ is type correct, since $T_+ = T_\times = \text{int}$ (from Table 1.A), while $x \mathbin{++} y \cup z$ is not type correct.

3. TRANSLATING OQL TO THE MONOID CALCULUS

Nearly all OQL expressions have a direct translation into the monoid calculus, with the exception of indexed OQL collections. (Section 5 presents 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 into the monoid calculus.

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

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

is translated into:

$$\uplus\{\, e \,[\!]\, x_1 \leftarrow e_1, \,\ldots, \,x_n \leftarrow e_n, \,\mathit{pred} \,\}.$$

Note that e, 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 e.name, e.address
 from el in (select d.instructors
               from d in Departments
              where d.name="CSE"),
         e in el
where exists c in e.teaches: c.name="cse5331"
```

is expressed in the comprehension syntax as follows:

$$\begin{array}{l} \uplus\{\ \langle\ \text{Name: e.name, Address: e.address}\ \rangle \\ \quad [\!]\ \text{el} \leftarrow \uplus\{\ \text{d.instructors}\ [\!]\ \text{d} \leftarrow \text{Departments, d.name=“CSE”}\ \}, \\ \quad\quad \text{e} \leftarrow \text{el}, \vee\{\ \text{c.name=“cse5331”}\ [\!]\ \text{c} \leftarrow \text{e.teaches}\ \}\ \}. \end{array}$$

The **select-distinct** OQL statement is translated into a set comprehension in a similar way. For example, the following query against the University schema finds all departments that have at least one instructor who teaches more than 8 courses per year:

```
select distinct e.dept.name
  from e in Instructors
 where count(e.teaches) > 8.
```

This query is translated into the following comprehension:

$$\cup\{\ \mathsf{e.dept.name}\ \rrbracket\ \mathsf{e} \leftarrow \mathsf{Instructors},\ +\{\ 1\ \rrbracket\ \mathsf{c} \leftarrow \mathsf{e.teaches}\ \} > 8\ \}.$$

OQL universal and existential quantifications are mapped into comprehensions too, as it is shown in the following query that finds the instructors who teach courses which all have cse2320 as a prerequisite:

```
select e.name
  from e in Instructors
 where for all c in e.teaches:
         exists d in c.has_prerequisites: d.name = "cse2320".
```

This query is translated into the following comprehension:

$$\uplus\{\ \mathsf{e.name}\ \rrbracket\ \mathsf{e} \leftarrow \mathsf{Instructors}, \wedge\{\ \vee\{\ \mathsf{d.name}{=}\text{“cse2320”}\ \rrbracket\ \mathsf{d} \leftarrow \mathsf{c.has\_prerequisites}\ \}\ \rrbracket\ \mathsf{c} \leftarrow \mathsf{e.teaches}\ \}\ \}.$$

The OQL group-by syntax is essentially syntactic sugar, since the same results can be accomplished with nested queries. For example, the query

```
   select dept: dname, largest: max(e.salary)
     from e in Instructors
 group by dname: e.dept.name,
```

which, for each department, it prints the largest salary of all instructors working in the department, is equivalent to

```
select dept: e.dept.name,
       largest: max(select s.salary
                       from s in Instructors
                      where e.dept.name = s.dept.name)
  from e in Instructors,
```

which, in turn, can be expressed in the calculus as follows:

$$\uplus\{\ \langle\ \mathsf{dept}\colon \mathsf{e.dept.name},\ \mathsf{largest}\colon max\{\ \mathsf{s.salary}\ \rrbracket\ \mathsf{s} \leftarrow \mathsf{Instructors},\ \mathsf{e.dept.name}{=}\mathsf{s.dept.name}\ \}\ \rangle\ \rrbracket\ \mathsf{e} \leftarrow \mathsf{Instructors}\ \}.$$

We will see in Section 7 that nested query forms such as these can be translated into efficient execution plans that make use of group-by algorithms. In general, a group-by OQL query takes the following form:

$$\begin{array}{rl}
\textbf{select} & e(A_1,\ldots,A_n,\mathsf{partition}) \\
\textbf{from} & x_1\ \textbf{in}\ u_1,\ x_2\ \textbf{in}\ u_2(x_1),\ \ldots,\ x_m\ \textbf{in}\ u_m(x_1,\ldots,x_{m-1}) \\
\textbf{where} & p(x_1,\ldots,x_m) \\
\textbf{group by} & A_1 : e_1(x_1,\ldots,x_m),\ldots,A_n : e_n(x_1,\ldots,x_m) \\
\textbf{having} & h(A_1,\ldots,A_n,\mathsf{partition}),
\end{array}$$

where the variable *partition* is bound to the set of all groups, where each group is a record of all x_i values that have the same $A_1,\ldots,A_n$ values (it is used for complex aggregations). This OQL form is equivalent to:

$$\begin{aligned}
\uplus\{\, & e(A_1,\ldots,A_n,\text{partition}) \\
& \,\|\, x_1 \leftarrow u_1,\, x_2 \leftarrow u_2(x_1),\, \ldots,\, x_m \leftarrow u_m(x_1,\ldots,x_{m-1}), \\
& \;A_1 \equiv e_1(x_1,\ldots,x_m),\, \ldots,\, A_n \equiv e_n(x_1,\ldots,x_m), \\
& \;\text{partition} \equiv \uplus\{\, \langle x_1 : y_1,\, \ldots,\, x_m : y_m \rangle \\
& \qquad\qquad\qquad \,\|\, y_1 \leftarrow u_1,\, y_2 \leftarrow u_2(y_1),\, \ldots,\, y_m \leftarrow u_m(y_1,\ldots,y_{m-1}), \\
& \qquad\qquad\qquad\;\; e_1(y_1,\ldots,y_m) = A_1,\ldots,e_n(y_1,\ldots,y_m) = A_n \,\}, \\
& \;p(x_1,\ldots,x_m),\, h(A_1,\ldots,A_n,\text{partition}) \,\}.
\end{aligned}$$

For example, the query

```
   select salary: max(e.salary), dept: dname
     from e in Instructors
 group by dname: e.dept.name,
          teaches: count(select *
                           from c in e.teaches
                          where c.offered_by.name = "CSE")
```

is equivalent to:

$$\begin{aligned}
\uplus\{\, & \langle\, \text{salary: } max\{\, \text{p.e.salary} \,\|\, \text{p} \leftarrow \text{partition} \,\},\, \text{dept: dname} \,\rangle \\
& \,\|\, \text{e} \leftarrow \text{Instructors}, \\
& \;\text{dname} \equiv \text{e.dept.name}, \\
& \;\text{teaches} \equiv +\{\, 1 \,\|\, \text{c} \leftarrow \text{e.teaches, c.offered_by.name=“CSE”} \,\} \\
& \;\text{partition} \equiv \uplus\{\, \langle\, \text{e: s} \,\rangle \,\|\, \text{s} \leftarrow \text{Instructors}, \\
& \qquad\qquad\qquad\qquad \text{s.dept.name=dname}, \\
& \qquad\qquad\qquad\qquad +\{\, 1 \,\|\, \text{c} \leftarrow \text{s.teaches, c.offered_by.name=“CSE”} \,\} \\
& \qquad\qquad\qquad\qquad\qquad = \text{teaches} \,\}\,\}.
\end{aligned}$$

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

$$\begin{array}{lcl}
e_1 \textbf{ intersect } e_2 & \longrightarrow & \cup\{\, x \,\|\, x \leftarrow e_1,\, x \in e_2 \,\} \\
\textbf{for all } x \textbf{ in } e : \textit{ pred} & \longrightarrow & \wedge\{\, \textit{pred} \,\|\, x \leftarrow e \,\} \\
\textbf{exists } x \textbf{ in } e : \textit{ pred} & \longrightarrow & \vee\{\, \textit{pred} \,\|\, x \leftarrow e \,\} \\
e_1 \textbf{ in } e_2 & \longrightarrow & \vee\{\, x = e_1 \,\|\, x \leftarrow e_2 \,\} \\
\textbf{count}(e) & \longrightarrow & +\{\, 1 \,\|\, x \leftarrow e \,\} \\
\textbf{sum}(e) & \longrightarrow & +\{\, x \,\|\, x \leftarrow e \,\} \\
\textbf{flatten}(e) & \longrightarrow & \cup\{\, x \,\|\, s \leftarrow e,\, x \leftarrow s \,\}.
\end{array}$$

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. Nevertheless, there are OQL language constructs that cannot be simulated in the monoid calculus. They include bag difference and set or bag element (which retrieves an element from a singleton set or bag, respectively).

4. PROGRAM NORMALIZATION

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

$$(\lambda v.e_1)\, e_2 \;\longrightarrow\; e_1[e_2/v] \qquad \text{beta reduction} \tag{N2}$$

$$\langle A_1 = e_1, \ldots, A_n = e_n \rangle.A_i \;\longrightarrow\; e_i \tag{N3}$$

$$\begin{aligned}\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow (\textbf{if}\; e_1 \;\textbf{then}\; e_2 \;\textbf{else}\; e_3),\, \overline{s}\,\} \;\longrightarrow\;& (\oplus\{\, e \,\|\, \overline{q},\, e_1,\, v \leftarrow e_2,\, \overline{s}\,\}) \\ &\oplus (\oplus\{\, e \,\|\, \overline{q},\, \neg e_1,\, v \leftarrow e_3,\, \overline{s}\,\}) \\ &\qquad \text{for commutative } \oplus \text{ or empty } \overline{q}\end{aligned} \tag{N4}$$

$$\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow \mathcal{Z}_\otimes,\, \overline{s}\,\} \;\longrightarrow\; \mathcal{Z}_\oplus \tag{N5}$$

$$\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow \mathcal{U}_\otimes(e'),\, \overline{s}\,\} \;\longrightarrow\; \oplus\{\, e \,\|\, \overline{q},\, v \equiv e',\, \overline{s}\,\} \tag{N6}$$

$$\begin{aligned}\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow (e_1 \otimes e_2),\, \overline{s}\,\} \;\longrightarrow\;& (\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow e_1,\, \overline{s}\,\}) \;\oplus\; (\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow e_2,\, \overline{s}\,\}) \\ &\qquad \text{for commutative } \oplus \text{ or empty } \overline{q}\end{aligned} \tag{N7}$$

$$\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow \otimes\{\, e' \,\|\, \overline{r}\,\},\, \overline{s}\,\} \;\longrightarrow\; \oplus\{\, e \,\|\, \overline{q},\, \overline{r},\, v \equiv e',\, \overline{s}\,\} \tag{N8}$$

$$\oplus\{\, e \,\|\, \overline{q},\, \vee\{\, \mathit{pred} \,\|\, \overline{r}\,\},\, \overline{s}\,\} \;\longrightarrow\; \oplus\{\, e \,\|\, \overline{q},\, \overline{r},\, \mathit{pred},\, \overline{s}\,\} \qquad \text{for idempotent } \oplus \tag{N9}$$

$$\oplus\{\, \oplus\{\, e \,\|\, \overline{r}\,\} \,\|\, \overline{s}\,\} \;\longrightarrow\; \oplus\{\, e \,\|\, \overline{s},\, \overline{r}\,\} \qquad \text{for a primitive monoid } \oplus \tag{N10}$$

Fig. 4. The Normalization Algorithm

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 and fusing two selections into one selection. While normalization does not perform cost-based optimization, it does provide a good starting point for it.

Figure 4 gives the normalization rules. For example, Rule (N5) applies when a comprehension has a generator whose domain is a zero value (i.e., $\mathcal{Z}_\otimes$) that appears between two possibly empty sequences of generators $\overline{q}$ and $\overline{s}$. In that case, the comprehension is normalized into the value $\mathcal{Z}_\oplus$, since no values are generated for the variable v. Rule (N8) is the most important: it flattens a nested comprehension (i.e., a comprehension that contains a generator whose domain is another comprehension). Rule (N9) unnests an existential quantification. There are other cases of query unnesting, which will be handled in Section 7.

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.

Theorem 3. *The normalization rules in Figure 4 are meaning preserving.*

A proof sketch of this theorem is given in Appendix A. In addition, this set of rewrite rules is terminating and confluent (i.e., it satisfies the Church-Rosser property). That is, it does not matter in which sequence rules are applied to a term, as all sequences result in the same canonical form at the end. The rewrite rules are terminating because they decrease the number of nesting levels. They are confluent as it can be seen from Rule N8:

$$\begin{aligned}&\oplus\{\, e \,\|\, \overline{q},\, v_1 \leftarrow \otimes\{\, e_1 \,\|\, \overline{r_1}\,\},\, \overline{r},\, v_2 \leftarrow \otimes\{\, e_2 \,\|\, \overline{r_2}\,\},\, \overline{s}\,\}\\ &= \oplus\{\, e \,\|\, \overline{q},\, \overline{r_1},\, v_1 \equiv e_1,\, \overline{r},\, v_2 \leftarrow \otimes\{\, e_2 \,\|\, \overline{r_2}\,\},\, \overline{s}\,\}\\ &= \oplus\{\, e \,\|\, \overline{q},\, \overline{r_1},\, v_1 \equiv e_1,\, \overline{r},\, \overline{r_2},\, v_2 \equiv e_2,\, \overline{s}\,\},\end{aligned}$$

which is also the result of $\oplus\{\, e \,\|\, \overline{q},\, v_1 \leftarrow \otimes\{\, e_1 \,\|\, \overline{r_1}\,\},\, \overline{r},\, \overline{r_2},\, v_2 \equiv e_2,\, \overline{s}\,\}$. This identity indicates that both chains of rewrites (left inner comprehension first or right inner comprehension first) lead to the same reduced form.

Rules (N8) and (N9) 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:

$$\begin{aligned}&\cup\{\ \mathsf{a}\ \|\ \mathsf{a} \leftarrow \cup\{\ \mathsf{a}\ \|\ \mathsf{a} \leftarrow \mathsf{x},\ \mathsf{q(a)}\ \},\ \mathsf{p(a)}\ \}\\ &= \cup\{\ \mathsf{a}\ \|\ \mathsf{a} \leftarrow \cup\{\ \mathsf{b}\ \|\ \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)}\ \},\ \mathsf{p(a)}\ \}\end{aligned}$$

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

$$\begin{aligned}&\cup\{\ \mathsf{a}\ \|\ \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)},\ \mathsf{a}{\equiv}\mathsf{b},\ \mathsf{p(a)}\ \}\\ &\rightarrow \cup\{\ \mathsf{b}\ \|\ \mathsf{b} \leftarrow \mathsf{x},\ \mathsf{q(b)},\ \mathsf{p(b)}\ \}\end{aligned}$$

(by Rules (N8) and (N1)), 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:

$$\cup\{\ \mathsf{r}\ \|\ \mathsf{r} \leftarrow \mathsf{R},\ \vee\{\ \mathsf{x{=}r.B}\ \|\ \mathsf{x} \leftarrow \cup\{\ \mathsf{s.D}\ \|\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C}\ \}\ \}\ \}$$

and it is normalized into:

$$\begin{aligned}&\cup\{\ \mathsf{r}\ \|\ \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{x} \leftarrow \cup\{\ \mathsf{s.D}\ \|\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C}\,\},\ \mathsf{x{=}r.B}\ \}\\ &\rightarrow \cup\{\ \mathsf{r}\ \|\ \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C},\ \mathsf{x}{\equiv}\mathsf{s.D},\ \mathsf{x{=}r.B}\ \}\\ &\rightarrow \cup\{\ \mathsf{r}\ \|\ \mathsf{r} \leftarrow \mathsf{R},\ \mathsf{s} \leftarrow \mathsf{S},\ \mathsf{r.C{=}s.C},\ \mathsf{s.D{=}r.B}\ \}\end{aligned}$$

(by Rules (N9), (N8), and (N1)), which is equivalent to the following OQL query:

select distinct r
  from r **in** R, s **in** S
  where r.C=s.C **and** s.D = r.B.

Note that, although we were able to rewrite this unnested form back into OQL, not every comprehension has a direct translation back to OQL.

Consider now the query, presented in Section 3, which finds all CSE instructors who teach cse5331:

$$\begin{aligned}&\uplus\{\ \langle\ \mathsf{Name{:}\ e.name,\ Address{:}\ e.address}\ \rangle\\ &\quad \|\ \mathsf{el} \leftarrow \uplus\{\ \mathsf{d.instructors}\ \|\ \mathsf{d} \leftarrow \mathsf{Departments},\ \mathsf{d.name{=}``CSE"}\ \},\\ &\qquad \mathsf{e} \leftarrow \mathsf{el},\ \vee\{\ \mathsf{c.name{=}``cse5331"}\ \|\ \mathsf{c} \leftarrow \mathsf{e.teaches}\ \}\ \}.\end{aligned}$$

This query is normalized into:

$$
\begin{array}{l}
\uplus\{\!\{ \; \langle \; \mathsf{Name: h.name, Address: h.address} \; \rangle \\
\quad [\![\; \mathsf{d} \leftarrow \mathsf{Departments, d.name{=}``CSE", el{\equiv}d.instructors,} \\
\quad\quad \mathsf{e} \leftarrow \mathsf{el}, \vee\{\!\{ \; \mathsf{c.name{=}``cse5331"} \; [\![\; \mathsf{c} \leftarrow \mathsf{e.teaches} \; \}\!\} \; \}\!\} \\
\rightarrow \uplus\{\!\{ \; \langle \; \mathsf{Name: h.name, Address: h.address} \; \rangle \\
\quad\quad [\![\; \mathsf{d} \leftarrow \mathsf{Departments, d.name{=}``CSE",} \\
\quad\quad\quad \mathsf{e} \leftarrow \mathsf{d.instructors}, \vee\{\!\{ \; \mathsf{c.name{=}``cse5331"} \; [\![\; \mathsf{c} \leftarrow \mathsf{e.teaches} \; \}\!\} \; \}\!\} \\
\rightarrow \uplus\{\!\{ \; \langle \; \mathsf{Name: h.name, Address: h.address} \; \rangle \\
\quad\quad [\![\; \mathsf{d} \leftarrow \mathsf{Departments, d.name{=}``CSE",} \\
\quad\quad\quad \mathsf{e} \leftarrow \mathsf{d.instructors, c} \leftarrow \mathsf{e.teaches, c.name{=}``cse5331"} \; \}\!\}
\end{array}
$$

(by Rules (N9), (N8), and (N1)).

Canonical Forms: A path *path* is a bound variable, 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 a comprehension has a commutative accumulator, then the generator domains in this comprehension can be normalized into paths. If in addition all predicates of a comprehension are moved at the end of the comprehension forming a conjunction *pred* of predicates, then monoid comprehensions can be put into the following canonical form:

$$\oplus\{\!\{ \, e \, [\![\, v_1 \leftarrow path_1, \ldots, v_n \leftarrow path_n, pred \, \}\!\},$$

where each $path_i$ is a path. The proof of this statement is easy. First, it is easy to see that the previous form is a canonical form (i.e., it cannot be normalized further) since there is no normalization rule that reduces generators over paths. Second, if the domain e of a generator $v \leftarrow e$ in a monoid comprehension is a form other than a path, then this domain is reduced to a simpler form by the normalization algorithm. This situation becomes apparent if we consider all the different forms that e can take. For each such form there is a normalization rule, thus leaving the only form that cannot be reduced, namely a path.

Performance: Our normalization algorithm unnests all type N and J nested queries [Kim 1982] (using Rules (N8) and (N9), respectively). The important question, though, is whether normalization always improves performance. Unfortunately not: Consider the term $\cup\{\!\{ \, (v,v) \, [\![\, v \leftarrow \cup\{\!\{ \, E \, [\![\, w \leftarrow X \, \}\!\} \, \}\!\}$, where E is an expensive query. This term is normalized into $\cup\{\!\{ \, (E,E) \, [\![\, w \leftarrow X \, \}\!\}$, which performs the computation of E twice. In this case, the normalized form may perform worse than the original term. Cases such as these occur frequently in lazy functional languages [Peyton Jones 1987]. In those languages, function application is evaluated using beta reduction (Rule (N2)), which, if it is implemented naively as term substitution, may repeat computations (if v appears more than once in e_1). To avoid situations like these, the evaluators of these languages use graph reduction techniques [Peyton Jones 1987] in which all occurrences of v in e_1 share the same term by pointing to the same memory address, thus forming an acyclic graph. When this term is reduced to a value, the term is replaced by this value in the graph, thus avoiding the need to compute this value twice. If we apply this technique to our normalization algorithm, the normalized form $\cup\{\!\{ \, (E,E) \, [\![\, w \leftarrow X \, \}\!\}$ will not repeat the evaluation of E; instead it will use two pointers to the same copy of the term E.

If we use graph reduction during normalization, then, under the naive cost function described below, normalization always improves cost. Of course, this result may not be necessarily true for realistic cost functions and physical algorithms. The purpose of this cost analysis is to give a simple explanation why normalization is an effective optimization technique. It should be noted though that, like any relational algebraic optimization, normalization is a heuristic that improves performance in most cases. The true effectiveness of an optimization algorithm can only be asserted by extensive testing applied to real-world applications.

We assume that we know the average sizes of all sets reached by paths in the database schema: For example, in $\cup\{\!\{ e \,\|\, x \leftarrow X,\, y \leftarrow x.A \}\!\}$, the size of $x.A$ is the average of all sets $x.A$, for every $x \in X$. The following cost function assumes a naive nested-loop evaluation that does not depend of the order of the joins:

$$
\begin{aligned}
\text{size}[\![\oplus\{\!\{ e \,\|\, \overline{r} \}\!\}]\!] &= \text{size}[\![\overline{r}]\!] \\
\text{size}[\![\overline{r},\, v \leftarrow e]\!] &= \text{size}[\![e]\!] \times \text{size}[\![\overline{r}]\!] \\
\text{size}[\![\overline{r},\, pred]\!] &= \text{size}[\![\overline{r}]\!] \times \text{selectivity}[\![pred]\!] \\
\text{size}[\![\overline{r},\, v \equiv e]\!] &= \text{size}[\![\overline{r}]\!] \\
\text{size}[\![\ ]\!] &= 1
\end{aligned}
$$

$$
\begin{aligned}
\text{cost}[\![\oplus\{\!\{ e \,\|\, \overline{r} \}\!\}]\!] &= \text{cost}[\![e]\!] + \text{cost}[\![\overline{r}]\!] + \text{size}[\![\overline{r}]\!] \\
\text{cost}[\![\overline{r},\, v \leftarrow e]\!] &= \text{cost}[\![e]\!] + \text{cost}[\![\overline{r}]\!] \\
\text{cost}[\![\overline{r},\, pred]\!] &= \text{cost}[\![pred]\!] + \text{cost}[\![\overline{r}]\!] \\
\text{cost}[\![\overline{r},\, v \equiv e]\!] &= \text{cost}[\![e]\!] + \text{cost}[\![\overline{r}]\!] \\
\text{cost}[\![\ ]\!] &= 0
\end{aligned}
$$

where size$[\![e]\!]$ of a collection e estimates the size of e while selectivity$[\![pred]\!]$ is the selectivity of the predicate *pred*. The variable-binding cost equation (for $v \equiv e$) assumes the use of graph reduction, i.e., the computation of e is done only once, even when v is used more than once.

For example, the join $X \bowtie_{x.A=y.B} Y$ has cost cost$[\![\cup\{\!\{ (x,y) \,\|\, x \leftarrow X,\, y \leftarrow Y,\, x.A = y.B \}\!\}]\!]$ equal to size$[\![X]\!]$ $\times$ size$[\![Y]\!]$ $\times$ selectivity$[\![x.A = y.B]\!]$. The size function satisfies size$[\![\overline{r}, \overline{s}]\!]$ = size$[\![\overline{r}]\!]$ $\times$ size$[\![\overline{s}]\!]$ and the cost function satisfies cost$[\![\overline{r}, \overline{s}]\!]$ = cost$[\![\overline{r}]\!]$ + cost$[\![\overline{s}]\!]$. Under this cost function, we can prove that each normalization rule improves cost. For example, the left-hand side of Rule (N8) has cost

$$
\begin{aligned}
&\text{cost}[\![\oplus\{\!\{ e \,\|\, \overline{q},\, v \leftarrow \cup\{\!\{ e' \,\|\, \overline{r} \}\!\},\, \overline{s} \}\!\}]\!] \\
&= \text{cost}[\![e]\!] + \text{cost}[\![\overline{q},\, v \leftarrow \cup\{\!\{ e' \,\|\, \overline{r} \}\!\},\, \overline{s}]\!] + \text{size}[\![\overline{q},\, v \leftarrow \cup\{\!\{ e' \,\|\, \overline{r} \}\!\},\, \overline{s}]\!] \\
&= \text{cost}[\![e]\!] + \text{cost}[\![\overline{q}]\!] + \text{cost}[\![\cup\{\!\{ e' \,\|\, \overline{r} \}\!\}]\!] + \text{cost}[\![\overline{s}]\!] + \text{size}[\![\overline{q}]\!] \times \text{size}[\![\cup\{\!\{ e' \,\|\, \overline{r} \}\!\}]\!] \times \text{size}[\![\overline{s}]\!] \\
&= \text{cost}[\![e]\!] + \text{cost}[\![\overline{q}]\!] + \text{cost}[\![e']\!] + \text{cost}[\![\overline{r}]\!] + \text{size}[\![\overline{r}]\!] + \text{cost}[\![\overline{s}]\!] + \text{size}[\![\overline{q}]\!] \times \text{size}[\![\overline{r}]\!] \times \text{size}[\![\overline{s}]\!]
\end{aligned}
$$

which has an excess of size$[\![\overline{r}]\!]$ over the cost of the right-hand side of the rule:

$$
\begin{aligned}
&\text{cost}[\![\oplus\{\!\{ e \,\|\, \overline{q},\, \overline{r},\, v \equiv e',\, \overline{s} \}\!\}]\!] \\
&= \text{cost}[\![e]\!] + \text{cost}[\![\overline{q},\, \overline{r},\, v \equiv e',\, \overline{s}]\!] + \text{size}[\![\overline{q},\, \overline{r},\, v \equiv e',\, \overline{s}]\!] \\
&= \text{cost}[\![e]\!] + \text{cost}[\![\overline{q}]\!] + \text{cost}[\![\overline{r}]\!] + \text{cost}[\![e']\!] + \text{cost}[\![\overline{s}]\!] + \text{size}[\![\overline{q}]\!] \times \text{size}[\![\overline{r}]\!] \times \text{size}[\![\overline{s}]\!].
\end{aligned}
$$

Limitations: Even though the normalization algorithm unnests many forms of nested queries, there still some forms of queries that cannot be unnested by it. The following query contains three examples of such forms:

$$\cup\{\ \langle\ \mathsf{E{=}e,\ M{=}}\cup\{\ \mathsf{c}\ \|\ \mathsf{c}\leftarrow\mathsf{e.children},\ \wedge\{\ \mathsf{c.age}>\mathsf{d.age}\ \|\ \mathsf{d}\leftarrow\mathsf{e.manager.children}\ \}\ \}\ \rangle\\ \|\ \mathsf{e}\leftarrow\mathsf{Employees},\ \mathsf{e.salary}>max\{\ \mathsf{m.salary}\ \|\ \mathsf{m}\leftarrow\mathsf{Managers},\ \mathsf{e.age}>\mathsf{m.age}\ \}\ \}.$$

The subquery $\cup\{\ \mathsf{c}\ \|\ \mathsf{c}\leftarrow\mathsf{e.children},\ \wedge\{\ \mathsf{c.age}>\mathsf{d.age}\ \|\ \mathsf{d}\leftarrow\mathsf{e.manager.children}\ \}\ \}$ cannot be unnested by the normalization algorithm because the computed set must be embedded in the result of every iteration of the outer set comprehension. Similarly, the universal quantification (the $\wedge$-comprehension) and the aggregation (the max-comprehension) cannot be unnested by the normalization algorithm. These cases (which are types A and JA nested queries [Kim 1982]) require the use of outer-joins and grouping and they will be treated properly in Section 7.

5. MODEL EXTENSIONS

5.1 Handling Inconsistencies due to Idempotence

In Section 2.2 we showed that if we allow homomorphisms from idempotent to non-idempotent monoids, such as set cardinality, $\mathrm{hom}[\cup,+](\lambda x.\,1)\,(A)$, semantic inconsistencies may result. The reason is that, by definition, monoid homomorphisms are divide-and-conquer computations. For set cardinality, this property implies that

$$\mathrm{hom}[\cup,+](\lambda x.\,1)\,(A\cup B)\ =\ (\mathrm{hom}[\cup,+](\lambda x.\,1)\,(A))+(\mathrm{hom}[\cup,+](\lambda x.\,1)\,(B)).$$

But this equation is not valid when $A\cap B\neq\{\,\}$, because it counts the common elements between A and B twice. The correct equation should be

$$\mathrm{hom}[\cup,+](\lambda x.\,1)\,(A\cup B)\ =\ (\mathrm{hom}[\cup,+](\lambda x.\,1)\,(A))+(\mathrm{hom}[\cup,+](\lambda x.\,1)\,(B-A)).$$

This observation can be generalized to any monoid. The third equation of Definition 2 should become:

$$\mathrm{hom}[\oplus,\otimes](f)\,(x\oplus y)\ =\ (\mathrm{hom}[\oplus,\otimes](f)\,(x))\ \otimes\ (\mathrm{hom}[\oplus,\otimes](\lambda v.F(v,x))\,(y)),$$

where function F is defined as follows:

$$F(v,x)\ =\ \begin{cases}\textbf{if}\ v\in x\ \textbf{then}\ \mathcal{Z}_\oplus\ \textbf{else}\ f(v) & \text{for idempotent}\ \otimes\\ f(v) & \text{otherwise.}\end{cases}$$

Given this definition, if $\otimes$ is an idempotent collection monoid, then according to Theorem 2:

$$\begin{aligned}\mathrm{hom}[\oplus,\otimes](\lambda v.F(v,x))\,(x)\ &=\ \otimes\{\ w\ \|\ v\leftarrow x,\ w\leftarrow F(v,x))\ \}\\ &=\ \otimes\{\ w\ \|\ v\leftarrow x,\ w\leftarrow(\textbf{if}\ v\in x\ \textbf{then}\ \mathcal{Z}_\oplus\ \textbf{else}\ f(v))\ \}\\ &=\ \otimes\{\ w\ \|\ v\leftarrow x,\ w\leftarrow\mathcal{Z}_\oplus\ \}\ =\ \mathcal{Z}_\otimes.\end{aligned}$$

Consequently,

$$\begin{aligned}\mathrm{hom}[\oplus,\otimes](f)\,(x\oplus x)\ &=\ (\mathrm{hom}[\oplus,\otimes](f)\,(x))\ \otimes\ (\mathrm{hom}[\oplus,\otimes](F)\,(x))\\ &=\ (\mathrm{hom}[\oplus,\otimes](f)\,(x))\ \otimes\ \mathcal{Z}_\otimes\\ &=\ \mathrm{hom}[\oplus,\otimes](f)\,(x),\end{aligned}$$

which holds even when $\oplus$ is anti-idempotent. A similar proof holds for an idempotent primitive monoid $\otimes$.

In addition to the definition above, Rule (N7) of the normalization algorithm (Figure 4) must be modified accordingly:

$$\begin{array}{l}\oplus\{\, e \,[\!]\, \overline{q},\, v \leftarrow (e_1 \otimes e_2),\, \overline{s}\,\} \\ \quad \longrightarrow \left\{ \begin{array}{ll} (\oplus\{\, e \,[\!]\, \overline{q},\, v \leftarrow e_1,\, \overline{s}\,\}) \;\oplus\; (\oplus\{\, e \,[\!]\, \overline{q},\, v \leftarrow e_2,\, v \notin e_1,\, \overline{s}\,\}) & \text{for } \oplus \not\preceq \otimes \\ (\oplus\{\, e \,[\!]\, \overline{q},\, v \leftarrow e_1,\, \overline{s}\,\}) \;\oplus\; (\oplus\{\, e \,[\!]\, \overline{q},\, v \leftarrow e_2,\, \overline{s}\,\}) & \text{otherwise} \end{array} \right.\end{array}$$

for commutative $\oplus$ and $\otimes$. Note that the first alternative is when $\oplus \not\preceq \otimes$, that is, when $\oplus$ is commutative and $\otimes$ is both commutative and idempotent. For example, $+\{\, 1 \,[\!]\, x \leftarrow (\{1\} \cup \{1\})\,\}$ is normalized into $(+\{\, 1 \,[\!]\, x \leftarrow \{1\}\,\}) + (+\{\, 1 \,[\!]\, x \leftarrow \{1\},\, x \notin \{1\}\,\})$, which is equal to $1 + 0 = 1$.

5.2 Advanced List Operations

A primitive monoid useful for lists is the *function composition monoid*, $(\circ, \lambda x.\, x)$, where the function composition, $\circ$, is defined by the equation $(f \circ g)\, x = f(g(x))$ and is associative, but neither commutative nor idempotent. Even though the type of this monoid, $T_\circ(\alpha) = \alpha \rightarrow \alpha$, is parametric, it is still a primitive monoid. For a list $X = [a_1, a_2, \ldots, a_n]$, $\circ\{\, \lambda x.\, f(a, x) \,[\!]\, a \leftarrow X\,\}$ is equal to $(\lambda x.\, f(a_1, x)) \circ (\lambda x.\, f(a_2, x)) \circ \cdots \circ (\lambda x.\, f(a_n, x))$. Consequently, $\circ\{\, \lambda x.\, f(a, x) \,[\!]\, a \leftarrow X\,\}\,(v)$ computes the value $f(a_1, f(a_2, \ldots, f(a_n, v)))$. For example, the reverse of the list X is $\circ\{\, \lambda x.\, x{+\!\!+}[a] \,[\!]\, a \leftarrow X\,\}\,([\,])$ and the first element of X is $\circ\{\, \lambda x.\, a \,[\!]\, a \leftarrow X\,\}\,(\textsc{null})$.

The function composition monoid offers enormous expressive power to list operations because it can be used to compose functions that propagate a state during a list iteration. For example, the n^{th} element of a list X, $\mathrm{nth}(X, n)$, is

$$(\circ\{\, \lambda(x, k).\, (\textbf{if}\; k = n \;\textbf{then}\; a \;\textbf{else}\; x, k - 1) \,[\!]\, a \leftarrow X\,\}\,(\textsc{null}, \mathrm{length}(X))).\mathrm{fst},$$

which uses a counter k as the state to propagate through the iteration over X. A more complex example is sorting the elements of a list X using a total order $\leq$:

$$\begin{array}{lcl} \mathrm{sort}(X) & = & \circ\{\, \lambda z.\, \mathrm{insert}(a, z) \,[\!]\, a \leftarrow X\,\}\,([\,]) \\ \mathrm{insert}(a, Y) & = & (\circ\{\, \lambda(z, p).\, \textbf{if}\; p \vee a \leq b \;\textbf{then}\; ([b]{+\!\!+}z, p) \;\textbf{else}\; ([b, a]{+\!\!+}z, \mathrm{true}) \,[\!]\, b \leftarrow Y\,\} \\ & & ([\,], \mathrm{false})).\mathrm{fst}. \end{array}$$

List comprehensions of the form $\circ\{\, \lambda x.\, f(a, x) \,[\!]\, a \leftarrow X\,\}\,(v)$ are equivalent to list folds [Fegaras 1993], also known as a catamorphism [Meijer et al. 1991]. Like list folds, list comprehensions can capture the list primitive recursive function, $\mathrm{P}(f, v)(X)$, defined by the following recursive equations:

$$\begin{array}{lcl} \mathrm{P}(f, v)([\,]) & = & v \\ \mathrm{P}(f, v)([a]{+\!\!+}x) & = & f(a, x, \mathrm{P}(f, v)(x)) \end{array}$$

as follows:

$$\mathrm{P}(f, v)(X) \;=\; (\circ\{\, \lambda(x, r).\, (f(a, r, x), [a]{+\!\!+}r) \,[\!]\, a \leftarrow X\,\}\,(v, [\,])).\mathrm{fst}.$$

Note that $\mathrm{P}(f, v)(X)$ represents a paramorphism [Meijer et al. 1991], which can be simulated easily by a catamorphism (i.e., a fold) by returning a pair that carries the rest of the list along with the result. When some simple syntactic restrictions are applied to the list primitive recursive functions, these functions precisely characterize the polynomial time space [Bellantoni and Cook 1992; Leivant 1993].

Unfortunately, the expressive power gained by the function composition monoid comes with a high cost. Comprehensions of the form $\oplus\{\, e \,[\!]\, \overline{r},\, v \leftarrow \circ\{\, f \,[\!]\, \overline{s} \,\}\,(u),\, \overline{q} \,\}$ are very difficult to normalize and unnest.

5.3 Vectors and Arrays

Vectors and arrays are important collection types for scientific and other applications [Maier and Vance 1993; Libkin et al. 1996]. 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 specified by

$$\Box\{\, a[n-i-1] \,[\!]\, a[i] \leftarrow x \,\},$$

where $\Box$ is the anticipated monoid for vectors. 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 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 [Fegaras and Maier 1995]. 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 we adopt here, is to choose the latter of the two values if this value is not null, otherwise to choose the first. That way, the last non-null value overwrites the previous values.

We will now formalize these observations. We introduce a new collection monoid $(\Box, \mathcal{Z}_\Box, \mathcal{U}_\Box)$ to represent vectors of any type. A vector of type vector(α) is represented by the type set(int $\times$ α), which specifies a partial mapping from positive integers (i.e., vector indexes) to values (of type α). This monoid has the following primitives: (The unit function here is binary, but in functional languages any n-ary function is equivalent to a unary function applied to a tuple.)

$$\begin{array}{lcl} \mathcal{Z}_\Box & = & \{\,\} \\ \mathcal{U}_\Box(i,a) & = & \{(i,a)\} \\ x \Box y & = & y \cup (\cup\{\, (i,a) \,[\!]\, (i,a) \leftarrow x,\, \wedge\{\, i \neq j \,[\!]\, (j,b) \leftarrow y \,\}\,\}). \end{array}$$

That is, when the vector elements of x are merged with the vector elements of y, the result is y union the elements of x that are not indexed in y. This monoid is idempotent but not commutative. In database terminology, the integer index (the first component of the pair int $\times$ α) is the key of set(int $\times$ α). The vector merge function, $\Box$, reflects the common knowledge that, in a vector assignment $A[i] := v$, the new value v overwrites the old value of $A[i]$. Like any other monoid, the only purpose of the vector monoid is to provide meaning to vector operations, not to suggest in any way how vectors should be implemented.

For example, the set $\{(4,10),(6,13),(2,11),(3,12)\}$ corresponds to the vector $<$ NULL$, 11, 12, 10,$ NULL$, 13 >$. If this set is merged with the set $\{(1,5),(5,7),(3,6)\}$, which corresponds to the vector $< 5,$ NULL$, 6,$ NULL$, 7 >$, it constructs the set $\{(1,5),(5,7),(3,6),(2,11),(4,10),(6,13)\}$, which corresponds to $< 5, 11, 6, 10, 7, 13 >$.

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

$$\begin{array}{lcl}
\mathrm{sum_all}(x) & = & +\{\, a \,[\!]\, a[i] \leftarrow x \,\} \\
\mathrm{subseq}(x,n,l) & = & \Box\{\, a[i-n] \,[\!]\, a[i] \leftarrow x,\ i \geq n,\ i < l+n \,\} \\
\mathrm{permute}(x,p) & = & \Box\{\, a[b] \,[\!]\, a[i] \leftarrow x,\ b[j] \leftarrow p,\ i = j \,\} \\
\mathrm{concat}(x,y,n) & = & x \,\Box\, (\Box\{\, a[n+i] \,[\!]\, a[i] \leftarrow y \,\}) \\
\mathrm{inner}(x,y) & = & +\{\, a \times b \,[\!]\, a[i] \leftarrow x,\ b[j] \leftarrow y,\ i = j \,\}.
\end{array}$$

Here $\mathrm{subseq}(x,n,l)$ returns the vector $x[i]$, $i = n, \ldots, l+n-1$, $\mathrm{permute}(x,p)$ returns the vector $x[p[i]]$, $i = 1, \ldots, n$, and $\mathrm{inner}(x,y)$ returns the inner product of the two vectors x and y. A matrix can be represented as a vector of vectors, but the syntax becomes a little bit awkward:

$$\mathrm{map}(f)\, x \;=\; \Box\{\, (\Box\{\, f(b) \,[\!]\, b[j] \leftarrow a \,\})[i] \,[\!]\, a[i] \leftarrow x \,\}.$$

Function $\mathrm{map}(f)$ maps function f to all the elements of a matrix. In the comprehension above, x is a matrix and thus a is a vector. Therefore, the head of the comprehension should be $z[i]$ where z is a vector. Thus, z must be another vector comprehension. Fortunately, there is a better approach to matrices. We can consider the monoid $\Box$ to be of type set(index $\times\ \alpha$), where index is a type with equality. For simple vectors, index=int; for matrices, index=int $\times$ int. The actual type of the index can be inferred at compile time. The following are examples of matrix operations (we use the shorthand $a[i,j]$ for $(a,(i,j))$):

$$\begin{array}{lcl}
\mathrm{map}(f)\, x & = & \Box\{\, (f(a))[i,j] \,[\!]\, a[i,j] \leftarrow x \,\} \\
\mathrm{transpose}(x) & = & \Box\{\, a[j,i] \,[\!]\, a[i,j] \leftarrow x \,\} \\
\mathrm{row}(x,k) & = & \Box\{\, a[j] \,[\!]\, a[i,j] \leftarrow x,\ i = k \,\} \\
\mathrm{column}(x,k) & = & \Box\{\, a[i] \,[\!]\, a[i,j] \leftarrow x,\ j = k \,\} \\
\mathrm{multiply}(x,y) & = & \Box\{\, v[i,j] \,[\!]\, a[i,j] \leftarrow x,\ v \equiv \mathrm{inner}(\mathrm{row}(x,i), \mathrm{column}(y,j)) \,\}.
\end{array}$$

Here $\mathrm{row}(x,k)$ returns the k^{th} row of matrix x while $\mathrm{column}(x,k)$ returns the k^{th} column. The only purpose of the generator $a[i,j] \leftarrow x$ in matrix multiplication $\mathrm{multiply}(x,y)$ is to generate the indexes i and j. The latter function is an example where the Haskell approach to vectors [Thompson 1998] (also adopted by Libkin et al. [1996]), in which vector indexes are first generated from integer domains and then they are used to access vectors, is more intuitive than ours.

Another example, is matrix tiling where a matrix X is partitioned into 4x4 tiles and each tile is replaced by the sum of the values in the tile:

$$\begin{array}{rl}
\mathrm{tiling}(X) \;=\; \Box\{\, (+\{\, b \,[\!]\, b[k,l] \leftarrow X,\ i-4 \leq k \leq i+4,\ j-4 \leq l \leq j+4 \,\})[\lceil \frac{i}{4} \rceil, \lceil \frac{j}{4} \rceil] \\
[\!]\, a[i,j] \leftarrow X \,\}.
\end{array}$$

Normalization can be used to optimize vector and array operations. For example, transpose(transpose(x)) can be normalized into a comprehension equivalent to x.

OQL treats arrays like lists with indexing; it does not provide a convenient random bulk manipulation, such as the one provided by our vector comprehensions. OQL can be extended though with syntactic constructs to support our vectors. For example, map(f) x can be expressed in an OQL-like syntax as:

$$\begin{array}{l}\textbf{select vector } (f(a))[i,j] \\ \textbf{from } a[i,j] \textbf{ in } x.\end{array}$$

6. THE INTERMEDIATE OBJECT ALGEBRA

In Section 2.3 we expressed comprehensions in terms of homomorphisms (Definition 3). If homomorphisms were implemented as iterations over collections, then comprehensions would be evaluated in a nested-loops fashion (since comprehensions with more than one generator are translated into nested homomorphisms). But we can do better than that in most cases. Research in relational query optimization has already addressed the related problem of efficiently evaluating join queries by considering different join orders, different access paths to data, and different join algorithms [Selinger et al. 1979]. To effectively adapt this technology to handle comprehensions, comprehensions must be expressed in terms of algebraic operators, which, eventually, will be mapped into physical execution algorithms, like those found in database systems (e.g., merge join, hash join, etc). Section 7 describes a framework for translating terms in our calculus into efficient algorithms. This translation is done in stages. Queries in our framework are first translated into monoid comprehensions (as it is described in Section 3), which serve as an intermediate form, and then are translated into a version of the nested relational algebra that supports aggregation, quantification, outer-joins, and outer-unnests. At the end, the algebraic terms are translated into execution plans. This algebra is called the *intermediate object algebra*. We use both a calculus and an algebra as intermediate forms because the calculus closely resembles current OODB languages and is easy to normalize, while the algebra is lower-level and can be directly translated into the execution algorithms supported by database systems.

We classify the intermediate algebraic operators into two categories: basic and extended. We will prove that the basic algebra is equivalent to the monoid comprehension calculus, that is, all basic algebraic operators can be expressed in terms of the monoid calculus and vice versa. The purpose of the extended algebraic operators will be apparent when we present our query unnesting method (Section 7).

6.1 The Basic Intermediate Algebra

As a convenient notation for the semantics of the algebraic operators, we define $\{\, e \,\|\, \overline{r} \,\}$ as follows:

$$\{\, e \,\|\, \overline{r} \,\} \;=\; \begin{cases} \cup\{\, e \,\|\, \overline{r} \,\} & \overline{r} \text{ has a commutative and idempotent generator} \\ \uplus\{\, e \,\|\, \overline{r} \,\} & \overline{r} \text{ has a commutative generator} \\ +\!\!+\{\, e \,\|\, \overline{r} \,\} & \text{otherwise.} \end{cases} \qquad \text{(D4)}$$

Figure 5 defines the basic algebraic operators in terms of the monoid calculus. These operators generalize the operators of the nested relational algebra, so they have a straightforward explanation:

$$X \bowtie_p Y \;=\; \{\!\{\, (v,w) \,[\!]\, v \leftarrow X,\; w \leftarrow Y,\; p(v,w) \,\}\!\} \tag{O1}$$

$$\sigma_p(X) \;=\; \{\!\{\, v \,[\!]\, v \leftarrow X,\; p(v) \,\}\!\} \tag{O2}$$

$$\mu_p^{path}(X) \;=\; \{\!\{\, (v,w) \,[\!]\, v \leftarrow X,\; w \leftarrow path(v),\; p(v,w) \,\}\!\} \tag{O3}$$

$$\Delta_p^{\oplus/e}(X) \;=\; \oplus\{\!\{\, e(v) \,[\!]\, v \leftarrow X,\; p(v) \,\}\!\} \tag{O4}$$

Fig. 5. The Semantics of the Basic Intermediate Algebra

$$\frac{\sigma \vdash X : T_\oplus(t_1),\; \sigma \vdash Y : T_\otimes(t_2),\; \sigma \vdash p : t_1 \times t_2 \rightarrow \text{bool},\; \odot = max(\oplus,\otimes)}{\sigma \vdash X \bowtie_p Y : T_\odot(t_1 \times t_2)} \tag{T17}$$

$$\frac{\sigma \vdash X : T_\oplus(t),\; \sigma \vdash p : t \rightarrow \text{bool}}{\sigma \vdash \sigma_p(X) : T_\oplus(t)} \tag{T18}$$

$$\frac{\sigma \vdash X : T_\oplus(t_1),\; \sigma \vdash path : t_1 \rightarrow T_\otimes(t_2),\; \sigma \vdash p : t_1 \times t_2 \rightarrow \text{bool},\; \odot = max(\oplus,\otimes)}{\sigma \vdash \mu_p^{path}(X) : T_\odot(t_2 \times t_1)} \tag{T19}$$

$$\frac{\sigma \vdash X : T_\otimes(t_1),\; \sigma \vdash e : t_1 \rightarrow T_\oplus,\; \sigma \vdash p : t_1 \rightarrow \text{bool}}{\sigma \vdash \Delta_p^{\oplus/e}(X) : T_\oplus} \tag{T20}$$

$$\frac{\begin{array}{c}\sigma \vdash X : T_\otimes(t_1),\; \sigma \vdash e : t_1 \rightarrow T_\oplus,\; \sigma \vdash f : t_1 \rightarrow t_2 \\ \sigma \vdash p : t_1 \rightarrow \text{bool},\; \sigma \vdash g : t_1 \rightarrow t_3\end{array}}{\sigma \vdash \Gamma_{p/g}^{\oplus/e/f}(X) : T_\oplus(t_1 \times T_\oplus)} \tag{T21}$$

Fig. 6. The Typing Rules of the (Basic and Extended) Algebraic Operators

- **join**, $X \bowtie_p Y$, joins the collections X and Y using the join predicate p. This join is not necessarily between two sets; if, for example, X is a list and Y is a bag, then, according to Equation (D4), the output is a bag.
- **selection**, $\sigma_p(X)$, selects all elements of X that satisfy the predicate p.
- **unnest**, $\mu_p^{path}(X)$, returns the collection of all pairs (x,y), for each $x \in X$ and for each $y \in x.path$, that satisfy the predicate $p(x,y)$.
- **reduce**, $\Delta_p^{\oplus/e}(X)$, collects the values $e(x)$ for all $x \in X$ that satisfy $p(x)$ using the accumulator $\oplus$. The reduce operator can be thought as a generalized version of the relational projection operator.

The first four rules in Figure 6 are the typing rules for the basic algebraic operators.

THEOREM 4. *The basic intermediate algebra is equivalent to the monoid calculus.*

PROOF. The rules in Figure 5 translate the algebra into the calculus. The calculus-to-algebra translation algorithm is accomplished by the following rewrite rules:

$$[\![\oplus\{\, e \,[\!]\, \}]\!]_w \; E \;=\; \Delta^{\oplus/\lambda w.e}_{\lambda w.\mathtt{true}}(E) \qquad \text{(C1)}$$

$$[\![\oplus\{\, e_1 \,[\!]\, v \leftarrow e_2, \overline{r}\, \}]\!]_w \; E \;=\; [\![\oplus\{\, e_1 \,[\!]\, \overline{r}\, \}]\!]_{(w,v)} \; (\mu^{\lambda w.e_2}_{\lambda(w,v).\mathtt{true}}(E)) \qquad \text{(C2)}$$

$$[\![\oplus\{\, e \,[\!]\, p, \overline{r}\, \}]\!]_w \; E \;=\; [\![\oplus\{\, e \,[\!]\, \overline{r}\, \}]\!]_w \; (\sigma_{\lambda w.p}(E)) \qquad \text{(C3)}$$

A monoid comprehension $\oplus\{\, e_1 \,[\!]\, v \leftarrow e_2, \overline{r}\,\}$ that may appear at any point in a calculus form (including the ones inside other comprehensions) is translated into an algebraic form by $[\![\oplus\{\, e_1 \,[\!]\, \overline{r}\,\}]\!]_v \; e_2$. The comprehension $\oplus\{\, e_1 \,[\!]\, \overline{r}\,\}$ is translated by compiling the qualifiers in $\overline{r}$ from left to right using the term E as a seed that grows at each step. That is, the term E is the algebraic tree derived at a particular point of compilation. The subscript variables, w, are all the range variables encountered so far during the translation. The validity of the above rules can be easily proved using the semantics in Figure 5. □

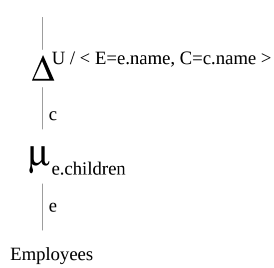
Note that the Rule (C2) compiles every comprehension generator into an unnest. This step may not be necessary if e_2 is a class extent, in which case it may be better compiled into a join. In Section 7, we will present an effective algorithm that translates the calculus into the algebra and at the same time performs heuristic optimizations and query unnesting. As an example of how comprehensions are translated into intermediate algebraic forms, consider the following

$$\cup\{\; \langle\; \mathsf{E} = \mathsf{e.name},\ \mathsf{C} = \mathsf{c.name} \;\rangle \;[\!]\; \mathsf{e} \leftarrow \mathsf{Employees},\ \mathsf{c} \leftarrow \mathsf{e.children} \;\},$$

which is translated into the algebraic form

$$\Delta^{\cup/\lambda(\mathsf{e},\mathsf{c}).\langle\; \mathsf{E}=\mathsf{e.name},\mathsf{C}=\mathsf{c.name} \;\rangle}_{\lambda(\mathsf{e},\mathsf{c}).\mathsf{true}}(\mu^{\lambda \mathsf{e}.\mathsf{e.children}}_{\lambda \mathsf{e}.\mathsf{true}}(\mathsf{Employees})).$$

Algebraic forms can be displayed as operator trees in which the tree leaves are collections of objects (e.g., class extents) and the output of the tree root is the output of the algebraic form. For example, the previous algebraic form has the following operator tree:



The functionality of an algebraic form can be better understood if we use a stream-based interpretation in which a stream of tuples flows from the leaves to the root of the tree. Under this interpretation, the algebraic form above generates a stream of tuples of type $\mathsf{set}(\langle\; \mathsf{e}\!:\ \mathsf{Employee} \;\rangle)$ from the extent $\mathsf{Employees}$. Variable e ranges over employees (i.e., it is of type $\mathsf{Employee}$). The unnest operator, $\mu_{\mathsf{e.children}}$, accepts the stream of tuples of type $\mathsf{set}(\langle\; \mathsf{e}\!:\ \mathsf{Employee} \;\rangle)$ and constructs a stream of tuples of type $\mathsf{set}(\langle\; \mathsf{e}\!:\ \mathsf{Employee},\ \mathsf{c}\!:\ \mathsf{Person} \;\rangle)$, connecting each employee with one of his or her children. The reduce operator, $\Delta^{\cup/<\mathsf{E}=\mathsf{e.name},\mathsf{C}=\mathsf{c.name}>}$, at the top of this

$$
\begin{aligned}
X \mathbin{=\!\!\bowtie}_p Y \;&=\; \{\, (v', w) \,\|\, v \leftarrow X,\; w \leftarrow \textbf{if } \wedge\{\, \neg p(v, w') \,\|\, v \neq \textsc{null},\; w' \leftarrow Y \,\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } [\textsc{null}] && \text{(O5)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \{\, w' \,\|\, w' \leftarrow Y,\; p(v, w') \,\}\,\} \\
=\!\!\mu_p^{path}(X) \;&=\; \{\, (v', w) \,\|\, v \leftarrow X,\; w \leftarrow \textbf{if } \wedge\{\, \neg p(v, w') \,\|\, v \neq \textsc{null},\; w' \leftarrow path(v) \,\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } [\textsc{null}] && \text{(O6)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } \{\, w' \,\|\, w' \leftarrow path(v),\; p(v, w') \,\}\,\} \\
\Gamma_{p/g}^{\oplus/e/f}(X) \;&=\; \{\, (v, \oplus\{\, e(w) \,\|\, w \leftarrow X,\; g(w) \neq \textsc{null},\; v \doteq f(w),\; p(w) \,\}) \\
&\qquad \|\, v \leftarrow \textstyle\prod_f(X) \,\} && \text{(O7)}
\end{aligned}
$$

Fig. 7. The Semantics of the Extended Algebra

algebraic form, is a generalization of the relational projection operator: It evaluates the expression $\langle$ E=e.name, C=c.name $\rangle$ for every input element and constructs a set from these tuples using $\cup$.

6.2 The Extended Intermediate Algebra

This section presents an extension to the basic algebra well-suited to be the target of the query unnesting algorithm described in Section 7. This algorithm uses outer-unnests and outer-joins to relate the data between the inner and outer queries of a nested query and then it reconstructs the result of the nested query using a group-by operator. The semantics of these operators cannot be given in terms of the monoid calculus if the outer query constructs a non-idempotent collection, because information about the exact number of copies of the data in the outer query may be lost after the outer join or unnest.

We are using the following new constructs to define the semantics of the extended algebra:

- Dot-equality $x \doteq y$ is defined if both x and y belong to the same collection C. If the collection C is idempotent, such as a set, then dot-equality is simply value equality. Otherwise, every element x in C is dot-different from any other element y in C. Note that dot-equality is stronger than OID equality. For example, the same object x may appear multiple times in a list, such as $[x, x]$, but the first element of this list is dot-different from the second. Even though dot-equality for C can be easily implemented by labeling the copies of an element in C with unique identifiers (such unique TIDs), it cannot be expressed in terms of C using our basic algebra.
- $\prod_f(X)$ is equal to $\{\, f(a) \,\|\, a \leftarrow X \,\}$ but with all duplicate elements under dot-equality removed.

These constructs satisfy the following properties for a function h:

$$\{\, h(\oplus\{\, e \,\|\, v' \leftarrow X,\; v' \doteq v,\; \overline{r} \,\}) \,\|\, v \leftarrow X \,\} \;=\; \{\, h(\oplus\{\, e \,\|\, v' \equiv v,\; \overline{r} \,\}) \,\|\, v \leftarrow X \,\} \qquad \text{(P1)}$$

$$\textstyle\prod_{\lambda(v,w).v}(\{\, (v', w') \,\|\, v' \leftarrow X,\; \overline{r} \,\}) \;=\; X \qquad \text{(P2)}$$

Property (P1) indicates that the generator $v' \leftarrow X$ can be safely removed from the inner comprehension since the elements v' are dot-equal to the elements v of the outer comprehension. Property (P2) indicates that, if we project over the elements of X and remove all duplicates, we get X.

The extended algebra operators are given in Figure 7 and are described below:

- **outer-join**, $X ⟕_p Y$, is a left outer-join between X and Y using the join predicate p. The domain of the second generator (the generator of w) in Equation (O5) is always non-empty. If Y is empty or there are no elements that can be joined with v (this condition is tested by the universal quantification), then the domain is the singleton value [NULL], i.e., w becomes null. Otherwise each qualified element w of Y is joined with v.
- **outer-unnest**, $=\!\!\!\mu_p^{path}(X)$, is similar to $\mu_p^{path}(X)$, but if $x.path$ is empty for $x \in X$ or $p(x, y)$ is false for all $y \in x.path$, then the pair (x, NULL) appears in the output.
- **nest**, $\Gamma_{p/g}^{\oplus/e/f}(X)$ in a way resembles $\Delta_p^{\oplus/e}(X)$; it combines the functionality of the nested relational operator, nest, with the functionality of reduce. In Equation (O7), the nest operator uses the group-by function f: If two values v and w from a set X are dot-equal under f (i.e., $f(v) \doteq f(w)$), their images under e (i.e., $e(v)$ and $e(w)$) are grouped together in the same group. After a group is formed, it is reduced by the accumulator $\oplus$ and a pair of the group-by value along with the result of the reduction of this group is returned. Function g indicates which nulls to convert into zeros (i.e., into $Z_\oplus$). For example, the nest operation

$$\Gamma_{\lambda(d,e).\mathrm{true}/\lambda(d,e).e}^{\cup/\lambda(d,e).e/\lambda(d,e).d}(X) \;=\; \cup\{\, (d', \cup\{\, e \,[\!]\, (d,e) \leftarrow X,\ e \neq \text{NULL},\ d' \doteq d \,\}) \;[\!]\; d' \leftarrow \textstyle\prod_{\lambda(d,e).d}(X) \,\}$$

groups the input (which consists of pairs (d, e) of a department d and an employee e) by d and converts the null e's into empty sets. $\prod_{\lambda(d,e).d}(X)$ is a generalized projection that retrieves the d component of each pair (d, e) in X and removes the duplicate d's.

7. QUERY UNNESTING

There are many recent proposals on OODB query optimization that are focused on unnesting nested queries [Cluet and Moerkotte 1995a; Cluet and Moerkotte 1995b; Claussen et al. 1997; Steenhagen et al. 1994] (also known as query decorrelation). Nested queries appear more often in OODB queries than in relational queries, because OODB query languages allow complex expressions at any point in a query. In addition, OODB types are allowed to have attributes with collection values (ie., nested collections), which lead naturally to nested queries. Current OODB systems typically evaluate nested queries in a nested-loop fashion, which does not leave many opportunities for optimization. Most unnesting techniques for OODB queries are actually based on similar techniques for relational queries [Kim 1982; Ganski and Wong 1987; Muralikrishna 1992]. For all but the trivial nested queries, these techniques require the use of outer-joins, to prevent loss of data, and grouping, to accumulate the data and to remove the null values introduced by the outer-joins.

If considered in isolation, query unnesting itself does not result in performance improvement. Instead, it makes possible other optimizations that would not be possible otherwise. More specifically, without unnesting, the only choice of evaluating nested queries is a naive nested-loop method: For each step of the outer query, all the steps of the inner query need to be executed. Query unnesting promotes all the operators of the inner query into the operators of the outer query. This operator mixing allows other optimization techniques to take place, such as the rearrangement of operators to minimize cost and the free movement of selection predicates between inner and outer operators, which enables operators to be more selective.

All early unnesting techniques were actually source-to-source transformations over SQL code, mostly due to the lack of a group-by operator in the relational algebra to express grouping. The absence of a formal theory in a form of an algebra to express these transformations resulted in a number of bugs (such as the infamous count bug [Ganski and Wong 1987]) which were eventually detected and corrected. Since OODB queries are far more complex than relational queries, it is more crucial to express the unnesting transformations in a formal algebra that will allow us to prove the soundness and completeness of these transformations. The first work with that goal in mind was by Cluet and Moerkotte [1995a, 1995b], which covered many cases of nesting, including nested aggregate queries, and validated all the transformations. Their work was extended by Claussen et al. [1997] to include universal quantification. Cherniack and Zdonik [1998a, 1998b] have used formal methods to validate query unnesting transformations, but their work is a moderate generalization of Kim's query unnesting techniques for relational queries.

This section describes a query decorrelation algorithm that unnests all possible forms of query nesting in our language. Our unnesting algorithm extends previous work in two ways. First, it is not only sound, but it is also complete. That is, our framework is capable of removing any form of nesting. Second, our unnesting algorithm is more concise, more uniform, and more general than earlier work, mostly due to the use of the monoid comprehension calculus as an intermediate form for OODB queries. The monoid comprehension calculus treats operations over multiple collection types, aggregates, and quantifiers in a similar way, resulting in a uniform way of unnesting queries, regardless of their type of nesting. In fact, many forms of nested queries can be unnested by the normalization algorithm for monoid comprehensions. The remaining forms require the introduction of outer-joins and grouping. Our algorithm requires only two rewrite rules to unnest the queries that cannot be handled by the normalization algorithm.

7.1 Examples of Unnesting

As an example of how OODB queries are unnested in our model, consider the following nested comprehension calculus expression:

QUERY A:

$$\cup\{\ \langle\ \mathsf{D{=}d},\ \mathsf{E}{=}\cup\{\ \mathsf{e}\ \|\ \mathsf{e} \leftarrow \mathsf{Employees},\ \mathsf{e.dno} = \mathsf{d.dno}\ \}\ \rangle\ \|\ \mathsf{d} \leftarrow \mathsf{Departments}\ \},$$

which, for each department, returns the employees of the department. The nesting inherent in this query can be avoided by using an outer-join combined with grouping [Muralikrishna 1992], as it is shown in Figure 8.A. The nest operator, $\Gamma_{\mathsf{d}}^{\cup/\mathsf{e}}$, groups the input by the range variable d, constructing a set of all

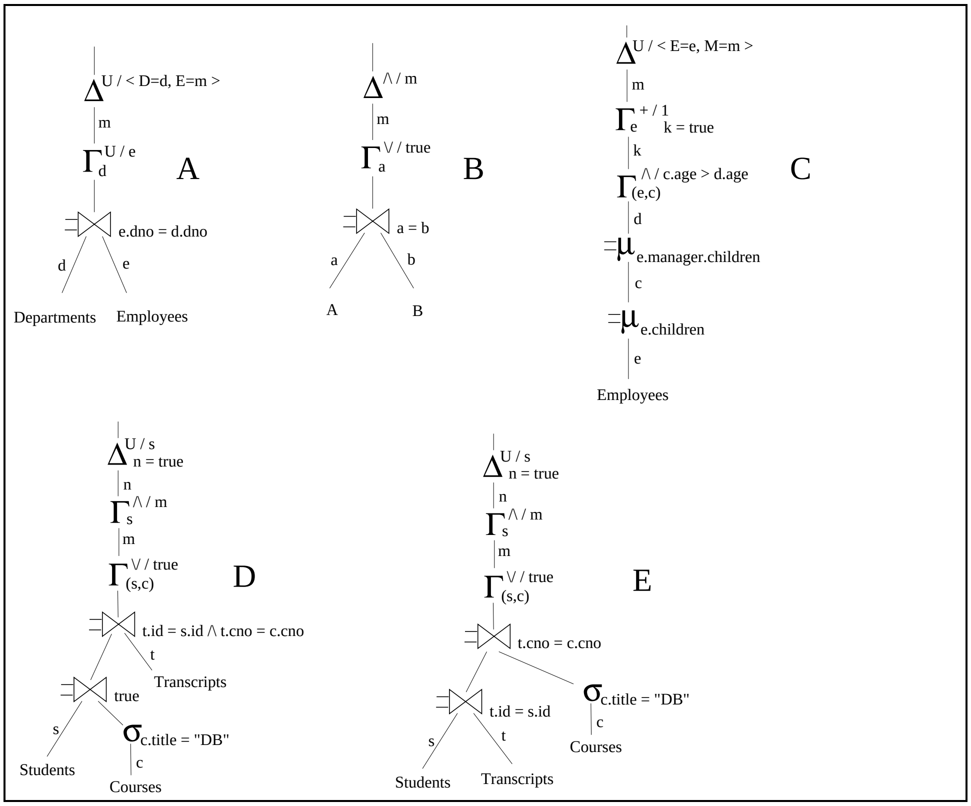


Fig. 8. The Algebraic Form of Some OODB Queries

e's that are associated with the value of d. This set becomes the value of the range variable m. That is, this nest operator reads a stream of tuples of type set(⟨ d: Department, e: Employee ⟩) and generates a stream of tuples of type set(⟨ d: Department, m: set(⟨ e: Employee ⟩) ⟩). The join before the nesting in Figure 8.A is a left outer-join: If there are no employees or the predicate e.dno=d.dno is false for all employees in a department d, then the result of the join is d associated with a null value, i.e., the value of the range variable e becomes null. The nest operator converts this null value into the empty set. That is, the outer-join introduces nulls and the nest operator converts nulls into zeros.

Another interesting example is the expression $A \subseteq B$, which is equivalent to $\forall a \in A : \exists b \in B : a = b$. It can be expressed in our calculus as follows:

QUERY B: $\wedge\{\ \vee\{\ \mathsf{true}\ \|\ \mathsf{b} \leftarrow \mathsf{B},\ \mathsf{a}{=}\mathsf{b}\ \}\ \|\ \mathsf{a} \leftarrow \mathsf{A}\ \}$.

The inner comprehension, which captures the existential quantification, checks if there is at least one $b \in B$ with $a = b$ (it can also be written as $\vee\{\,\mathsf{a}{=}\mathsf{b}\,\|\,\mathsf{b} \leftarrow \mathsf{B}\,\}$). The algebraic form of QUERY B after unnesting is shown in Figure 8.B. During the outer-join, if there is no a with b=a, then a will be matched with a null value. In that case, the nest operator will convert the null value into a false value, which is the zero element of ∨. The reduction at the root tests whether all m's are true.

If there is at least one false m, then there is at least one a in A that has no equal in B.

A more challenging example is the following double-nested comprehension:

QUERY C:

$$\cup\{\ \langle\ \mathsf{E{=}e},\ \mathsf{M{=}}{+}\{\ 1\ \rrbracket\ \mathsf{c} \leftarrow \mathsf{e.children},\ \wedge\{\ \mathsf{c.age} > \mathsf{d.age}\ \rrbracket\ \mathsf{d} \leftarrow \mathsf{e.manager.children}\ \}\ \}\ \rangle\ \rrbracket\ \mathsf{e} \leftarrow \mathsf{Employees}\ \},$$

which, for each employee, computes the number of the employee's children who are older than the oldest child of the employee's manager. The algebraic form of QUERY C is shown in Figure 8.C. Here, instead of an outer-join, we are using an outer-unnesting: the $=\!\!\!\mu_{\mathsf{e.children}}$ operator, which introduces a new range variable, c, whose value is the unnesting of the set e.children. If this set is empty, then c becomes null, i.e., the employee e is padded with a null value. Since we have a double-nested query, we need to use two nested unnest-nest pairs. The top nest operator groups the input stream by each employee e, generating the number of children of e. Note that every operator in our algebra can be assigned a predicate (such as the predicate k=true in the top nest operation) to restrict the input data. The second nest operator groups the input stream by both e and c (i.e., by the tuple (e, c)) and, for each group, evaluates the predicate c.age $>$ d.age and extends the output stream with an attribute k bound to the conjunction of all the predicates (indicated by the $\wedge$ accumulator).

The following comprehension finds the students who have taken all database courses. It uses a predicate that has the same pattern as that of $A \subseteq B$:

QUERY D:

$$\cup\{\ \mathsf{s}\ \rrbracket\ \mathsf{s} \leftarrow \mathsf{Students},\ \wedge\{\ \vee\{\ \mathsf{true}\ \rrbracket\ \mathsf{t} \leftarrow \mathsf{Transcript},\ \mathsf{t.id{=}s.id},\ \mathsf{t.cno{=}c.cno}\ \}\ \rrbracket\ \mathsf{c} \leftarrow \mathsf{Courses},\ \mathsf{c.title} = \text{“DB”}\ \}\ \}.$$

The algebraic form of this query is shown in Figure 8.D. This query can be evaluated more efficiently if we switch Courses with Transcripts, as it is shown in Figure 8.E. In that case, the resulting outer-joins are both assigned equality predicates, thus making them more efficient. Optimizations such as this justify query unnesting.

7.2 Our Query Unnesting Method

Figure 9 shows how our unnesting algorithm unnests QUERY D. Every comprehension is first normalized and then translated into an algebraic form consisting of regular joins, selections, unnests, and reductions, the last being the root of the algebraic form. We will see that this translation is straightforward. The outermost comprehension has one output (the result of the reduction) and no input streams. For example, the dashed box A in Figure 9 represents the outer comprehension in QUERY D. The shaded box on the reduction represents a nested query. The algebraic form of an inner comprehension (i.e., a comprehension inside another comprehension) has one input stream and one output value: The input stream is the same stream of tuples as that of the operation in which this form is embedded and the output is the result of this form. For example, the dashed box B in Figure 9 represents the universally quantified comprehension (with accumulator $\wedge$): the input of this box is the same input stream as that of the reduction in box A

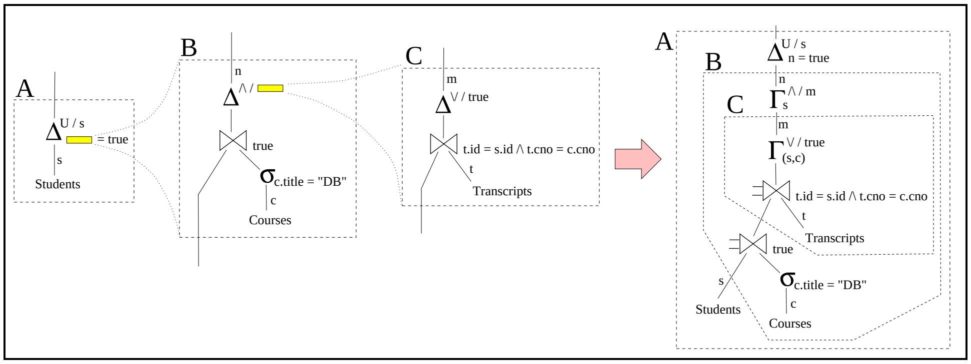


Fig. 9. Unnesting QUERY D

(namely the stream of employees), since box B is embedded in the predicate of the reduction. The output value of the box B, n, is used in the predicate of the reduction in the box A. Similarly, the existential quantification (with accumulator $\vee$) is translated into box C.

Our unnesting algorithm is simple: For each box that corresponds to a nested query (i.e., Boxes B and C in the current example), it converts reductions into nests (group-bys), and the joins and unnests that lay on the input-output path of the box into outer-joins and outer-unnests, respectively. At the same time, it embeds the resulting boxes at the points immediately before they are used. For example, box C will be embedded before the reduction in box B and the output value of box C will be used as the result of the innermost form. Similarly, box B will be embedded immediately before its output value is used in box A.

There is a very simple explanation why this algorithm is correct (a formal proof is given in Appendix A): A nested query is represented as a box, say box C, that consumes the same input stream as that of the embedding operation and computes a value m that is used in the embedding query (box B). If we want to splice this box onto the stream of the embedding query we need to guarantee two things. First, box C should not block the input stream by removing tuples from the stream. This condition is achieved by converting the blocking joins into outer-joins and the blocking unnests into outer-unnests. Second, we need to extend the stream with the new value m of box C before it is used in the reduction of box B. This manipulation can be done by converting the reduction of box C into a nest, since the main difference between nest and reduce is that, while the reduce returns a value (a reduction of a stream of values), nest embeds this value to the input stream. At the same time, the nest operator will convert null values to zeros so that the stream that comes from the output of the spliced box C will be exactly the same as it was before the splice. There are some important details that we omitted here, but we will present them later when we describe the unnesting algorithm in detail. The most important one is which nulls to convert to zeros each time: If we convert the null c's (i.e., the courses) to false in the second nest operation $\Gamma^{\vee/true}_{(s,c)}$ in the final unnested form in Figure 9, it will be too soon. This nest should convert null t's to false and the first nest should convert null c's to true. This difference is indicated by an extra parameter to the nest operator, which is not shown here.

$$
\begin{array}{rcll}
\llbracket \oplus\{\, e \,\|\, v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{()}_{()}\ \{()\} & = & \llbracket \oplus\{\, e \,\|\, \overline{r},\, p[\overline{v}] \,\} \rrbracket^{()}_{v}\ (\sigma_{\lambda v.p[v]}(X)) & \text{(C4)} \\[6pt]
\llbracket \oplus\{\, e \,\|\, p \,\} \rrbracket^{()}_{w}\ E & = & \Delta^{\oplus/\lambda w.e}_{\lambda w.p}(E) & \text{(C5)} \\[6pt]
\llbracket \oplus\{\, e \,\|\, v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{()}_{w}\ E & = & \llbracket \oplus\{\, e \,\|\, \overline{r},\, p[\overline{(w,v)}] \,\} \rrbracket^{()}_{(w,v)}\ (E \bowtie_{\lambda(w,v).p[(w,v)]} & \\
 & & \qquad\qquad (\sigma_{\lambda v.p[v]}(X))) & \text{(C6)} \\[6pt]
\llbracket \oplus\{\, e \,\|\, v \leftarrow path,\, \overline{r},\, p \,\} \rrbracket^{()}_{w}\ E & = & \llbracket \oplus\{\, e \,\|\, \overline{r},\, p[\overline{v}] \,\} \rrbracket^{()}_{(w,v)}\ (\mu^{\lambda w.path}_{\lambda(w,v).p[v]}(E)) & \text{(C7)} \\
\hline
\llbracket \oplus\{\, e \,\|\, p \,\} \rrbracket^{u}_{w}\ E & = & \Gamma^{\oplus/\lambda w.e/\lambda w.u}_{\lambda w.p/\lambda w.w \setminus u}(E) & \text{(C8)} \\[6pt]
\llbracket \oplus\{\, e \,\|\, v \leftarrow X,\, \overline{r},\, p \,\} \rrbracket^{u}_{w}\ E & = & \llbracket \oplus\{\, e \,\|\, \overline{r},\, p[\overline{(w,v)}] \,\} \rrbracket^{u}_{(w,v)}\ (E \mathbin{=\!\bowtie}_{\lambda(w,v).p[(w,v)]} & \\
 & & \qquad\qquad (\sigma_{\lambda v.p[v]}(X))) & \text{(C9)} \\[6pt]
\llbracket \oplus\{\, e \,\|\, v \leftarrow path,\, \overline{r},\, p \,\} \rrbracket^{u}_{w}\ E & = & \llbracket \oplus\{\, e \,\|\, \overline{r},\, p[\overline{v}] \,\} \rrbracket^{u}_{(w,v)}\ (=\!\mu^{\lambda w.path}_{\lambda(w,v).p[v]}(E)) & \text{(C10)} \\
\hline
\llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(\otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\} \rrbracket^{u}_{w}\ E & = & \llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(v) \,\} \rrbracket^{u}_{(w,v)}\ (\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\} \rrbracket^{w}_{w}\ E) & \text{(C11)} \\
 & & \otimes\{\, e_2 \,\|\, \overline{r} \,\} \text{ does not depend on the } \overline{s} \text{ generators} & \\[6pt]
\llbracket \oplus\{\, f(\otimes\{\, e \,\|\, \overline{r} \,\}) \,\|\, p \,\} \rrbracket^{u}_{w}\ E & = & \llbracket \oplus\{\, f(v) \,\|\, p \,\} \rrbracket^{u}_{(w,v)}\ (\llbracket \otimes\{\, e \,\|\, \overline{r} \,\} \rrbracket^{w}_{w}\ E) & \text{(C12)}
\end{array}
$$

Fig. 10. Rules for Translating and Unnesting Comprehensions ($u \neq ()$ in Rules C8-C10)

7.3 The Query Unnesting Algorithm

The query unnesting algorithm is expressed in Figure 10 in terms of a set of rewrite rules. We assume that all comprehensions in a query have been put into the canonical form $\oplus\{\, e \,\|\, v_1 \leftarrow path_1,\, \ldots,\, v_n \leftarrow path_n,\, pred \,\}$ before this algorithm is applied. That is, all generator domains have been reduced to paths and all predicates have been collected to the right of the comprehension into *pred* by anding them together (*pred* is set to true if no predicate exists). The translation of a monoid comprehension $\oplus\{\, e \,\|\, \overline{r} \,\}$ is accomplished by using $\llbracket \oplus\{\, e \,\|\, \overline{r} \,\} \rrbracket^{u}_{w}\ E$. The comprehension $\oplus\{\, e \,\|\, \overline{r} \,\}$ is translated by compiling the qualifiers in $\overline{r}$ from left to right using the term E as a seed that grows at each step. That is, the term E is the algebraic tree derived at this point of compilation. The variables in w are all the variables encountered so far during the translation and u are the variables that need to be converted to zeros during nesting if they are nulls. The situation $u = ()$ (i.e., when we have no variables in u) indicates that we are compiling an outermost comprehension (not a nested one). Rules (C4) through (C7) compile outermost comprehensions while Rules (C8) through (C10) compile inner comprehensions. Rules (C11) and (C12) do the actual unnesting (here u can be of any value, including $()$, and $\otimes$ is not necessarily the same monoid as $\oplus$).

Rule (C4) is the first step of the unnesting algorithm: The comprehension must be the outermost comprehension; thus, the first generator must be over an extent

X. In that case, the seed becomes a selection over X. The notation $p[v]$ specifies the part of the predicate p that refers to v exclusively and does not contain any embedded comprehensions. The rest of the predicate is denoted by $p[\overline{v}]$ and satisfies $p[v] \wedge p[\overline{v}] = p$. This decomposition is used for pushing predicates to the appropriate operators. Rule (C5) is the last rule to be performed, after all generators have been compiled. Rule (C6) converts a generator over an extent (a variable) into a join. Here we split the predicate p into three parts: $p[v]$ that refers to v exclusively, $p[(w, v)]$ that refers to both w and v, and $p[\overline{(w, v)}]$ for the rest of the predicate. Note that all predicates with embedded comprehensions should be assigned to $p[\overline{(w, v)}]$, since otherwise these embedded comprehensions will not be unnested. Details of how predicates are split into parts will be given later in this section when we present the unnesting algorithm in a more operational form. Rule (C7) compiles generators with path domains into unnests.

Rules (C8) through (C10) apply to inner comprehensions and are similar to Rules (C5) through (C7) with the only difference that reductions become nests, joins become left outer-joins, and unnests become outer-unnests. The notation $w \backslash u$ indicates all the variables in w that do not appear in u. Those variables are the attributes to group by (u are the attributes to convert into zeros when they are nulls). Rules (C11) and (C12) perform the actual unnesting. They do exactly what we have done in Figure 9 when we composed boxes: here the boxes are actually the results of the translation of the outer and inner comprehensions. Rule (C11) unnests a nested comprehension in the predicate p. It is applied as early as possible, that is, immediately when the generators $\overline{s}$ do not affect the inner comprehension (i.e., when the free variables of the inner comprehensions do not depend on the generator variables in $\overline{s}$). Rule (C12) unnests a nested comprehension in the head of a comprehension. This unnesting is performed when all the generators of the outer comprehension have been reduced.

For example, QUERY C, Q_C, is compiled as follows:

$$\begin{aligned} Q_C \; = \; & [\![\cup\{ \langle E = e,\, M = +\{ 1 \,\|\, c \leftarrow e.\text{children}, \\ & \qquad\qquad \wedge\{ c.\text{age} > d.\text{age} \,\|\, d \leftarrow e.\text{manager.children},\, \text{true} \} \} \rangle \\ & \quad \|\, e \leftarrow \text{Employees},\, \text{true} \}]\!]^{()}_{()} \; \{()\} \\ = \; & [\![\cup\{ \langle E = e,\, M = +\{ 1 \,\|\, c \leftarrow e.\text{children}, \\ & \qquad\qquad \wedge\{ c.\text{age} > d.\text{age} \,\|\, d \leftarrow e.\text{manager.children},\, \text{true} \} \} \rangle \\ & \quad \|\, \text{true} \}]\!]^{()}_{e} \; \text{Employees} \end{aligned}$$

from Rule (C4), if we ignore the selection over Employees (because it has a true predicate). From Rule (C12) we have:

$$\begin{aligned} Q_C \; = \; & [\![\cup\{ \langle E = e,\, M = m \rangle \,\|\, \text{true} \}]\!]^{()}_{(e,m)} \\ & \quad ([\![+\{ 1 \,\|\, c \leftarrow e.\text{children},\, \wedge\{ c.\text{age} > d.\text{age} \,\|\, d \leftarrow e.\text{manager.children},\, \text{true} \} \}]\!]^{e}_{e} \\ & \quad \text{Employees}) \end{aligned}$$

to handle the inner + comprehension. From Rule (C10) we have:

$$\begin{aligned} Q_C \; = \; & [\![\cup\{ \langle E = e,\, M = m \rangle \,\|\, \text{true} \}]\!]^{()}_{(e,m)} \\ & \quad ([\![+\{ 1 \,\|\, \wedge\{ c.\text{age} > d.\text{age} \,\|\, d \leftarrow e.\text{manager.children},\, \text{true} \} \}]\!]^{e}_{(e,c)} \\ & \quad (\overline{\mu}^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees}))) \end{aligned}$$

to translate the e.children into an unnest:

$$
\begin{aligned}
Q_C \;=\;& \llbracket \cup\{\, \langle E=e,\, M=m\rangle \parallel \text{true} \,\}\rrbracket^{()}_{(e,m)} \\
&\quad (\llbracket +\{\, 1 \parallel k \,\}\rrbracket^{e}_{((e,c),k)}\; (\llbracket \wedge\{\, c.\text{age} > d.\text{age} \parallel d \leftarrow e.\text{manager.children},\, \text{true} \,\}\rrbracket^{(e,c)}_{(e,c)} \\
&\qquad (=\!\!\!\mu^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees})))) \\
=\;& \llbracket \cup\{\, \langle E=e,\, M=m\rangle \parallel \text{true} \,\}\rrbracket^{()}_{(e,m)} \\
&\quad (\llbracket +\{\, 1 \parallel k \,\}\rrbracket^{e}_{((e,c),k)}\; (\llbracket \wedge\{\, c.\text{age} > d.\text{age} \parallel \text{true} \,\}\rrbracket^{(e,c)}_{((e,c),d)}\; (=\!\!\!\mu^{\lambda(e,c).\, e.\text{manager.children}}_{\lambda((e,c),d).\text{true}} \\
&\qquad (=\!\!\!\mu^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees}))))) \\
=\;& \llbracket \cup\{\, \langle E=e,\, M=m\rangle \parallel \text{true} \,\}\rrbracket^{()}_{(e,m)} \\
&\quad (\llbracket +\{\, 1 \parallel k \,\}\rrbracket^{e}_{((e,c),k)}\; (\Gamma^{\wedge/\lambda((e,c),d).\, c.\text{age}>d.\text{age}/\lambda((e,c),d).(e,c)}_{\lambda((e,c),d).\text{true}/\lambda((\text{e},\text{c}),\text{d}).\text{d}} \\
&\qquad (=\!\!\!\mu^{\lambda(e,c).\, e.\text{manager.children}}_{\lambda((e,c),d).\text{true}}\; (=\!\!\!\mu^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees}))))) \\
=\;& \llbracket \cup\{\, \langle E=e,\, M=m\rangle \parallel \text{true} \,\}\rrbracket^{()}_{(e,m)} \\
&\quad (\Gamma^{+/\lambda((e,c),k).1/\lambda((e,c),k).e}_{\lambda((e,c),k).k/\lambda((e,c),k).(c,k)}\; (\Gamma^{\wedge/\lambda((e,c),d).\, c.\text{age}>d.\text{age}/\lambda((e,c),d).(e,c)}_{\lambda((e,c),d).\text{true}/\lambda((e,c),d).d} \\
&\qquad (=\!\!\!\mu^{\lambda(e,c).\, e.\text{manager.children}}_{\lambda((e,c),d).\text{true}}\; (=\!\!\!\mu^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees}))))) \\
=\;& \Delta^{\cup/\lambda(e,m).\langle E=e,\, M=m\rangle}_{\lambda(e,m).\text{true}} \\
&\quad (\Gamma^{+/\lambda((e,c),k).1/\lambda((e,c),k).e}_{\lambda((e,c),k).k/\lambda((e,c),k).(c,k)}\; (\Gamma^{\wedge/\lambda((e,c),d).\, c.\text{age}>d.\text{age}/\lambda((e,c),d).(e,c)}_{\lambda((e,c),d).\text{true}/\lambda((e,c),d).d} \\
&\qquad (=\!\!\!\mu^{\lambda(e,c).\, e.\text{manager.children}}_{\lambda((e,c),d).\text{true}}\; (=\!\!\!\mu^{\lambda e.\, e.\text{children}}_{\lambda(e,c).\text{true}}(\text{Employees})))))
\end{aligned}
$$

(using Rules (C11), (C10), (C8), (C8), and (C5)), which is the algebraic form shown in Figure 8.C.

Figure 11 uses C-like pseudo code to present the unnesting rules in Figure 10 in a more operational way. The function call $T(e, u, w, E)$ corresponds to $\llbracket e \rrbracket^u_w\; E$. Thus, $T(e, (), (), \{()\})$ translates the term e into an algebraic form. Terms are tree-like data structures that represent both calculus and algebraic terms. A pattern u is a binary tree structure that represents an empty pattern $()$, a variable v, or a pair of patterns (u_1, u_2). Central to this program is the function split_predicate(Term p, Pattern left, Pattern right) that returns a triple (p_1, p_2, p_3) of terms. Terms p, p_1, p_2, and p_3 are conjunctions of simple predicates with $p = p_1 \wedge p_2 \wedge p_3$. In particular, p_1 holds all simple predicates in p that refer to the pattern variables in right exclusively; p_2 holds all predicates that refer to the pattern variables in both left and right, and p_3 holds the rest of the predicates. Note that there are no predicates in p that refer to the pattern variables in left exclusively, because these predicates have been assigned to the operators during previous steps. In addition, if a simple predicate is a comprehension (i.e., an embedded query in the predicate), then it is always assigned to p_3 in order to be unnested later by T. Line (2) in Figure 11 checks whether Rule (C11) can be applied. This case holds when the outermost comprehension $e' = \otimes\{\, e_2 \parallel \overline{r} \,\}$ of the predicate p does not depend on the range variables of the generators $\overline{r}$. This case always applies when $\overline{r}$ is empty and there is an embedded comprehension in p. Lines (3) and (4) implement the right side of the Rule (C11). The notation $p[e'/v]$ indicates that a copy of the term p is created but with the subtree e' in p substituted for the variable v. Lines (5) through (7) check whether Rule (C12) can be applied and Line (8) applies this rule. Otherwise, either Rule (C5) or Rule (C8) is applied,

(1) Term T (Term e, Pattern u, Pattern w, Term E) {
(2) **if** $e = \oplus\{\, e_1 \Vert \overline{s},\, p \,\}$ and the outermost subterm $e' = \otimes\{\, e_2 \Vert \overline{r} \,\}$ of p
(if one exists) does not depend on $\overline{s}$;
(3) { Pattern v := new_variable();
(4) **return** $T(\, \oplus\{\, e_1 \Vert \overline{s},\, p[e'/v] \,\},\, u,\, (w,v),\, T(e', w, w, E)\,)$;
}
(5) **else if** $e = \oplus\{\, e_1 \Vert p \,\}$
(6) { Pattern v := new_variable();
(7) find the outermost term $e' = \otimes\{\, e_2 \Vert \overline{r} \,\}$ in term e_1;
(8) **if** found **return** $T(\, \oplus\{\, e_1[e'/v] \Vert p \,\},\, u, (w,v),\, T(e', w, w, E)\,)$;
(9) **else if** u=() **return** $\Delta^{\oplus/\lambda w.e_1}_{\lambda w.p}(E)$;
(10) **else return** $\Gamma^{\oplus/\lambda w.e_1/\lambda w.u}_{\lambda w.p/\lambda u \backslash v}(E)$;
}
(11) **else if** $e = \oplus\{\, e_1 \Vert v \leftarrow X,\, \overline{r},\, p \,\}$
(12) { $(p_1, p_2, p_3) \leftarrow$ split_predicate(p, w, v);
(13) **if** $u = ()$
(14) **if** $w = ()$ **return** $T(\, \oplus\{\, e_1 \Vert \overline{r},\, p_3 \,\},\, u,\, v,\, \sigma_{\lambda v.np}(E)\,)$;
else if X is a variable
(15) **return** $T(\, \oplus\{\, e_1 \Vert \overline{r},\, p_3 \,\},\, u,\, (w,v),\, E \bowtie_{\lambda(w,v).p_2} (\sigma_{\lambda v.p_1}(E))\,)$;
(16) **else return** $T(\, \oplus\{\, e_1 \Vert \overline{r},\, p_3 \,\},\, u,\, (w,v),\, \mu^{\lambda w.X}_{\lambda(w,v).p_1 \wedge p_2}(E)\,)$;
else if X is a variable
(17) **return** $T(\, \oplus\{\, e_1 \Vert \overline{r},\, p_3 \,\},\, u,\, (w,v),\, E =\!\!\bowtie_{\lambda(w,v).p_2} (\sigma_{\lambda v.p_1}(E))\,)$;
(18) **else return** $T(\, \oplus\{\, e_1 \Vert \overline{r},\, p_3 \,\},\, u,\, (w,v),\, =\!\!\mu^{\lambda w.X}_{\lambda(w,v).p_1 \wedge p_2}(E)\,)$;
} }

Fig. 11. The Query Unnesting Algorithm

depending whether $u = ()$ or not. Line (14) applies when both $u = ()$ and $w = ()$ (Rule (C4)). Otherwise we have four cases: whether $u = ()$ or not and whether the first generator X is over a path or it is over a variable. Lines (15), (16), (17), and (18) capture the Rules (C6), (C7), (C9), and (C10), respectively.

We can easily prove that the unnesting algorithm in Figure 10 is complete:

THEOREM 5. *The rules in Figure 10 unnest all nested comprehensions.*

PROOF. After normalization, the only places where we can find nested queries are the comprehension predicate and the head of the comprehension. These cases are handled by Rules (C11) and (C12) respectively. Even though Rule (C11) has a precondition, it will be eventually applied to unnest any nested query in a predicate. (In the worst case, it will be applied when all generators of the outer comprehension have been compiled by the other rules.) □

The soundness of our unnesting algorithm is a consequence of the following theorem:

THEOREM 6. *The rules in Figure 10 are meaning preserving. That is:*

$$[\![\oplus\{\, e \Vert \overline{r} \,\}]\!]^{()}_{()}\ \{()\} \;=\; \oplus\{\, e \Vert \overline{r} \,\}.$$

The proof of this theorem is given in Appendix A (as Corollary 1 based on Theorem 7). Here we illustrate the validity of this theorem using an example. When we apply the rules in Figure 10, QUERY A becomes:

$$
\begin{aligned}
&[\![\cup\{\, \langle D = d,\, M = \cup\{\, e \,\|\, e \leftarrow \text{Employees},\, e.\text{dno} = d.\text{dno} \,\}\rangle \,\|\, d \leftarrow \text{Department} \,\}]\!]^{()}_{()}\ \{()\} \\
&= \Delta^{\cup/\lambda(d,m).\langle D=d,\, M=m\rangle}_{\lambda(d,m).\text{true}} \ (\Gamma^{\cup/\lambda(d,e).e/\lambda(d,e).d}_{\lambda(d,e).\text{true}/\lambda(d,e).e} \\
&\qquad (\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees})).
\end{aligned}
$$

We will prove that the algebraic form above is equivalent to the original comprehension. The tools that we will use for this proof are the operator definitions in Figures 5 and 7, and the normalization algorithm. According to Rule (O4) in Figure 7, the above algebraic form is equal to

$$
\begin{aligned}
&\cup\{\, \langle D = d,\, M = m\rangle \,\|\, (d,m) \leftarrow (\Gamma^{\cup/\lambda(d,e).e/\lambda(d,e).d}_{\lambda(d,e).\text{true}/\lambda(d,e).e} \\
&\qquad\qquad (\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees})) \,\} \\
&= \cup\{\, \langle D = d,\, M = m\rangle \\
&\quad \|\, (d,m) \leftarrow \cup\{\, (d, \cup\{\, e' \,\|\, (d',e') \leftarrow (\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees}), \\
&\qquad\qquad\qquad e' \neq \text{NULL},\, d \doteq d' \,\}) \\
&\qquad\qquad \|\, d \leftarrow \textstyle\prod_{\lambda(d,e).d}(\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees}) \,\} \,\} \\
&= \cup\{\, \langle D = d,\, M = \cup\{\, e' \,\|\, (d',e') \leftarrow (\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees}), \\
&\qquad\qquad\qquad e' \neq \text{NULL},\, d \doteq d' \,\}\rangle \\
&\quad \|\, d \leftarrow \textstyle\prod_{\lambda(d,e).d}(\text{Department} =\!\!\bowtie_{\lambda(d,e).\ e.\text{dno}=d.\text{dno}} \text{Employees}) \,\} \\
&= \cup\{\, \langle D = d,\, M = \cup\{\, e' \,\|\, (d',e') \leftarrow \cup\{\, (d,e) \,\|\, d \leftarrow \text{Department},\, e \leftarrow F(d) \,\}, \\
&\qquad\qquad\qquad e' \neq \text{NULL},\, d \doteq d' \,\}\rangle \\
&\quad \|\, d \leftarrow \textstyle\prod_{\lambda(d,e).d}(\cup\{\, (d,e) \,\|\, d \leftarrow \text{Department},\, e \leftarrow F(d) \,\}) \,\},
\end{aligned}
$$

where $F(d)$ is the right branch of the outer join in Rule (O5):

$$
\begin{aligned}
F(d) \ = \ &\textbf{if} \wedge\{\, e.\text{dno} \neq d.\text{dno} \,\|\, d \neq \text{NULL},\, e \leftarrow \text{Employees} \,\}\,\textbf{then}\,[\text{NULL}] \\
&\textbf{else} \cup\{\, e \,\|\, e \leftarrow \text{Employees},\, e.\text{dno} = d.\text{dno} \,\}.
\end{aligned}
$$

But, according to Property (P2), $\prod_{\lambda(d,e).d}(\cup\{\, (d,e) \,\|\, d \leftarrow \text{Department},\, e \leftarrow F(d) \,\})$ is equal to Department. Therefore, after normalization, we get:

$$
\begin{aligned}
&= \cup\{\, \langle D = d,\, M = \cup\{\, e' \,\|\, d' \leftarrow \text{Department},\, e' \leftarrow F(d'),\, e' \neq \text{NULL},\, d \doteq d' \,\}\rangle \\
&\quad \|\, d \leftarrow \text{Department} \,\} \\
&= \cup\{\, \langle D = d,\, M = \cup\{\, e' \,\|\, d' \leftarrow \text{Department},\, e' \leftarrow \cup\{\, e \,\|\, e \leftarrow \text{Employees},\, e.\text{dno} = d.\text{dno} \,\}, \\
&\qquad\qquad\qquad d \doteq d' \,\}\rangle \\
&\quad \|\, d \leftarrow \text{Department} \,\} \\
&= \cup\{\, \langle D = d,\, M = \cup\{\, e \,\|\, d' \leftarrow \text{Department},\, e \leftarrow \text{Employees},\, e.\text{dno} = d.\text{dno},\, d \doteq d' \,\}\rangle \\
&\quad \|\, d \leftarrow \text{Department} \,\}.
\end{aligned}
$$

Finally, according to Property (P1), we can safely remove the generator $d' \leftarrow$ Department from the inner comprehension. The final form is:

$$
\cup\{\, \langle D = d,\, M = \cup\{\, e \,\|\, e \leftarrow \text{Employees},\, e.\text{dno} = d.\text{dno} \,\}\rangle \,\|\, d \leftarrow \text{Department} \,\},
$$

which is the original comprehension.

7.4 Simplifications

There is a large class of nested queries that can be further improved after unnesting. Consider for example the following OQL query that, for each department, it finds the total salary of all the employees older than 30 in the department:

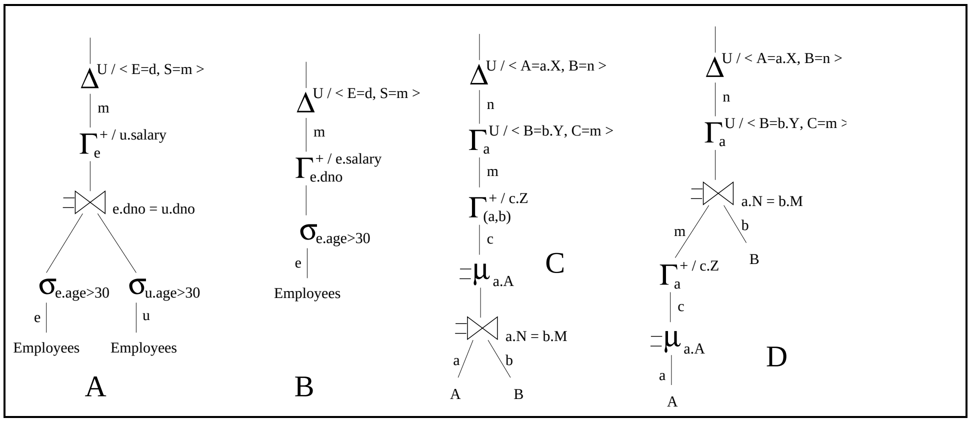


Fig. 12. Various Simplifications

```
  select distinct e.dno, sum(e.salary)
    from Employees e
   where e.age>30
group by e.dno.
```

Even though this query does not seem to be nested at a first glance, its translation to the monoid calculus is in fact nested:

$$\begin{aligned}&\cup\{\, \langle E = e.\text{dno},\, S = +\{\, u.\text{salary} \,\|\, u \leftarrow \text{Employees}, u.\text{age} > 30,\, e.\text{dno} = u.\text{dno} \,\}\rangle \\ &\quad \|\, e \leftarrow \text{Employees},\, e.\text{age} > 30 \,\}.\end{aligned}$$

Our unnesting algorithm generates the algebraic form in Figure 12.A, but we would prefer to have the form in Figure 12.B, which is more efficient.

This simplification can be easily accomplished with the help of the following transformation rule:

$$\Gamma_{a}^{f(b)}(g(a) =\!\bowtie_{a.M=b.M} g(b)) \;\rightarrow\; \Gamma_{a.M}^{f(a)}(g(a)),$$

where a and b are range variables in $g(a)$ and $g(b)$, respectively. That way, it does not make a difference whether this query is expressed as a group-by query or a nested query; both forms are optimized to the same efficient algebraic form.

As another simplification, we can use materialized views to handle some forms of uncorrelated nested queries [Cluet and Moerkotte 1995b]. For example, the following comprehension

$$\cup\{\, \langle C = c,\, D = \cup\{\, a.X + b.Y \,\|\, a \leftarrow A,\, b \leftarrow B,\, a.N = b.M \,\}\rangle \,\|\, c \leftarrow C \,\}$$

can be translated as follows:

$$\begin{aligned}&\mathit{view} \;:=\; \cup\{\, a.X + b.Y \,\|\, a \leftarrow A,\, b \leftarrow B,\, a.N = b.M \,\} \\ &\cup\{\, \langle C = c,\, D = \mathit{view}\rangle \,\|\, c \leftarrow C \,\},\end{aligned}$$

where the operator := creates a materialized view. All other cases of uncorrelated nested queries that cannot be handled with global views do not require any special treatment by our framework. For example, in the query

$$\cup\{\!\{\, \langle A = a.X,\, B = \cup\{\!\{\, \langle A = a.X,\, B = +\{\!\{\, c.Z \;[\!]\; c \leftarrow a.A \,\}\!\}\rangle \;[\!]\; b \leftarrow B,\, a.N = b.M \,\}\!\}\rangle \;[\!]\; a \leftarrow A \,\}\!\},$$

the + comprehension has the same value for every a in A. Our unnesting algorithm will generate the algebraic form shown in Figure 12.C for this query, but it is an easy task for an optimizer to transform this algebraic form into the form shown in Figure 12.D (by pulling the outer-join up in the operator tree).

8. IMPLEMENTATION

We have already built a prototype ODMG database management system, called λ-DB , based on our framework [Fegaras et al. 2000]. Our system can handle most ODL declarations and can process most OQL query forms. The λ-DB prototype is not ODMG compliant. Instead it supports its own C++ binding that provides a seamless integration between OQL and C++ with low impedance mismatch. It allows C++ variables to be used in queries and results of queries to be passed back to C++ programs. Programs expressed in our C++ binding are compiled by a preprocessor that performs query optimization at compile time, rather than run-time, as is proposed by ODMG. In addition to compiled queries, λ-DB provides an interpreter that evaluates ad-hoc OQL queries at run-time.

The λ-DB evaluation engine is written in SDL (the SHORE Data Language) of the SHORE object management system [Carey et al. 1994], developed at the University of Wisconsin. ODL schemas are translated into SDL schemas in a straightforward way and are stored in the system catalog. The λ-DB OQL compiler is a C++ preprocessor that accepts a language called λ-OQL, which is C++ code with embedded DML commands to perform transactions, queries, updates, etc. The preprocessor translates λ-OQL programs into C++ code that contains calls to the λ-DB evaluation engine. We also provide a visual query formulation interface, called VOODOO [Fegaras 1999b], and a translator from visual queries to OQL text, which can be sent to the λ-DB OQL interpreter for evaluation.

Our OQL optimizer is expressed in a very powerful optimizer specification language, called OPTL, and is implemented in a flexible optimization framework, called OPTGEN, which extends our earlier work on optimizer generators [Fegaras et al. 1993]. OPTL is a language for specifying query optimizers that captures a large portion of the optimizer specification information in a declarative manner. It extends C++ with a number of term-manipulation constructs and with a rule language for specifying query transformations. OPTGEN is a C++ preprocessor that maps OPTL specification into C++ code.

The λ-DB OQL optimizer proceeds in eight phases: 1) parsing of OQL queries and translation from OQL into calculus, 2) type checking, 3) normalization, 4) translation into algebra and query unnesting, 5) join permutation, 6) physical plan generation, 7) generation of intermediate evaluation code, and 8) translation from intermediate code into C++ code, or interpretation of the intermediate code.

Our unnesting algorithm, which is based on the framework described in earlier sections, turned out to be very easy to implement (it is only 160 lines of C++ code). In addition to query unnesting, λ-DB converts path expressions into pointer joins between class extents (when is possible) and it uses information about 1:N class relationships to convert unnests into pointer joins between extents (by using the inverse relationship).

Our query optimizer translates monoid comprehensions into algebraic operators, which, in turn, are mapped into physical execution algorithms, like those found in relational database systems. This translation is done in stages: queries in our framework are first translated into monoid comprehensions, which serve as an intermediate form, and then are translated into a version of the nested relational algebra that supports aggregation, quantification, outer-joins, and outer-unnests. At the end, the algebraic terms are translated into execution plans.

A very important task of any query optimizer is finding a good order to evaluate the query operations. In the context of relational databases, this task is known as join ordering. The λ-DB system uses a polynomial-time heuristic algorithm, called GOO [Fegaras 1998a], that generates a "good quality" order of the monoid algebra operators in the query algebraic form. GOO is a bottom-up greedy algorithm that always performs the most profitable operations first. The measure of profit is the size of the intermediate results, but it can be easily modified to use real cost functions. GOO is based on query graphs and takes into account both the output sizes of the results constructed in earlier steps and the predicate selectivities. It generates bushy join trees that have a very low total size of intermediate results.

After the best evaluation order of operators is derived, the algebraic form is mapped into an evaluation plan by a rule-based rewriting system. Our system considers the available access paths (indexes) and the available physical algorithms to generate different plans. During this phase, all alternative plans (for the derived order of operators) are generated and costed and the best plan is selected. Finally, each evaluation plan is translated into an intermediate evaluation code that reflects the signatures of the evaluation algorithms used in λ-DB . This code can then be translated straightforwardly into C++ or interpreted by the λ-DB interpreter.

The λ-DB evaluation engine is built on top of SHORE [Carey et al. 1994]. We use the SDL layer of SHORE exclusively in our implementation, because we believe it is more resilient to changes than the Shore Storage Manager and is easier to use. An alternative would have been to write our own value-added server on top of the storage manager. Much of our effort has been devoted to make the implementation of the evaluation engine as simple as possible without sacrificing performance. This required a very careful design. For example, one of our design requirements was to put all the evaluation algorithms in a library so that the query evaluation code would consist of calls to these algorithms. This approach sounds obvious enough and simple to implement, but it required some special techniques (described below) borrowed from the area of functional programming.

Algebraic operators, such as $R \bowtie_{R.A=S.B} S$, are intrinsically higher-order. That is, they are parameterized by pieces of code (i.e., functions) that specify some of the operation details. For the previous join, the piece of code is the predicate $R.A = S.B$. Most commercial DBMSs evaluate query plans using a plan interpreter. Very few systems actually compile plans into code, and when they do so, it is only for procedures written in a database language. If it were to be implemented in C++, the join operator could be specified as

Relation join (Relation x, Relation y, bool (pred) (tuple,tuple)),

where pred is the address of the predicate function. The same holds for any evaluation plan operator, such as the nested-loop join operator. If a query is to be compiled into execution code with no run-time interpretation of predicates, it should make use of higher-order evaluation algorithms. The alternative would be to define the evaluation algorithms as kinds of macros to be macroexpanded and individually tailored for each different instance of the evaluation operator in the query plan. For example, the nested-loop join would have to be a function with a "hole" in its body, to be filled with the predicate code. Then the entire code, the evaluation function and the predicate, would be generated and inserted inside the query code. This approach is clumsy and makes the development and extension of the evaluation algorithms very tedious and error-prone.

One very important issue to consider when writing evaluation algorithms is stream-based processing of data (sometimes called iterator-based processing or pipelining). It is highly desirable to avoid materializing the intermediate data in a secondary storage whenever it is possible. The λ-DB system pipelines all evaluation algorithms. Instead of using threads to implement pipelining, as it is traditionally done in most systems, we developed a special technique borrowed from the area of lazy functional languages. Each of our evaluation algorithms is written in such a way that it returns a piece of data (one tuple) as soon as it constructs one. To retrieve all the tuples, the algorithm must be called multiple times. For example, the pipeline version of the nested-loop join will have the signature

tuple nested_loop (Stream sx, Stream sy, bool (pred) (tuple,tuple)),

where Stream is a stream of tuples equipped with standard operations, such as first and next. In contrast to the standard nested-loop algorithm, this algorithm exits when it finds the first qualified tuple (that satisfies pred). To pipeline an algorithm, we construct a *suspended stream*, which is a structure with just one component: an embedded function, which, when invoked, calls the evaluation algorithm to construct one tuple. For example, to pipeline our nested-loop algorithm, we construct a suspended stream whose embedded function, F, is defined as tuple F () return nested_loop(s1,s2,pred); , where s1 and s2 are the streams that correspond to the join inputs and pred is the address of the predicate function. When a tuple is needed from a suspended stream, its embedded function is called with no arguments to return the next tuple. This approach is a clean and efficient way of implementing pipelining. This type of evaluation resembles lazy evaluation in functional programming languages. Here, though, we provide an explicit control over the lazy execution, which gives a better handle of controlling the data materialization in a secondary storage.

The intermediate evaluation code, which is generated from a query evaluation plan, is a purely functional program that consists of calls to the evaluation algorithms. The functional parameters of the calls (such as the predicate function of the nested-loop join), are represented as anonymous functions (lambda abstractions). If the intermediate code is compiled into C++, the lambda abstractions are translated into named C++ functions by a simple defunctionalization process. If the intermediate code is interpreted, each call to an evaluation algorithm is executed without much overhead because function addresses are stored into a vector

```
1   select x: e.name, y: (select c.name from c in e.teaches) from e in Instructors

2   select x: e.name, y: count(e.dept.instructors) from e in Instructors

3   select x: e.name, y: (select x: c.name, y: count(c.has_prerequisites)
    from c in e.teaches) from e in Instructors

4   select x: dn, y: count(partition) from e in Instructors group by dn: e.rank

5   select d.name, c: count(select * from e in d.instructors where e.rank=“professor”)
    from d in Departments order by count(select * from e in d.instructors
    where e.rank=“professor”)

6   select e.name, c: count(e.teaches) from e in Instructors where count(e.teaches)≥4

7   select x, y, c: count(c) from e in Instructors, c in e.teaches
    group by x: e.ssn, y: c.name having x>60 and y>“CSE5330”

8   select x: x, y: count(e) from e in Instructors group by x: count(e.teaches)
    having x>0

9   select x, y, c: count(e) from e in Instructors group by x: count(e.teaches),
    y: (exists c in e.teaches: c.name=“CSE5330”) having x>0

10  select x: x, y: count(e) from d in Departments, e in d.instructors
    group by x: count(e.teaches) having x>0

11  sum(select sum(select e.salary from e in d.instructors) from d in Departments)

12  select e.name, X: (select x from c in e.teaches
    group by x: count(c.has_prerequisites)) from d in Departments, e in d.instructors

13  select e from e in Instructors where for all c in e.teaches:
    exists d in c.is_prerequisite_for: d.name = “CSE5330”
```

Fig. 13. The Benchmark Queries

and calls to functions are dispatched in constant time by retrieving the function directly from the vector. Lambda abstractions are interpreted by using the address of the interpreter itself as a parameter and by pushing the body of the lambda abstraction into a special stack. This form of dispatching makes the interpreter very compact (it is 400 lines of C++ code only) and very fast.

Currently, the λ-DB evaluation engine supports table scan of a class extent, index scan, external sorting, block nested-loop join, indexed nested-loop join, sort-merge join, pointer join, unnesting, nesting (group-by), and reduction (to evaluate aggregations and quantifications). The λ-DB OODBMS is available as an open source software through the web at `http://lambda.uta.edu/lambda-DB.html`.

9. PERFORMANCE EVALUATION

To evaluate the performance of our query unnesting algorithm, we used the λ-DB OODBMS to optimize and run 13 nested OQL queries against a sample database. The database schema is given in Figure 1 and the OQL queries are shown in Figure 13. The experiments were performed under the following assumptions:

First, the queries that were not unnested by our decorrelation algorithm were evaluated in a naive nested loop fashion: for each step of the outer query, the entire inner query was evaluated. The query unnesting algorithm introduces outer-joins and outer-unnests to preserve the tuples of the outer query that are not joined with any tuple from the inner query. The outer-joins introduced by query unnesting are evaluated using outer-block-nested-loop joins. An outer-block-nested-loop join uses a fixed-size memory buffer to store a large number of tuples from the outer stream, thus reducing the number of the inner stream scans. Similarly, a left outer-join is implemented by marking the joined tuples in the buffer and then by checking for unmarked tuples before the buffer is replaced. We used a buffer of 1000 tuples in our tests, which means that the inner stream of a join was scanned about 1000 times less than that of the naive nested-loop implementation.

Second, in contrast to joins, there are very few ways of evaluating the unnest operator. In fact, λ-DB provides only one implementation for the unnest operator: a naive nested loop between each tuple and its collection component being unnested. This case is also true for the outer-unnests. Thus, if there are no joins in a query, then the only improvement introduced by query unnesting would come from moving predicates between the inner and outer query, which may result in more selective unnest operations. Since "predicate move around" has already been shown to be effective by others [Levy et al. 1994; Mumick et al. 1990], we decided to do benchmarks over queries with joins, by translating outer-joins into outer-block-nested loops. This modification was accomplished mainly by mapping path expressions, such as e.dept.instructors in Query 2, into pointer joins between class extents. This optimization, which is known as materialization of path expressions, is very effective when there are multiple paths with the same prefix. It is also very effective when combined with query unnesting, since query unnesting promotes the pointer joins of the inner queries to the outer query, thus allowing the mixing of the inner and outer operators.

Third, the group-by queries (queries 7, 8, 9, and 10) were optimized into group-by operations only when the queries were unnested. This step was necessary since the group-by queries in λ-DB are first translated into nested queries (as it is described in Section 3) and then these nested queries are translated into group-by operators as it is described in Section 7.4 (using the simplification transformations for query unnesting). The reason for doing this transformation in two steps instead of one is that the original group-by query and the translated nested query are semantically equivalent and the resulting evaluation plan should not depend on the way the user chooses to express the query. Of course, other systems may choose to do this translation in one step, in which case there would be no measurement differences between the nested and unnested versions of queries 7 through 10, since both will result in the same plan.

Finally, both nested and unnested queries were optimized extensively by the λ-DB optimizer and the evaluation plans were evaluated in a stream-based fashion.

The test database was created in four different sizes, as it is indicated by the cardinalities of the class extents for Departments, Instructors, and Courses in Figure 14. The benchmarks were run on a lightly loaded Linux workstation with a single 800MHz Pentium III, 256MB memory, and an ultra 160/m scsi 10000rpm

Departments/ Instructors/Courses	10/100/50	20/200/100	30/300/150	50/500/200
Query 1:	0.03 / 0.09	0.04 / 0.46	0.14 / 0.99	0.17 / 2.57
Query 2:	0.41 / 0.19	0.93 / 1.08	1.51 / 2.05	2.59 / 6.96
Query 3:	0.05 / 0.19	0.13 / 0.59	0.18 / 1.29	0.43 / 2.70
Query 4:	0.01 / 0.27	0.10 / 1.02	0.03 / 2.42	0.05 / 6.65
Query 5:	0.01 / 0.24	0.03 / 0.78	0.04 / 1.96	0.08 / 7.36
Query 6:	0.01 / 0.15	0.07 / 0.56	0.10 / 1.16	0.20 / 2.53
Query 7:	0.00 / 0.16	0.03 / 2.83	0.07 / 5.56	0.05 / 10.94
Query 8:	0.03 / 10.69	0.06 / 76.44	0.15 / 255.92	0.18 / 745.03
Query 9:	0.04 / 19.53	0.09 / 168.63	0.26 / 590.28	0.42 / *
Query 10:	0.07 / 11.94	0.16 / 92.09	0.24 / 301.30	0.39 / *
Query 11:	0.02 / 0.06	0.07 / 0.12	0.09 / 0.21	0.13 / 0.70
Query 12:	0.19 / 0.17	0.17 / 0.61	0.30 / 1.14	0.74 / 2.96
Query 13:	0.05 / 0.26	0.11 / 1.30	0.16 / 2.59	0.32 / 5.81

Fig. 14. Performance Evaluation of the Query Unnesting Algorithm

disk. The buffer pool size of the SHORE client was set to 20MB. The numbers shown in Figure 14 indicate processor time in seconds calculated using the unix function clock. Each entry in the table has two values, n/m, to indicate the execution time of the query in seconds with (n) and without (m) query unnesting. A star indicates incomplete execution due to a SHORE server crash. From the measurements above it is clear that query unnesting offers a significant performance improvement for the selected queries.

10. RELATED WORK

There are many proposals for object query algebras [Leung et al. 1993; Danforth and Valduriez 1992; Cluet and Delobel 1992; Beeri and Kornatzky 1990; Pistor and Traunmueller 1986]. 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 Breazu-Tannen et al. [1991, 1992] and Breazu-Tannen and Subrahmanyam [1991]. 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 [Breazu-Tannen and Subrahmanyam 1991], 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 ext(f) operation (which is equivalent to hom$[\oplus,\oplus](f)$ for a collection monoid $\oplus$). 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, $\mathrm{ext}(f)$ is not expressive enough, since it does not capture operations that involve different collection types and it cannot express predicates and aggregates. Our monoid comprehensions is the most expressive subset of SRU proposed so far, in which 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 Wadler [1990] as a generalization of list comprehensions. 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 mixing 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 database language by Trinder, Chan, and Wadler [Trinder and Wadler 1989; Trinder 1991; Chan and Trinder 1994], 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 Buneman et al. [1994] 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).

In an earlier work [Fegaras 1993], we used an alternative representation for lists, called the insert-representation, to construct list values. This method is commonly used in functional programming, mainly because it uses two constructors only, cons and nil, to construct list values, instead of using three: append, singleton, and nil. We have found that the append-representation is superior to the insert-representation when it comes to adding extra properties to the append constructor, such as commutativity and idempotence, to define bags and sets.

Our normalization algorithm is influenced by Wong's work on normalization of monad comprehensions [Wong 1993; Wong 1994]. He presented some very powerful rules for flattening nested comprehensions into canonical comprehension forms whose generators are over simple paths. These canonical forms are equivalent to our canonical forms for monoid comprehensions. His work, though, does not address query unnesting for complex queries (in which the embedded query is part of the predicate or the comprehension head), which cannot be unnested without using outer-joins and grouping.

There is an increasing number of recent proposals on OODB query optimization. Some of them are focused on handling nested collections [Ozsoyoglu and Wang 1992; Colby 1989], others on converting path expressions into joins [Kemper and Moerkotte 1990; Cluet and Delobel 1992], others on unnesting nested queries [Cluet and Moerkotte 1995b; Cluet and Moerkotte 1995a], while others are focused on handling encapsulation and methods [Daniels et al. 1991].

Query decorrelation was first introduced in the context of relational queries [Kim 1982; Ganski and Wong 1987; Muralikrishna 1992], mostly in the form of source-to-source transformations. There is a number of proposals lately that, instead of rewriting correlated queries into more efficient queries or algebraic forms, propose new evaluation algorithms that perform better than the default nested loop evaluation of embedded queries. Examples of such methods include magic decorrelation [Mumick et al. 1990], which promotes predicates in the inner loop of the nested loop evaluation of correlated queries, and predicate move around [Levy et al. 1994], which moves predicates around the query graph. Our decorrelation algorithm not only moves predicates to the right place, it also intermixes operators between outer and inner queries, thus supporting more possibilities for join permutations, which may lead to better plans.

Our query unnesting algorithm is influenced by the work of Cluet and Moerkotte [1995a, 1995b], which covered many cases of nesting in OODBs, including nested aggregate queries, and, more importantly, validated all the transformations. Their work was extended by Claussen et al. [1997] to include universal quantification. Our work proposes a rewriting system for complete unnesting, while their work considers algebraic equalities for some forms of unnesting.

The work of Lin and Ozsoyoglu [1996] addresses nested queries in a different way. Nested OQL queries in their approach are first translated into generalized path expressions that have enough expressive power to capture outer-joins and grouping. This translation may be very inefficient at the beginning due to a large number of redundant joins, but an optimization phase exists where path expressions are translated into joins and redundancies are eliminated. No proof was provided though to show that their two-phase unnesting method avoids redundancies in all forms and generates the same quality of plans as those of other unnesting techniques. Furthermore, no proof was given about the correctness and completeness of their method. We believe though that when these issues are addressed properly, their technique may turn out to be very effective and practical.

Another approach to query unnesting that has the same goals as ours is that of Cherniack and Zdonik [1998a, 1998b]. In contrast to our approach, they used an automatic theorem prover to prove the soundness of their unnesting rewrite rules, while we proved our transformations by hand. Their work, though, is a moderate generalization of Kim's query unnesting techniques for relational queries, while ours covers all forms of query nesting for complex OODB queries. The use of a theorem prover is highly desirable for extensible systems, since it makes the query optimizer very flexible. In particular, with the help of a theorem prover, an optimizer does not require validation in a form of a formal proof each time a new algebraic operator or a new rewrite rule is introduced.

11. CURRENT AND FUTURE WORK

Even though the basic framework for query unnesting has already been developed, there are some issues that we are planning to address in future research. The most important one is query unnesting in the presence of methods. The problem is not in handling a method invocation that contains a query in one of its arguments; this case can be handled effectively by our unnesting framework. Instead, the problem is when the method body itself contains queries. There are a number of proposals for handling this case. One is breaking the encapsulation and unfolding the method body in the place of the call. This approach does not apply to recursive methods and does not support separate compilation very well, since it requires query recompilation when a method changes. Another approach is attaching user-defined optimization rules to every class or method, which requires a substantial effort from programmers. Yet another approach is considering a method as a view and materializing it into a table (known as memoization in programming languages). We are planning to investigate all these approaches more thoroughly and choose one that best fits our unnesting algorithm.

We are currently developing a formal framework for optimizing object-oriented queries in the presence of side effects [Fegaras 1999a]. These queries may contain object updates at any place and in any form. We have proposed a language extension to the monoid comprehension calculus to express these object-oriented features and we have given a formal meaning to these extensions. Our method is based on denotational semantics, which is often used to give a formal meaning to imperative programming languages. The semantics of our language extensions is expressed in terms of our monoid calculus, without the need of any fundamental change to our basic framework. Our initial results suggest that our method not only maintains referential transparency, which allows us to do meaningful query optimization, but it is also practical for optimizing OODB queries since it allows the same optimization techniques applied to regular queries to be used with minimal changes for OODB queries.

12. CONCLUSION

We have presented a uniform calculus based on comprehensions that captures many advanced features found in modern object-oriented and object-relational query languages. We showed that this calculus is easy to manipulate by presenting a normalization algorithms that unnests many forms of nested comprehensions. The main contribution of this paper is the presentation of an effective framework for optimizing OODB queries. Queries in our framework are first translated into comprehensions and then into a lower-level algebra that reflects many relational DBMS physical algorithms. The algorithm that translates the calculus into the algebra performs query decorrelation by unnesting any form of nested comprehensions. Finally, we reported on an implementation of an ODMG-based DBMS based on our optimization framework.

REFERENCES

Beech, D. 1993. Collections of Objects in SQL3. In *Proceedings of the 19th VLDB Conference, Dublin, Ireland* (August 1993), pp. 244–255.

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

Bellantoni, S. and Cook, S. 1992. A new Recursion-Theoretic Characterization of the Polynomial Time. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia* (May 1992), pp. 283–293.

Breazu-Tannen, V., Buneman, P., and Naqvi, S. 1991. Structural Recursion as a Query Language. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece* (Aug. 1991), pp. 9–19. Morgan Kaufmann Publishers, Inc.

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

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

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

Buneman, P., Naqvi, S., Tannen, V., and Wong, L. 1995. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science 149*, 3–48.

Carey, M. and DeWitt, D. 1996. Of Objects and Databases: A Decade of Turmoil. In *Proceedings of the 22th VLDB Conference, Bombay, India* (Sept. 1996), pp. 3–14.

Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C., Tsatalos, O., White, S., and Zwilling, M. 1994. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota* (May 1994), pp. 383–394.

Cattell, R. Ed. 2000. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann.

Chan, D. and Trinder, P. 1994. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Twelfth British Conference on Databases, Guildford, United Kingdom* (July 1994), pp. 55–72. Morgan Kaufmann Publishers, Inc. LNCS 826.

Cherniack, M. and Zdonik, S. 1998a. Changing the Rules: Transformations for Rule-Based Optimizers. In *ACM SIGMOD International Conference on Management of Data, Seattle, Washington* (June 1998), pp. 61–72.

Cherniack, M. and Zdonik, S. 1998b. Inferring Function Semantics to Optimize Queries. In *Proceedings of the 24th VLDB Conference, New York City, NY* (Aug. 1998), pp. 239–250.

Claussen, J., Kemper, A., Moerkotte, G., and Peithner, K. 1997. Optimizing Queries with Universal Quantification in Object-Oriented and Object-Relational Databases. In *Proceedings of the 23th VLDB Conference, Athens, Greece* (Sept. 1997), pp. 286–295.

Cluet, S. and Delobel, C. 1992. A General Framework for the Optimization of Object-Oriented Queries. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California* (June 1992), pp. 383–392.

Cluet, S. and Moerkotte, G. 1995a. Efficient Evaluation of Aggregates on Bulk Types. Technical report (Oct.), Aachen University of Technology. Technical Report 95-05.

Cluet, S. and Moerkotte, G. 1995b. Nested Queries in Object Bases. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy* (Sept. 1995), pp. 8.

Colby, L. S. 1989. A Recursive Algebra and Query Optimization for Nested Relations. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Portland, Oregon* (1989), pp. 273–283.

Danforth, S. and Valduriez, P. 1992. A FAD for Data Intensive Applications. *ACM Transactions on Knowledge and Data Engineering 4*, 1 (Feb.), 34–51.

Daniels, S., Graefe, G., Keller, T., Maier, D., Schmidt, D., and Vance, B. 1991. Query Optimization in Revelation, an Overview. *IEEE Data Eng. Bull. 14*, 2 (June), 58.

Deux, O. 1990. The Story of O2. *ACM Transactions on Knowledge and Data Engineering 2*, 1 (March), 91–108.

Eisenberg, A. and Melton, J. 1999. SQL:1999, formerly known as SQL3. *SIGMOD Record 28*, 1 (March), 131–138.

Fegaras, L. 1993. Efficient Optimization of Iterative Queries. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City* (Aug. 1993), pp. 200–225. Springer-Verlag, Workshops on Computing.

Fegaras, L. 1998a. A New Heuristic for Optimizing Large Queries. In *9th International Conference, DEXA'98, Vienna, Austria* (Aug. 1998), pp. 726–735. Springer-Verlag. LNCS 1460.

Fegaras, L. 1998b. Query Unnesting in Object-Oriented Databases. In *ACM SIGMOD International Conference on Management of Data, Seattle, Washington* (June 1998), pp. 49–60.

Fegaras, L. 1999a. Optimizing Queries with Object Updates. *Journal of Intelligent Information Systems 12*, 219–242. Special issue on the Functional Approach to Intelligent Information Systems.

Fegaras, L. 1999b. VOODOO: A Visual Object-Oriented Database Language for ODMG OQL. In *1st ECOOP Workshop on Object-Oriented Databases, Lisbon - Portugal* (June 1999), pp. 61–72.

Fegaras, L. and Maier, D. 1995. Towards an Effective Calculus for Object Query Languages. In *ACM SIGMOD International Conference on Management of Data, San Jose, California* (May 1995), pp. 47–58.

Fegaras, L., Maier, D., and Sheard, T. 1993. Specifying Rule-based Query Optimizers in a Reflective Framework. In *Deductive and Object-Oriented Databases, Phoenix, Arizona* (Dec. 1993), pp. 146–168. Springer-Verlag, LNCS 461.

Fegaras, L., Srinivasan, C., Rajendran, A., and Maier, D. 2000. λ-db: An odmg-based object-oriented dbms. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas* (May 2000), pp. 583.

Ganski, R. and Wong, H. 1987. Optimization of Nested SQL Queries Revisited. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California* (May 1987), pp. 23–33.

Kemper, A. and Moerkotte, G. 1990. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia* (Aug. 1990), pp. 290–301. Morgan Kaufmann Publishers, Inc.

Kim, W. 1982. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems 7*, 3 (Sept.), 443–469.

Leivant, D. 1993. Stratified Functional Programs and Computational Complexity. In *Proceedings of the Twentieth ACM Symposium on Principles of Programming Languages, Charleston, South Carolina* (Jan. 1993), pp. 325–333.

Leung, T., Mitchell, G., Subramanian, B., Vance, B., Vandenberg, S., and Zdonik, S. 1993. The AQUA Data Model and Algebra. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City* (Aug. 1993), pp. 157–175. Springer-Verlag, Workshops on Computing.

Levy, A., Mumick, I., and Sagiv, Y. 1994. Query Optimization by Predicate Move-Around. In *Proceedings of the 20th VLDB Conference, Santiago, Chile* (Sept. 1994), pp. 96–107.

Libkin, L., Machlin, R., and Wong, L. 1996. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada* (May 1996), pp. 228–239.

Lin, J. and Ozsoyoglu, M. 1996. Processing OODB Queries by O-Algebra. In *International Conference on Information and Knowledge Management (CIKM), Rockville, Maryland* (Nov. 1996), pp. 134–142.

Maier, D. and Vance, B. 1993. A Call to Order. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC* (May 1993), pp. 1–16.

Meijer, E., Fokkinga, M., and Paterson, R. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts* (Aug. 1991), pp. 124–144. LNCS 523.

Mumick, I., Finkelstein, S., Pirahesh, H., and Ramakrishnan, R. 1990. Magic is Relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ* (May 1990), pp. 247–258.

MURALIKRISHNA, M. 1992. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *Proceedings of the International Conference on Very Large Data Bases* (Vancouver, BC, Canada, Aug. 1992), pp. 91–102.

OZSOYOGLU, Z. AND WANG, J. 1992. A Keying Method for a Nested Relational Database Management System. In *Proc. IEEE International Conference on Data Engineering, Tempe, AZ* (Feb. 1992), pp. 438–446.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall, Inc.

PIERCE, B. 1991. *Basic Category Theory for Computer Scientists.* The MIT Press.

PISTOR, P. AND TRAUNMUELLER, R. 1986. A Database Language for Sets, Lists, and Tables. *Information Systems 11*, 4, 323–336.

SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts* (May 1979), pp. 23–34.

STEENHAGEN, H., APERS, P., AND BLANKEN, H. 1994. Optimization of Nested Queries in a Complex Object Model. In M. JARKE, J. BUBENKO, AND K. JEFFERY Eds., *Advances in Database Technology – EDBT '94* (1994), pp. 337–350. Springer-Verlag. LNCS 779.

THOMPSON, S. 1998. *Haskell: The Craft of Functional Programming.* Addison-Wesley.

TRINDER, P. 1991. Comprehensions: A Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages: Bulk Types and Persistent Data, Nafplion, Greece* (Aug. 1991), pp. 55–68. Morgan Kaufmann Publishers, Inc.

TRINDER, P. AND WADLER, P. 1989. Improving List Comprehension Database Queries. In *in Proceedings of TENCON'89, Bombay, India* (Nov. 1989), pp. 186–192.

WADLER, P. 1990. Comprehending Monads. In *Proceedings of the ACM Symposium on Lisp and Functional Programming, Nice, France* (June 1990), pp. 61–78.

WONG, L. 1993. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems, Washington, DC* (May 1993), pp. 26–36.

WONG, L. 1994. *Querying Nested Collections.* Ph. D. thesis, Univerity of Pennsylvania. Also appeared as a Univerity of Pennsylvania technical report IRCS Report 94-09.

APPENDIX

A. PROOFS OF THEOREMS

THEOREM 3. *The normalization rules in Figure 4 are meaning preserving.*

Proof: We will prove the correctness of Rule (N8) only, since it is the most difficult rule to prove; there are similar proofs for the other rules. Even though the proof of Rule (N8) assumes the correctness of the other rules, there is no circularity since the other rules can be proved without a reference to Rule (N8). We define $\mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![e]\!]$, where $\overline{q}$ is a sequence of qualifiers and e is a term in the calculus, as follows:

$$\mathcal{F}_\oplus[\![\,]\!]\,[\![e]\!] \;=\; e \tag{F1}$$

$$\mathcal{F}_\oplus[\![x \leftarrow u, \overline{q}]\!]\,[\![e]\!] \;=\; \mathrm{hom}[\otimes,\oplus](\lambda x.\, \mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![e]\!])\, u \tag{F2}$$

$$\mathcal{F}_\oplus[\![\mathrm{pred}, \overline{q}]\!]\,[\![e]\!] \;=\; \textbf{if } \mathrm{pred} \textbf{ then } \mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![e]\!] \textbf{ else } \mathcal{Z}_\oplus \tag{F3}$$

where $\otimes$ in Equation (F2) is the collection monoid associated with u. Note that $\mathcal{F}$ is not a function over values but a function over terms in the calculus. It gets a sequence of qualifiers and a term as input and returns a term as output. It can be easily proved using these equations and Definition 3 that

$$\oplus\{\, e \,\|\, \overline{q}, \overline{r} \,\} \;=\; \mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![\oplus\{\, e \,\|\, \overline{r}\,\}]\!] \tag{W1}$$

First, we will prove that

$$\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, \overline{r}\,\},\, \overline{s}\,\} \;=\; \oplus\{\, e \,\|\, \overline{r},\, v \equiv e',\, \overline{s}\,\} \tag{W2}$$

using induction over the number of generators in $\overline{r}$ and using structural induction over the domain of the first generator in $\overline{r}$. If $\overline{r}$ does not contain any generator (i.e., it contains only predicates), then the equation above is true. If the first qualifier of $\overline{r}$ is a predicate, then we can move the predicate at the end of $\overline{r}$. Let $\overline{r} = w \leftarrow u, \overline{t}$, where t has n generators. We assume that Equation (W2) is true for a n generators in $\overline{r}$ (induction hypothesis), and we will prove it for $\overline{r}$ (induction step). Let u be a collection associated with the monoid $\odot$. Then u is $\mathcal{Z}_\odot$, or it is $\mathcal{U}_\odot(a)$ for some a, or it is $X \odot Y$ for some X and Y. If $u = \mathcal{Z}_\odot$, then both sides of the equation are equal to $\mathcal{Z}_\oplus$. If $u = \mathcal{U}_\odot(a)$, then

$$\begin{aligned}
\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, w \leftarrow \mathcal{U}_\odot(a),\, \overline{t}\,\},\, \overline{s}\,\} &= \oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, w \equiv a,\, \overline{t}\,\},\, \overline{s}\,\} \\
&= \oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e'[a/w] \,\|\, \overline{t}[a/w]\,\},\, \overline{s}\,\} \\
&= \oplus\{\, e \,\|\, \overline{t}[a/w],\, v \equiv e'[a/w],\, \overline{s}\,\} \\
&= \oplus\{\, e \,\|\, w \leftarrow \mathcal{U}_\odot(a),\, \overline{t},\, v \equiv e',\, \overline{s}\,\}
\end{aligned}$$

from Rule (N6), from Rule (N1), and from the induction hypothesis (assuming that there are no name conflicts between variables). We assume that the Equation (W2) is true for X and Y and we will prove it for $u = X \odot Y$:

$$\begin{aligned}
&\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, w \leftarrow (X \odot Y),\, \overline{t}\,\},\, \overline{s}\,\} \\
&= \oplus\{\, e \,\|\, v \leftarrow (\otimes\{\, e' \,\|\, w \leftarrow X,\, \overline{t}\,\}) \otimes (\otimes\{\, e' \,\|\, w \leftarrow X,\, \overline{t}\,\}),\, \overline{s}\,\} \\
&= (\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, w \leftarrow X,\, \overline{t}\,\},\, \overline{s}\,\}) \oplus (\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, w \leftarrow Y,\, \overline{t}\,\},\, \overline{s}\,\}) \\
&= (\oplus\{\, e \,\|\, w \leftarrow X,\, \overline{t},\, v \equiv e',\, \overline{s}\,\}) \oplus (\oplus\{\, e \,\|\, w \leftarrow Y,\, \overline{t},\, v \equiv e',\, \overline{s}\,\}) \\
&= \oplus\{\, e \,\|\, w \leftarrow (X \odot Y),\, \overline{t},\, v \equiv e',\, \overline{s}\,\}
\end{aligned}$$

from Rule (N7), and from induction hypothesis. Therefore, the left part of Rule (N8) becomes

$$\begin{aligned}
\oplus\{\, e \,\|\, \overline{q},\, v \leftarrow \otimes\{\, e' \,\|\, \overline{r}\,\},\, \overline{s}\,\} &= \mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![\oplus\{\, e \,\|\, v \leftarrow \otimes\{\, e' \,\|\, \overline{r}\,\},\, \overline{s}\,\}]\!] \\
&= \mathcal{F}_\oplus[\![\overline{q}]\!]\,[\![\oplus\{\, e \,\|\, \overline{r},\, v \equiv e',\, \overline{s}\,\}]\!] \\
&= \oplus\{\, e \,\|\, \overline{q},\, \overline{r},\, v \equiv e',\, \overline{s}\,\}
\end{aligned}$$

from Equation (W2). □

THEOREM 7. *The rules in Figure 10 satisfy the following equations:*

$$[\![\oplus\{\, e \,\|\, \overline{r}\,\}]\!]_w^{()}\; E \;=\; \oplus\{\, e \,\|\, w \leftarrow E,\, \overline{r}\,\} \tag{TH2}$$

$$\begin{aligned}
[\![\oplus\{\, e \,\|\, \overline{r}\,\}]\!]_w^{u}\; E \;=\; &\{\, (u', \oplus\{\, e \,\|\, w \leftarrow E,\, w\backslash u \neq \text{NULL},\, u' \doteq u,\, \overline{r}\,\}) \\
&\;\|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E)\,\} \quad \textit{for } u \neq ()
\end{aligned} \tag{TH3}$$

PROOF. Notice that, for $u = w$, we have $w\backslash u = ()$, and $\prod_{\lambda w.w}(E) = E$, in which case Equation (TH3) becomes:

$$[\![\oplus\{\, e \,\|\, \overline{r}\,\}]\!]_w^{w}\; E \;=\; \{\, (w, \oplus\{\, e \,\|\, \overline{r}\,\}) \,\|\, w \leftarrow E\,\} \tag{TH4}$$

since w in the inner comprehension is restricted to be equal to u' in the outer comprehension, and, according to Property (P1), the generator $w \leftarrow E$ of the inner comprehension can be safely removed.

We will prove the Equations (TH2) through (TH4) using structural induction. That is, assuming that all the subterms of a term satisfy the theorem (induction hypothesis), we will prove that a term itself satisfies the theorem (induction step). The induction basis is for the terms that do not contain subterms. All rules in Figure 10 are compositional, which means that the translation of a term is expressed in terms of the translation of its subterms. Thus, the structural induction will follow the recursion pattern of the rules in Figure 10. We will prove the theorems for Rules (C5), (C6), (C8), (C9), and (C11). The proofs for the other rules are similar.

Proof for Rule (C5):

$$\begin{aligned}[\![\oplus\{\, e \,[\!]\, p \,\}]\!]^{()}_{w}\; E \;&=\; \Delta^{\oplus/\lambda w.e}_{\lambda w.p}(E) && \text{from Rule (C5)}\\ &=\; \oplus\{\, e \,[\!]\, w \leftarrow E,\, p \,\} && \text{from Equation (O4)}\end{aligned}$$

which proves Theorem (TH2). Proof for Rule (C6):

$$\begin{aligned}&[\![\oplus\{\, e \,[\!]\, v \leftarrow X,\, \overline{r},\, p \,\}]\!]^{()}_{w}\; E\\ &= [\![\oplus\{\, e \,[\!]\, \overline{r},\, p[\overline{(w,v)}] \,\}]\!]^{()}_{(w,v)}\; (E \bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X)))\\ &= \oplus\{\, e \,[\!]\, (w,v) \leftarrow (E \bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X))),\, \overline{r},\, p[\overline{(w,v)}] \,\}\\ &= \oplus\{\, e \,[\!]\, (w,v) \leftarrow \{\, (w,v) \,[\!]\, w \leftarrow E,\, v \leftarrow \{\, v \,[\!]\, v \leftarrow X,\, p[v] \,\},\, p[(w,v)] \,\},\, \overline{r},\, p[\overline{(w,v)}] \,\}\\ &= \oplus\{\, e \,[\!]\, w \leftarrow E,\, v \leftarrow X,\, p[v],\, p[(w,v)],\, \overline{r},\, p[\overline{(w,v)}] \,\}\\ &= \oplus\{\, e \,[\!]\, w \leftarrow E,\, v \leftarrow X,\, \overline{r},\, p \,\}\end{aligned}$$

from Rule (C6), from induction hypothesis (TH2), from Eqs (O2) and (O1), and by using normalization. This proves Theorem (TH2). Proof for Rule (C8):

$$\begin{aligned}&[\![\oplus\{\, e \,[\!]\, p \,\}]\!]^{u}_{w}\; E = \Gamma^{\oplus/\lambda w.e/\lambda w.u}_{\lambda w.p/\lambda w.w\backslash u}(E)\\ &\;= \{\, (u[v/w], \oplus\{\, e \,[\!]\, w \leftarrow E,\, w\backslash u \neq \textsc{null},\, u[v/w] \doteq u,\, p \,\}) \,[\!]\, v \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\}\\ &\;= \{\, (u', \oplus\{\, e \,[\!]\, w \leftarrow E,\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, p \,\}) \,[\!]\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\}\end{aligned}$$

(from Rule (C8) and from Equation (O7)), which proves Theorem (TH3). Proof for Rule (C9):

$$\begin{aligned}&[\![\oplus\{\, e \,[\!]\, v \leftarrow X,\, \overline{r,\, p} \,\}]\!]^{u}_{w}\; E\\ &= [\![\oplus\{\, e \,[\!]\, \overline{r},\, p[\overline{(w,v)}] \,\}]\!]^{u}_{(w,v)}\; (E \;=\!\bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X)))\\ &= \{\, (u', \oplus\{\, e \,[\!]\, \underline{(w,v)} \leftarrow (E \;=\!\bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X))),\, (w,v)\backslash u \neq \textsc{null},\, u' \doteq u,\\ &\qquad\qquad \overline{r},\, p[\overline{(w,v)}] \,\}) \,[\!]\, u' \leftarrow \textstyle\prod_{\lambda(w,v).u}(E \;=\!\bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X))) \,\}\end{aligned}$$

from Rule (C9) and from induction hypothesis (TH3). From Equations (O5) and (O2) and after normalization, we get:

$$E \;=\!\bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X)) \;=\; \{\, (w,v) \,[\!]\, w \leftarrow E,\, v \leftarrow F \,\}$$

where $F = \textbf{if}\ \wedge\{\, \neg p[(w,v)] \,[\!]\, w \neq \textsc{null},\, v \leftarrow X,\, p[v] \,\}\ \textbf{then}\ [\textsc{null}]\ \textbf{else}\ \{\, v \,[\!]\, v \leftarrow X,\, p[v],\, p[(w,v)] \,\}$. Thus, $\prod_{\lambda(w,v).u}(E \;=\!\bowtie_{\lambda(w,v).p[(w,v)]}(\sigma_{\lambda v.p[v]}(X)))$ is equal to $\prod_{\lambda(w,v).u}(\{\, (w,v) \,[\!]\, w \leftarrow E,\, v \leftarrow F \,\})$, which is equal to $\prod_{\lambda w.u}(E)$, according to Property (P2). Therefore, we have:

$$
\begin{aligned}
&\llbracket \oplus\{\, e \,\|\, v \leftarrow X,\, \overline{r},\, p \,\}\rrbracket_w^u\; E \\
&\;=\; \{\, (u', \oplus\{\, e \,\|\, (w,v) \leftarrow \{\, (w,v) \,\|\, w \leftarrow E,\, v \leftarrow F \,\},\, v \neq \textsc{null},\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{r}, \\
&\qquad\qquad p[\overline{(w,v)}] \,\}) \,\|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e \,\|\, w \leftarrow E,\, v \leftarrow F,\, v \neq \textsc{null},\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{r},\, p[\overline{(w,v)}] \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e \,\|\, w \leftarrow E,\, v \leftarrow \{\, v \,\|\, v \leftarrow X,\, p[v],\, p[(w,v)] \,\},\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{r}, \\
&\qquad\qquad p[\overline{(w,v)}] \,\}) \,\|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e \,\|\, w \leftarrow E,\, v \leftarrow X,\, p[v],\, p[(w,v)],\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{r},\, p[\overline{(w,v)}] \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e \,\|\, w \leftarrow E,\, v \leftarrow X,\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{r},\, p \,\}) \,\|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\}
\end{aligned}
$$

since $v \neq$ NULL. Thus we have a proof of Rule (C9). For Rule (C11), we have two cases; when $u = ()$:

$$
\begin{aligned}
&\llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(\otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\}\rrbracket_w^{()}\; E \\
&\qquad\qquad = \llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(v) \,\}\rrbracket_{(w,v)}^{()}\; (\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\}\rrbracket_w^w\; E) \\
&\qquad\qquad = \oplus\{\, e_1 \,\|\, (w,v) \leftarrow (\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\}\rrbracket_w^w\; E),\, \overline{s},\, p(v) \,\} \\
&\qquad\qquad = \oplus\{\, e_1 \,\|\, (w,v) \leftarrow \{\, (w, \otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\|\, w \leftarrow E \,\},\, \overline{s},\, p(v) \,\} \\
&\qquad\qquad = \oplus\{\, e_1 \,\|\, w \leftarrow E,\, \overline{s},\, p(\otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\}
\end{aligned}
$$

(from Rule (C11), from hypotheses (TH2) and (TH4), and by using normalization) and when $u \neq ()$:

$$
\begin{aligned}
&\llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(\otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\}\rrbracket_w^u\; E \\
&\;=\; \llbracket \oplus\{\, e_1 \,\|\, \overline{s},\, p(v) \,\}\rrbracket_{(w,v)}^u\; (\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\}\rrbracket_w^w\; E) \\
&\;=\; \{\, (u', \oplus\{\, e_1 \,\|\, (w,v) \leftarrow (\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\}\rrbracket_w^w\; E),\, (w,v)\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{s},\, p(v) \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda(w,v).u}(\llbracket \otimes\{\, e_2 \,\|\, \overline{r} \,\}\rrbracket_w^w\; E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e_1 \,\|\, (w,v) \leftarrow (\{\, (w, \otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\|\, w \leftarrow E \,\}),\, (w,v)\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{s},\, p(v) \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda(w,v).u}(\{\, (w, \otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\|\, w \leftarrow E \,\}) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e_1 \,\|\, (w,v) \leftarrow (\{\, (w, \otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\|\, w \leftarrow E \,\}),\, (w,v)\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{s},\, p(v) \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\} \\
&\;=\; \{\, (u', \oplus\{\, e_1 \,\|\, w \leftarrow E,\, w\backslash u \neq \textsc{null},\, u' \doteq u,\, \overline{s},\, p(\otimes\{\, e_2 \,\|\, \overline{r} \,\}) \,\}) \\
&\qquad \|\, u' \leftarrow \textstyle\prod_{\lambda w.u}(E) \,\}
\end{aligned}
$$

from Rule (C11), from induction hypotheses (TH3) and (TH4), from property (P2), and by using normalization. □

COROLLARY 1. *The rules in Figure 10 are meaning preserving. That is:*

$$\llbracket \oplus\{\, e \,\|\, \overline{r} \,\}\rrbracket_{()}^{()}\; \{()\} \;=\; \oplus\{\, e \,\|\, \overline{r} \,\}$$

PROOF. This corollary is a consequence of Eq. (TH2) of Theorem 7 (for $w = ()$ and $E = \{()\}$). □