Translating OQL into Monoid Comprehensions—Stuck with Nested Loops?

Torsten Grust and Marc H. Scholl

3/1996
(Revised Version: September 1996)

University of Konstanz
Department of Mathematics and Computer Science*
Database Research Group

e-mail: {Torsten.Grust, Marc.Scholl}@uni-konstanz.de

Abstract

This work tries to employ the monoid comprehension calculus—which has proven to be an adequate framework to capture the semantics of modern object query languages featuring a family of collection types like sets, bags, and lists—in a twofold manner: First, serving as a target language for the translation of ODMG OQL queries. We review work done in this field and give comprehension calculus equivalents for the recently introduced OQL 1.2 concepts. This also presents a novel chance to understand the OQL semantics completely.

Our second concern is the fact that calculus-based languages are often said to be not efficiently implementable in the database context, having its reason in the fact that database query engines rather tend to realize algebraic interfaces to access bulk data, e.g. different join types. The main problem coming up here is the "nested-loop nature" of the calculus expressions. While these loop-based semantics for evaluating comprehensions at least provide a way for executing OQL queries, their execution is almost always much less efficient than alternative physical algorithms of the database engine.

Nonetheless, this paper shows how the calculus can serve as a good starting point from which advanced execution methods for complex queries can be derived. We present a non-standard interpretation of monoid comprehensions that gives rise to query processing techniques—namely bypass plans—developed recently. Furthermore, methods are proposed that allow for an efficient mapping of comprehensions to the query engine's primitives.

1 Introduction

The improvements that make object-oriented data models superior to their relational antecedents also have an impact on the formalisms that are needed to capture these new models completely. Speaking of the structural part of an object-oriented data model, the development led from flat tables, over nested relations to arbitrarily nested collections types, like sets, bags, and lists, that may include each other to any depth with no restrictions ("orthogonality of type constructors"). Reasoning about these structures is quite hard, and one can observe that some of the recently proposed object database models lack an underlying clean formalism, but rather solely depend on pragmatics or intuition when it comes to details of the model itself or the query language defined for it. This may have its reason in the fact that the well-understood relational set-based theory is

*University of Konstanz, Department of Mathematics and Computer Science, P.O.Box 5560/D188, D-78434 Konstanz, Germany. Fax: +49 7531 88-3577.

not adequate for "richer" object models and query languages like the one proposed by the ODMG [Cat93], anymore.

A reasonable approach to handle this whole family of collection types would be to identify the very basic similarities that all these share. Doing so, it would not be necessary to introduce special treatments for every collection type that the object model features, when defining query operations for it. Otherwise, one may end up with a huge set of type specific operators, like *list-select*, *bag-list-join*, making the query language somewhat intractable.

By understanding the bulk types as *monoids*, that is, algebraic structures equipped with an associative *merge* operation that obey a few simple laws, the treatment for the different types indeed becomes uniform. Identities and properties we proved for one of the types can be immediately applied to others as long as they can be captured by the monoid formalism [Feg93].

Comprehensions provide a convenient notation for computations carried out on monoid elements. Comprehension expressions resemble the relational calculus very closely and much of the effort already spent in this field can be easily adapted. Based solely on the *set* monoid, the relational calculus is in fact part of the more general monoid calculus. Using the monoid notion, we regain the power and the elegance of a calculus for the richer typed object models and their query languages. This is worth mentioning, since object query algebras (often derived from NF^2 algebras) dominate calculus approaches by far in today's work on formalization and optimization.

Furthermore, OQL, the query language proposed by the standardization efforts of the ODMG, is a calculus-based language like its relational predecessor SQL. A mapping of OQL's constructs to the monoid calculus is therefore rather easily obtained, and, not less important, lends a precise semantics.

On the other hand, implementing the evaluation of relational calculus expressions efficiently is considered tricky (if not hard) and the monoid calculus inherits these problems. The degree of orthogonality (or the freedom of using nested subqueries everywhere, to put it in other words) of modern query languages like OQL makes the whole case even more complex. Often, nested loop processing is the last resort then. This paper shows that it is possible to find efficient execution strategies for the monoid comprehension calculus. Moreover, we present a transformation strategy that yields efficient execution plans. We will especially integrate the recently proposed *bypass plans* [SPMK95]. Such approaches have not been undertaken for the monoid calculus yet and go beyond the optimization through normalization ideas of [FM95a]. Also, algebra-based optimizers that are aware of such execution strategies are still to be developed.

Its apparent potential made the monoid comprehension calculus the subject of recent reseach efforts. [TW89] and [Tri91] argued that list comprehensions, initially prominent as a construct found in functional programming languages, are a reasonable tool for querying databases. [Wad90] generalised list comprehensions to monads, making the framework applicable to monoids. Based on this work, [BLS+94] and [Feg94] explored calculi for collection types in general and showed that database query languages map to these in a natural way. Actually, the expressiveness of the monoid homomorphisms we investigate in the sequel is exactly that of a restricted form of structural recursion on collections, $\mathsf{ext}(f)$ in [BLS+94]. A step toward optimization through rewriting was the normalization algorithm of [Won93]. [FM95b] applied the idea of the monoid calculus and its normalization to OQL, providing a framework for query translation and also updates. Finally, [FM95a] used comprehensions to model the physical storage layout of a database. The monoid formalism is exploited to map conceptual to physical queries via an automatically generated abstraction function. However, optimization is solely done in terms of normalization and we will show methods that go beyond this approach.

This paper is organized as follows. The next section introduces the monoid comprehension calculus to the depth needed for the following discussion. The comprehensions are put down to monoid homomorphisms providing a clean semantics for queries dealing with multiple collection types. Nested loops provide an operational way to understand the calculus' semantics and we will explain this in the next section, too. Then, a mapping of the main OQL constructs to the comprehension calculus is given in Section 3. We will identify the characteristic calculus expressions this translation produces. By means of an example we explain how the unified treatment of scalars and collections influences optimizer search spaces positively. The main Section 4 presents techniques that improve the evaluation of monoid calculus expressions, namely *bypass plans* and the detection of *join patterns*. We conclude in Section 5 and point out how our further efforts in this field will look like. A translation of the complete body of OQL 1.2 clauses is then provided in Appendix A.

2 The Monoid Comprehension Calculus

To employ the **monoid comprehension calculus** ([Tri91], [WT91], [BLS+94], [Feg94], [FM95b]) as a database query language is an idea motivated by the above observations: sets, bags, and lists can all be captured by reducing them to their basic algebraic property, namely being monoids. A monoid $\mathcal{M}$ is an algebraic structure having an identity, $\mathit{zero}[\mathcal{M}]$, with respect to the monoid's associative $\mathit{merge}[\mathcal{M}]$ operation (borrowing notation from [Feg94]). For sets, the natural *merge* operation would be the set union $\cup$, having the empty set as its identity. In fact, it is only the additional properties of the set union, commutativity and idempotence, that makes sets different from bags (the additive union $\uplus$ not being idempotent) or lists (the list concatenation $+\!\!+$ neither being idempotent nor commutative).

Speaking of the type $\alpha\ \mathcal{M}$, that is, a collection whose members are of type α, we attach the function $\mathit{unit}[\mathcal{M}] : \alpha \to \alpha\ \mathcal{M}$, converting elements into single-element collections, thus making $\mathcal{M}$ free over member type α. Viewed within the monoid context, we now represent the bulk types *set*, *bag*, and *list*, as the triples $(\mathit{zero}[\mathcal{M}], \mathit{unit}[\mathcal{M}], \mathit{merge}[\mathcal{M}])$, resulting in $(\{\}, \{a\}, \cup)$, $(\{\!\!\{\}\!\!\}, \{\!\!\{a\}\!\!\}, \uplus)$, and $([], [a], +\!\!+)$, respectively.

An *int list* containing the elements $1 \ldots 3$ would be constructed as follows:

$$\begin{aligned}&\mathit{merge}[\mathit{list}](\mathit{unit}[\mathit{list}]1, \mathit{merge}[\mathit{list}](\mathit{unit}[\mathit{list}]2, \mathit{unit}[\mathit{list}]3))\\ &\quad = [1] +\!\!+ ([2] +\!\!+ [3])\\ &\quad = [1, 2, 3]\end{aligned}$$

In analogy to [Feg93] we call a monoid **primitive** if it is not free over a parametric type α, having the identity as its *unit* function. Examples for primitive monoids are $\mathit{sum} = (0, \mathrm{id}, +)$ and $\mathit{some} = (\mathit{false}, \mathrm{id}, \vee)$, both serving the obvious purpose:

$$\begin{aligned}&\mathit{merge}[\mathit{sum}](\mathit{unit}[\mathit{sum}]1, \mathit{merge}[\mathit{sum}](\mathit{unit}[\mathit{sum}]2, \mathit{unit}[\mathit{sum}]3))\\ &\quad = 1 + 2 + 3 = 6\end{aligned}$$

and

$$\begin{aligned}&\mathit{merge}[\mathit{some}](\mathit{unit}[\mathit{some}]\mathit{false}, \mathit{merge}[\mathit{some}](\mathit{unit}[\mathit{some}]\mathit{true}, \mathit{unit}[\mathit{some}]\mathit{false}))\\ &\quad = \mathit{false} \vee \mathit{true} \vee \mathit{false} = \mathit{true}\end{aligned}$$

(we omit parentheses because $\mathit{merge}[\mathcal{M}]$ is associative by definition). For any monoid $\mathcal{M}$ we will write $\mathcal{M}\{e_1, \ldots, e_n\}$ as a shorthand for the cascading $\mathit{merge}[\mathcal{M}]$ operations over the respective $\mathit{unit}[\mathcal{M}]e_i$.

Yet another example is the monoid of pairs over $\mathcal{M}$, $\mathit{pair}[\mathcal{M}]$, representing ordered pairs with a component-wise *merge* (let x_i, y_i be of type $\mathcal{M}$):

$$\begin{aligned}\mathit{zero}[\mathit{pair}[\mathcal{M}]] &= \langle \mathit{zero}[\mathcal{M}], \mathit{zero}[\mathcal{M}]\rangle\\ \mathit{merge}[\mathit{pair}[\mathcal{M}]](\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) &= \langle \mathit{merge}[\mathcal{M}](x_1, x_2), \mathit{merge}[\mathcal{M}](y_1, y_2)\rangle\end{aligned}$$

We mention it here because it will be of good use later on. The obvious generalization of this idea provides a means to reason about the OQL `struct` type constructor in the monoid framework, too.

The monoid framework is extensible in the sense that any structure having the monoid properties can be uniformly incorporated. For example, ordered sets and arrays have been dealt with that way, although the natural representation of the latter needs some further investigation (see [Bun93], for example). This gives us the freedom to introduce new monoids whenever this seems advantageous for the translation of OQL.

Let us turn to a convenient notation for bulk operations on monoid elements first, namely

2.1 Monoid Comprehensions

An expression of the form $\mathcal{M}\{e \mid e_1 \dots e_n\}$ is termed a **comprehension over monoid** $\mathcal{M}$. The notion of a "comprehension" originates from set theory (also known as *ZF-expressions* in this context), where these expressions denote sets containing exactly those elements satisfying the conditions specified by the terms e_i. e, the **head** of the comprehension, plays the role of a constructor that gets sequentially bound to arbitrarily complex values. All these bindings are then accumulated via $merge[\mathcal{M}]$ to make up the resulting value of the comprehension.

The terms e_i, the **qualifiers**, generate respectively filter out the values c gets bound to during the comprehension evaluation process:

- if e_i is of the form $x \leftarrow E$ (a **generator**) it sequentially binds variable x to the elements of expression E's value. The type of E, the generator's domain, may be arbitrary, especially a monoid $\mathcal{N} \neq \mathcal{M}$, as long as $\mathcal{N}$ is idempotent respectively commutative whenever $\mathcal{M}$ is (see [FM95b] for a motivation of this restriction; it turns out that this restriction does not influence the application of monoid comprehensions as a formal computational model for OQL). Variable x appears to be bound in $e_{i+1} \dots e_n$ and e.

- Otherwise, e_i is evaluated as a **filter** predicate in the context of previous generator variable bindings. If e_i evaluates to *false* the current bindings are not propagated any further.

Examples are:

$$\{e.A \mid e \leftarrow R\}$$

is the "projection" of all tuples of relation R on their A attribute (if e is bound to the record $\langle A = e_A, B = e_B, \dots \rangle$ with record labels $A, B, \dots$, then $e.A = e_A$). The result is of type β *set*, with β being the type of $e.A$.

$$\{\!\{e \mid e \leftarrow [1,2,3,1], e \neq 2\}\!\}$$

extracts all elements of the list that satisfy the filter. We do not care for order but for duplicates, so we choose *bag* as the outermost collecting monoid, resulting in $\{\!\{1,1,3\}\!\}$.

The exact semantics of these (standard) comprehensions are given by monoid homomorphisms, providing expressions that involve different monoids with a precise meaning.

2.2 Understanding Comprehensions as Monoid Homomorphisms

We introduce the idea of a standard mapping of comprehensions to homomorphisms here as it can be found in [Feg94]. The inherent nested-loop nature of the comprehension evaluation process will be evident thereafter. Let us not proceed without noting that this does *not* imply a nested-loop strategy for the final query execution plan. The monoid calculus provides us with a framework in which we can uniformly reason (i.e. aim for optimization) about computations that involve multiple collection types and scalars. Deducing an efficient execution strategy is a matter we tackle in Section 4.

In fact—driven by efficiency considerations—we will present a non-standard version of this translation when considering the bypass plan ideas.

As usual, a **monoid homomorphism** from monoid $\mathcal{N}$ to monoid $\mathcal{M}$, $\hom^{\mathcal{N}\mapsto\mathcal{M}}(f)$, maps $zero[\mathcal{N}]$ to $zero[\mathcal{M}]$, replaces $unit[\mathcal{M}](x)$ with $f(x)$, and $merge[\mathcal{N}]$ with $merge[\mathcal{M}]$. $\mathcal{N}$ and $\mathcal{M}$ undergo the aforementioned constraint with respect to their *merge* operations. Details can be found in [Feg94].

Defining the semantics of the comprehension $\mathcal{M}\{e \mid L\}$ involves an induction over the list of qualifiers, L. We define the base, $L = nil$, as

$$\mathcal{M}\{e \mid \} \quad = \quad unit[\mathcal{M}](e) \tag{1}$$

and the two possible cases for L being non-empty, namely having a generator $x \leftarrow E$ (with E representing a value of type $\mathcal{N}$) respectively a predicate p as its head, as

$$\begin{aligned} \mathcal{M}\{e \mid x \leftarrow E, l\} &= \hom^{\mathcal{N} \mapsto \mathcal{M}}(\lambda x.\mathcal{M}\{e \mid l\})E & (2) \\ \mathcal{M}\{e \mid p, l\} &= \mathit{if}\ p\ \mathit{then}\ \mathcal{M}\{e \mid l\}\ \mathit{else}\ \mathit{zero}[\mathcal{M}] & (3) \end{aligned}$$

Equation (2) applies function $\lambda x.\mathcal{M}\{e \mid l\}$ to every element x of E, assembling the results via *merge*$[\mathcal{M}]$. Note, that in this translation equation (3) cuts any further evaluation short if predicate p turns out to be *false*.

Following this scheme, the bag comprehension shown in the previous section is then evaluated as follows:

$$\begin{aligned} \{\!\{e \mid e \leftarrow [1,2,3,1], e \neq 2\}\!\} &= \hom^{list\ \mapsto\ bag}(\lambda e.\{\!\{e \mid e \neq 2\}\!\})\ [1,2,3,1] \\ &= \hom^{list\ \mapsto\ bag}(\lambda e.\mathit{if}\ e \neq 2\ \mathit{then}\ \{\!\{e \mid\ \}\!\}\ \mathit{else}\ \{\!\{\}\!\})\ [1,2,3,1] \\ &= \hom^{list\ \mapsto\ bag}(\lambda e.\mathit{if}\ e \neq 2\ \mathit{then}\ \{\!\{e\}\!\}\ \mathit{else}\ \{\!\{\}\!\})\ [1,2,3,1] \\ &= \{\!\{1,3,1\}\!\} \end{aligned}$$

The comprehension $\mathcal{M}\{e \mid x \leftarrow E\}$ may be equivalently computed by a loop (featuring a **foreach** construct known from several programming languages) like the one below:

```
C := zero[M];
foreach x in E
   C := merge[M](C, unit[M]e);
return C;
```

Since the generator domain E, in general, is a comprehension itself, this scheme immediately leads to nested loops, if E is replaced by its nested loop equivalent form, too. Any multi-generator comprehension yields nested loop forms. For example, $\mathcal{M}\{e \mid x \leftarrow E, y \leftarrow E'\}$ computes

```
C := zero[M];
foreach x in E
   foreach y in E'
      C := merge[M](C, unit[M]e);
return C;
```

In general, the comprehension

$$\mathcal{M}\{e \mid x_1 \leftarrow E_1, \ldots, x_n \leftarrow E_n, P\}$$

yields a nested loop to depth n, with n being the number of involved collections and P denoting a conjunction of filter predicates.

[Won93, Feg94] describe normalization algorithms which produce expressions of the above general form for any comprehension. These algorithms are given by a confluent set of rewrite rules. The core of the rewrite rule system is the following equation

$$\mathcal{M}\{e \mid x \leftarrow E, y \leftarrow \mathcal{N}\{e' \mid x' \leftarrow E'\}\} \quad = \quad \mathcal{M}\{e \mid x \leftarrow E, x' \leftarrow E', y \equiv e'\}$$

that allows for unnesting of the inner comprehension over monoid $\mathcal{N}$ and results in a multi-generator comprehension (the **binding** $y \equiv e'$ makes the variable y a synonym for the expression e' in all subsequent qualifiers and in the head of the comprehension). The normal form avoids the construction of the intermediate results the nested comprehensions would compute (see Section 4.2).

It is important to observe that this rewriting rule—as almost all we will consider—is generic, in the sense that we may replace $\mathcal{M}$ and $\mathcal{N}$ with any monoid. We reason about some sort of template by abstracting types and avoid to maintain a bunch of special rule instantiations for the respective types of our model.

As pointed out before, one is not stuck with a naive nested-loop strategy of comprehension evaluation, which has its algebraic analogy in computing cartesian products. Firstly, certain comprehension forms—*patterns*— can be equivalenty mapped to efficient algorithms, say, for example, materialisation or join strategies like *semijoins* or *nestjoins* [Ste95], of the underlying query engine.

Another way to improve query performance is to analyze the ways data flows through potentially expensive predicates. This gives rise to techniques like the recently proposed *bypass-joins* [SPMK95] which can be smoothly incorporated into the monoid calculus framework.

We will discuss both ways in Section 4.

3 Monoid Comprehensions as a Target for OQL Translation

This section shows that the ODMG OQL constructs can be mapped to the comprehension calculus rather straightforward. As the query language itself, the calculus nests to arbitrary depth in generators, predicates, and constructors. Free variables that (implicitly) occur in OQL `from`-clauses or quantifiers directly correspond with the calculus' generator variables. A complete translation of OQL 1.2 into a variant of the monoid comprehension calculus can be found in Appendix A.

Note that we come up with a precise semantics of OQL this way, thanks to the underlying monoid homomorphisms. As far as we know, there have not been successful efforts, based on object algebras to define the *complete* OQL semantics. What *has* been achieved this way is a formalization of a subset of OQL that can be explained by "classic" algbraic operators, like *selection*, *projection*, several *joins*, or *grouping*. We come back to this in Section 3.1.

Due to the fact that the collections and scalars OQL can manipulate have monoid representations as they was in Section 2 the mapping turns out to be simple.

OQL's `select-from-where` block is equivalently translated into (let $\overline{x}$ be an abbreviation for the variables $x_1, \ldots, x_n$):

$$\begin{array}{rl} \texttt{select} & e(\overline{x}) \\ \texttt{from} & E_1 \,\texttt{as}\, x_1, \ldots, E_n \,\texttt{as}\, x_n \\ \texttt{where} & p(\overline{x}) \qquad\qquad\qquad \mapsto \end{array}$$
$$\{\!\{ e(x_1, \ldots, x_n) \mid x_1 \leftarrow E_1, \ldots, x_n \leftarrow E_n,\ p(x_1, \ldots, x_n) \}\!\}$$

with E_i being an arbitrary collection-valued OQL expression, and p denoting an OQL query of type *bool*. Finally, e represents the constructor used to build up records from the retrieved values or to call actual user-defined type constructors to yield new objects (see [Cat95]). Since the *merge*[*bag*] is used to collect the query's result, we meet the semantics of the OQL `select-from`. Use of the keyword `distinct` would instruct us to employ the *set* monoid instead.

The orthogonal definition of OQL's grammar allows for arbitrary nesting of queries. Nesting in the `from`-clause is one of the degrees of freedom:

$$\begin{array}{rl} \texttt{select} & e(x) \\ \texttt{from} & (\texttt{select} \;\; e'(y) \\ & \qquad\;\; \texttt{from} \;\; E \;\texttt{as}\; y \\ & \qquad\; \texttt{where} \;\; p'(y)) \;\texttt{as}\; x \\ \texttt{where} & p(x) \end{array}$$

We apply the rule to translate a `select-from-where` block twice (first for the inner query, then for the outer query) and get

$$\{\!\{ e(x) \mid x \leftarrow \{\!\{ e'(y) \mid y \leftarrow E, p'(y) \}\!\}, p(x) \}\!\}$$

that is, a nested comprehension. By applying the normalization rewrite rule presented in Section 2.2, we obtain the canonical

$$\{\!\{ e(x) \mid y \leftarrow E, p'(y), x \equiv e'(y), p(x) \}\!\}$$

Even more complicated queries have a rather elegant calculus equivalent. The OQL 1.2 definition [Cat95] features a `group-by-having` construct that resembles the SQL form:

$$
\begin{array}{rl}
\texttt{select} & * \\
\texttt{from} & E_1 \texttt{as}\ x_1, \ldots, E_r \texttt{as}\ x_r \\
\texttt{group by} & l_1 : e_1(\overline{x}), \ldots, l_n : e_n(\overline{x}) \\
\texttt{having} & p \qquad\qquad\qquad\qquad \mapsto
\end{array}
$$

$$
\begin{array}{l}
\{\langle l_1 = e_1(\overline{x}), \ldots, l_n = e_n(\overline{x}), \mathit{partition}\rangle \mid x_1 \leftarrow E_1, \ldots, x_r \leftarrow E_r, \\
\qquad \mathit{partition} \equiv \\
\qquad \{\!\!\{\langle o_1, \ldots, o_r\rangle \mid o_1 \leftarrow E_1, \ldots, o_r \leftarrow E_r, \\
\qquad\qquad e_1(\overline{o}) = e_1(\overline{x}), \ldots, e_n(\overline{o}) = e_n(\overline{x})\}\!\!\}, \\
\qquad p\}
\end{array}
$$

The *partition* attribute for each group carries all objects that deliver the same results for the grouping expressions $e_1, \ldots, e_n$. Naive evaluation of the `group-by`-clause leads to deeply nested loops but the query's principal structure is that of a nesting in the `select`-clause.

Since the "natural" collection operators have a direct counterpart in the calculus, translations can be as simple as

$$
\begin{array}{c}
E_1 \ \texttt{union}\ E_2 \quad \mapsto \\
\mathit{merge}[\mathit{set}](E_1, E_2)
\end{array}
$$

because $\mathit{merge}[\mathit{set}] = \cup$. Intersection is mapped to

$$
\begin{array}{c}
E_1 \ \texttt{intersect}\ E_2 \quad \mapsto \\
\{e \mid e \leftarrow E_1, \mathit{some}\{e' = e \mid e' \leftarrow E_2\}\}
\end{array}
$$

(note how the inner *some* implements the predicate $e \in E_2$).

The type conversion function $\texttt{listtoset}(E)$ that constructs a set containing list E's elements is computed by

$$
\begin{array}{c}
\texttt{listtoset}(E) \quad \mapsto \\
\{e \mid e \leftarrow E\}
\end{array}
$$

in which the implicit $\mathit{merge}[\mathit{set}]$ carries out the necessary duplicate elimination.

The primitive monoids can serve to handle OQL queries that do not operate on collections, namely arithmetic and boolean expressions: *sum* obviously implements OQL's operator `+`, while monoid $\mathit{min} = (\mathit{int\_min},\ \mathit{id},\ \mathit{if}\ a < b\ \mathit{then}\ a\ \mathit{else}\ b)$ serves to cope with the OQL `min` (or analogously `max`).

Although the nature of OQL queries can be quite diverse, they receive a uniform treatment in the monoid comprehension calculus. The monoid $\mathit{all} = (\mathit{true}, \mathrm{id}, \vee)$ allows for the simple translation of universal quantification while the monoid *some* introduced earlier can express collection membership (OQL `in`):

$$
\begin{array}{c}
\texttt{for all}\ e\ \texttt{in}\ E : p(e) \quad \mapsto \\
\mathit{all}\{p(e) \mid e \leftarrow E\}
\end{array}
\qquad\qquad
\begin{array}{c}
E_1\ \texttt{in}\ E_2 \quad \mapsto \\
\mathit{some}\{e = E_1 \mid e \leftarrow E_2\}
\end{array}
$$

As a last example, consider the construction of arbitrarily complex results in the `select` clause, which was not possible with SQL, but is an integral part of OQL:

$$
\begin{array}{rl}
\texttt{select} & \texttt{list}(x\ \texttt{+}\ y\texttt{,}\ x\texttt{,}\ y) \\
\texttt{from} & E_1\ \texttt{as}\ x,\ E_2\ \texttt{as}\ y \qquad \mapsto
\end{array}
$$

$$
\{\!\!\{\mathit{merge}[\mathit{list}](\mathit{merge}[\mathit{sum}](x,y), \mathit{merge}[\mathit{list}](\mathit{unit}[\mathit{list}]x, \mathit{unit}[\mathit{list}]y)) \mid x \leftarrow E_1, y \leftarrow E_2\}\!\!\}
$$

Such arbitrary value constructions lead to complex *nesting* or *grouping* expressions and inherent problems with common subexpression detection (note the multiple use of x and y in the constructor) in the variable-less algebraic approaches. Recent efforts [CM93, Ste95] have enhanced algebras with special operators to cope with the important (but not all) cases. Unfortunately, these operators do enjoy few interesting algebraic properties making them hard to treat during rewrite optimization.

3.1 A Rewriting Example

Now that we have an idea of how to map query constructs to the underlying calculus, let us review a short example.

Consider an OQL query stated against the following logical schema that we will use in the sequel of the paper. An instance of this schema represents a set of films along with their titles and directors as well as its cast of actors, modelled as a list of the actor's names and count of scenes in which they appear (Figure 1).

```
films:  set(struct(title:      string,
                   directors:  bag(string),
                   cast:       list(struct(actors:  string,
                                           scenes:  int ))
                   ))
```

Figure 1: Logical schema of the running example.

The query computes a film's number of scenes in which two given different actors appear (not necessarily together):

```
sum( select  c.scenes                                 sum( select  c.scenes
       from  films as f, f.cast as c            +            from  films as f, f.cast as c
      where  f.title = "Star Wars" and                      where  f.title = "Star Wars" and
             c.actor = "Ford" )                                    c.actor = "Hamill" )
```

This is a typical OQL example in the sense that a mix of bulk accesses (the `select-from-where` blocks), aggregation (`sum`) and operations on scalars (addition of two *int*s via `+`) make up the query as a whole. Optimization approaches that rely on rather classical σ-π-$\bowtie$ algebras have to treat scalar functions like `+` as "black boxes" that are moved around during the query rewriting process: since the optimizers know nothing about `+`'s algebraic properties they are forced to optimize the two `select`s independently. Algebras have been enhanced to handle the application of functions, leading to *map* operators of various forms. However, many approaches suffer from the fact that the function results are not elements of the domains (i.e. bulk types) these languages were invented for. For our simple example, we are forced to enrich our hypothetical algebra with a *fold* operator—which would be needed to compute the `sum` aggregation—and some way to reason about arithmetics.

The monoid calculus representation of this query is different. We first apply the mapping procedure as sketched at the beginning of this section and obtain

$$\begin{aligned} \mathit{merge}[\mathit{sum}](\mathit{sum}\{c.\text{scenes} \mid\ & f \leftarrow \text{films}, c \leftarrow f.\text{cast}, \\ & f.\text{title} = \text{“Star Wars”}, c.\text{actor} = \text{“Ford”}\}, \\ \mathit{sum}\{c.\text{scenes} \mid\ & f \leftarrow \text{films}, c \leftarrow f.\text{cast}, \\ & f.\text{title} = \text{“Star Wars”}, c.\text{actor} = \text{“Hamill”}\}) \end{aligned}$$

The aggregation and final summation ($\mathit{merge}[\mathit{sum}]$) are nothing else than operations on a specific monoid like any other computation in our framework. As a consequence, rewriting rules treat operations on scalars and aggregations as first-class citizens, not as black boxes as mentioned above. We are not forced to optimize the two `select` blocks locally only and may proceed with two applications (from right to left) of the generic equivalence

$$\mathcal{M}\{c \mid e_1, x \leftarrow \mathit{merge}[\mathcal{N}](E_1, E_2), e_2\} \quad = \quad \begin{aligned} & \mathit{merge}[\mathcal{M}](\mathcal{M}\{c \mid e_1, x \leftarrow E_1, e_2\}, \\ & \qquad \mathcal{M}\{c \mid e_1, x \leftarrow E_2, e_2\}) \end{aligned}$$

Instantiating this rule with $\mathcal{M} = sum$, $\mathcal{N} = bag$, respectively $\mathcal{M} = bag$, $\mathcal{N} = some$, and successive application results in

$$\begin{aligned}&sum\{c.\text{scenes} \mid f \leftarrow \text{films}, c \leftarrow f.\text{cast},\\ &\qquad\qquad f.\text{title} = \text{“Star Wars”}, merge[some](c.\text{actor} = \text{“Ford”}, c.\text{actor} = \text{“Hamill”})\}\end{aligned}$$

Remember that $merge[some] = \vee$. This comprehension scans the *films* collection only once and thus is much cheaper with respect to expected I/O costs.

In general, treating all query operators as first-class citizens will result in more complete search spaces for rewrite-based query optimizers. The solution we just obtained is not element of the search space of an σ-π-$\bowtie$ algebra based rewriter.

4 Calculus Evaluation Strategies beyond Nested Loops

If we assume for a second that the comprehension we derived above is the final output of our query optimization phase we could then map this expression to nested **foreach** loops as outlined in Section 2.2. One could even implement a fully-functional OQL query engine by realizing the *zero*, *unit*, and *merge* operations of the respective monoids.

But, as we have mentioned, the nested loop execution strategy is not appropiate in presence of sophisticated join algorithms, indices, and the various fine-tuned physical storage structures known today. Consequently, our major concern in what follows will be the derivation of more efficient execution strategies for monoid comprehensions. We shed some light on this now.

4.1 Bypass-Comprehensions

Careful observation of the flow of data that a specific query execution plan (QEP) defines led to the development of the *bypass technique* [KMPS94, SPMK95]. Bypass plans avoid the evaluation of predicates—which can impose dominating cost on a QEP—whenever this is possible. Special algebraic operators, namely bypass-selections and bypass-joins, generate two disjoint streams of objects: those that qualify with respect to a given predicate and may bypass other (expensive) predicates, and those that do not. The two streams then undergo their own separate optimization until they are merged to make up the final result. These merges are actually cheap because the streams are guaranteed to be disjoint and duplicate elimination is not needed.

The bypass technique was shown to be superior to the well-known CNF and DNF driven strategies. Here, we will show how this advanced query evaluation strategy can be smoothly integrated into the monoid comprehension framework in its most general form.

Recall that the standard mapping from comprehensions to monoid homomorphisms we have shown in Section 2.2 discards elements from the final result as soon as a predicate evaluates to *false*, losing the elements that belong to the “not qualified” stream: $\mathcal{M}\{e \mid p, l\} = \textit{if } p \textit{ then } \mathcal{M}\{e \mid l\}$ $\textit{else } zero[\mathcal{M}]$. We will therefore alter the mapping as follows: (1) use the $pair[\mathcal{M}]$ monoid to model the two streams as a pair of collections, and (2) insert objects that qualify into the first component of this pair, all others into the second.

The following translation scheme consisting of two groups of inductive equations does just that:

$$\overline{\mathcal{M}}\{e \mid \} = \langle unit[\mathcal{M}](e), zero[\mathcal{M}]\rangle \tag{4}$$

$$\overline{\mathcal{M}}\{e \mid x \leftarrow E, l\} = \hom^{\mathcal{N} \mapsto pair[\mathcal{M}]}(\lambda x.\overline{\mathcal{M}}\{e \mid l\})E \tag{5}$$

$$\overline{\mathcal{M}}\{e \mid p, l\} = \textit{if } p \textit{ then } \overline{\mathcal{M}}\{e \mid l\} \textit{ else } \overline{\overline{\mathcal{M}}}\{e \mid l\} \tag{6}$$

and

$$\overline{\overline{\mathcal{M}}}\{e \mid \} = \langle zero[\mathcal{M}], unit[\mathcal{M}](e)\rangle \tag{7}$$

$$\overline{\overline{\mathcal{M}}}\{e \mid x \leftarrow E, l\} = \hom^{\mathcal{N} \mapsto pair[\mathcal{M}]}(\lambda x.\overline{\overline{\mathcal{M}}}\{e \mid l\})E \tag{8}$$

$$\overline{\overline{\mathcal{M}}}\{e \mid p, l\} = \overline{\overline{\mathcal{M}}}\{e \mid l\} \tag{9}$$

Whenever a predicate p turns out to be *false*, equation (6) directs control to equations (7) through (9) which insert *any* element regardless of further predicate outcomes (9) into the second component of the resulting pair (7). We refer to comprehensions that are interpreted this way as **bypass-comprehensions**.

As one can easily see by unfolding, the above scheme can be simplified for comprehensions that are already in normal form (i.e. $\mathcal{M}\{e \mid x_1 \leftarrow E_1, \ldots, x_n \leftarrow E_n, P\}$):

$$\begin{aligned}
\overline{\mathcal{M}}\{e \mid \} &= \langle unit[\mathcal{M}](e), zero[\mathcal{M}]\rangle && (4') \\
\overline{\mathcal{M}}\{e \mid x \leftarrow E, l\} &= \hom^{\mathcal{N}\mapsto pair[\mathcal{M}]}(\lambda x.\overline{\mathcal{M}}\{e \mid l\})E && (5') \\
\overline{\mathcal{M}}\{e \mid p, l\} &= \textit{if } p \textit{ then } \overline{\mathcal{M}}\{e \mid l\} \textit{ else } \langle zero[\mathcal{M}], unit[\mathcal{M}](e)\rangle && (6')
\end{aligned}$$

Let us turn back to the bag comprehension example we examined in Section 2.1 and apply the new mapping:

$$\begin{aligned}
&\overline{bag}\{e \mid e \leftarrow [1,2,3,1], e \neq 2\} \\
&\qquad = \hom^{list\ \mapsto\ pair[bag]}(\lambda e.\overline{bag}\{e \mid e \neq 2\})\ [1,2,3,1] \\
&\qquad = \hom^{list\ \mapsto\ pair[bag]}(\lambda e.\textit{if } e \neq 2 \textit{ then } \overline{bag}\{e \mid \} \textit{ else } \overline{\overline{bag}}\{e \mid \})\ [1,2,3,1] \\
&\qquad = \hom^{list\ \mapsto\ pair[bag]}(\lambda e.\textit{if } e \neq 2 \textit{ then } \langle \{\!\{e\}\!\}, \{\!\{\}\!\}\rangle \textit{ else } \langle \{\!\{\}\!\}, \{\!\{e\}\!\}\rangle)\ [1,2,3,1] \\
&\qquad = \langle \{\!\{1,3,1\}\!\}, \{\!\{2\}\!\}\rangle
\end{aligned}$$

The bypass translation scheme applies to any comprehension expression, making it possible to compute the result and its inverse (with respect to the result one gets for the same comprehension with all predicates set to *true*) in one go. Consider

$$\overline{set}\{(x,y) \mid x \leftarrow [1,2], \text{even}(x), y \leftarrow \{2,3\}, y = x\} \quad = \quad \langle \{(2,2)\}, \{(1,2),(1,3),(2,3)\}\rangle$$

Bypass plans may now be constructed easily. Assume a query that contains a disjunctive predicate whose condition p_2 dominates p_1 significantly in terms of cost:

$$\mathcal{M}\{e \mid e \leftarrow E, p_1 \vee p_2, p_3\}$$

Clearly we would like to bypass the expensive p_2 test for all elements that allow us to. The bypass technique makes it possible:

$$\begin{array}{rl}
\textbf{let} & C = \overline{\mathcal{M}}\{e \mid e \leftarrow E, p_1\} \\
\textbf{in} & \mathcal{M}\{e \mid e \leftarrow \overline{merge[\mathcal{M}]}(\pi_1 C, \mathcal{M}\{e' \mid e' \leftarrow \pi_2 C, p_2\}), p_3\} \\
\textbf{end} &
\end{array}$$

Only those objects that did not pass predicate p_1, which are exactly those in the second component of C, have to be tested against p_2. $\overline{merge[\mathcal{M}]}$ reminds us that we merge disjoint arguments (which can be efficiently implemented and may be a no-op in specific setups). The **let** environment and the projections π_i on the ith component of a pair realize the "fork" in the flow of data (see Figure 2).

Observe that the three atomic predicates p_i can be independently positioned in the flow of data to achieve globally minimum cost, while DNF- or CNF-based plans place boolean summands respectively factors as a whole (e.g. $p_1 \vee p_2$).

There is no need to verify that the participating collections (E in the last example) are non-empty to preserve the OQL semantics of delivering an empty result otherwise, since we have $\mathcal{M}\{e \mid l_1, x \leftarrow zero[\mathcal{N}], l_2\} = zero[\mathcal{M}]$. [SPMK95] enforced these semantics by explicit **if-then-else** clauses that check for empty collections first.

Bypass-comprehensions may be introduced into QEPs whenever this promises to optimize the flow of data through expensive predicates, or subqueries in general. Since we put the comprehensions down to monoid homomorphisms, the usual rewriting and normalization rules apply. In contrast to the algebraic approach where the bypass ideas originated from, we do not need to introduce special bypass-select, bypass-join, or bypass-semijoin operators. But still, the complete theory developed in [KMPS94] and [SPMK95] is applicable.

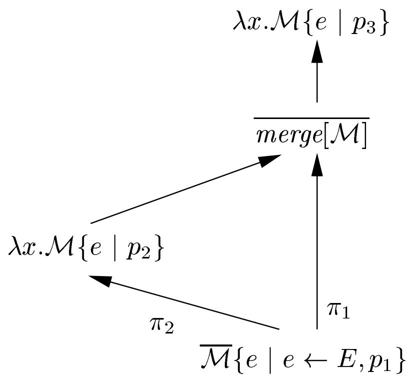

Figure 2: Example bypass plan.

4.2 Executions Plans for Comprehensions

Access paths and algorithms of database query engines tend to be designed for the efficient interaction with query algebras. In fact, core algebra operators, like joins or certain types of selections that may be carried out by index lookups, correspond in a one-to-one manner with query engine primitives. The mapping from queries stated in a conceptual algebra to an appropiate query execution plan stated in some "physical algebra" is therefore rather straightforward.

Achieving the same goal for the monoid calculus is not that apparent. Provided we believe the calculus to be a valuable approach to object query optimization, we face the problem of how to find good QEPs for potentially deeply nested comprehension expressions. In the following we show a way out of this dilemma, presuming that the target we generate execution plans for is of the above mentionend algebraic nature. In doing so, the basics are provided to realize a full-featured OQL interface on top of available storage subsystems now. Advantage can be taken of the sophisticated access paths these backends provide. However, it is a challenge for the future to identify the primitives a comprehension-tailored query engine should feature.

The forthcoming discussion takes place in a setup where *physical data indepedence* is achieved by representing the database physical level in terms of queries (materialized views) over the base extents stored in the database. These queries describe (potentially redundant) configurations of the data itself, access support relations, and materialized functions. A discussion of the relevant issues can be found in [Sch93] or [TSI94], where similar approaches are proposed.

Due to lack of space we focus on **vertical decomposition** as one option for physical design. The monoid calculus can represent the common physical data structures that are prominent in database research today, but decomposition suffices to illustrate the problems and possible solutions in this context.

To vertically decompose a conceptual type means to partition its set of attributes into different physical files. The number of partitions is a measure of how (de)clustered instances of this type are stored. Using only one partition results in a fully clustered storage. Storing potentially large attributes (i.e. collections) in separate files has several advantages:

- If the query engine does not need to read all attributes to answer a certain query, it is possible to leave whole partitions untouched and to avoid the unnecessary reading of files and thus avoid the unnecessary file I/O, and
- scanning a declustered collection is generally faster because its elements may be stored physically contiguous (not scattered, clustered with their parent objects).

Partitions may be overlapping (contain the same attributes), thus introducing redundancy. Decomposition rises the question of how one reassembles the conceptual objects from the several partitions again. This is where sophisticated join strategies come into play. Here, we consider the possibility of using record identifiers (think of *TIDs* used in the relational context). We denote the address of a record r as $r.\#$, i.e. handling it as a conceptual invisible attribute. Depending on where the record identifier is stored (in the partition itself or its parent or both), we may then use value-based joins to reassemble an object from its parts.

A reasonable option of decomposing type film (see Section 3.1) would be to store the cast and the directors separately. Let the partitions carry the record identifier of their parent object record:

$$\begin{array}{rll}
\text{Films:}\ \ \mathit{set}(\mathit{struct}(\text{title:} & \mathit{string}\ )) \\
\text{Cast:}\ \ \mathit{list}(\mathit{struct}(\text{actor}: & \mathit{string}, \\
\text{scenes:} & \mathit{int} \\
\text{film:} & \#\text{Films}\ )) \\
\text{Directors:}\ \ \mathit{bag}(\mathit{struct}(\text{director:} & \mathit{string}, \\
\text{film:} & \#\text{Films}\ ))
\end{array}$$

The complete reassembly comprehension involves the respective scans over the partitions and two joins. At the same time this query serves as a description of the mapping of our database physical level.

$$\begin{array}{lll}
\text{films} & = & \{\langle \text{title} = f_p.\text{title}, \\
& & \quad \text{directors} = \{\!\{ d_p.\text{director} \mid d_p \leftarrow \text{Directors}, d_p.\text{film} = f_p.\# \}\!\}, \\
& & \quad \text{cast} = [\langle \text{actor} = c_p.\text{actor}, \text{scenes} = c_p.\text{scenes} \rangle \mid c_p \leftarrow \text{Cast}, \\
& & \qquad\qquad\qquad c_p.\text{film} = f_p.\#] \rangle \mid f_p \leftarrow \text{Films}\}
\end{array}$$

The potential of this approach lies in the fact that a *complete* reassembly is not necessary in general. Ideally, we choose a physical representation that provides just the attributes referenced in a (sub)query, thus saving I/O costs. If we are lucky, accessing the OIDs suffices.

Now let us examine another query against the logical schema in Figure 1 and then view it in the light of the chosen physical storage configuration. The query retrieves the titles of all films that have an actor, appearing in exactly one scene, that is also one of the directors of the film in question (this is of special interest because we like Alfred Hitchcock):

```
select distinct  f.title
           from  films as f
          where  exists d in f.directors : d in ( select  c.actor
                                                    from  f.cast as c
                                                   where  c.scenes = 1)
```

This provides an example of nesting in the `where`-clause, where we compute, for each film f, the bag of actors appearing exactly once. The outer selection checks if at least one of the directors' names appears in this collection and, if the condition holds, outputs this film's title.

Applying the translation rules of Section 3 results in the following initial comprehension expression:

$$\begin{array}{l}
\{f.\text{title} \mid f \leftarrow \text{films}, \\
\qquad \mathit{some}\{\mathit{some}\{d = c.\text{actor} \mid c \leftarrow f.\text{cast}, c.\text{scenes} = 1\} \mid d \leftarrow f.\text{directors}\}\}
\end{array}$$

The nested query in the `where`-clause shows up as a nested comprehension serving as a predicate against the stream of film objects. Note that the monoid *some* is used to implement the `in` element test as well as the existential quantification we used in the OQL query.

Mapping a query to its initial physical counterpart is now accomplished by replacing collections by their reassembly expressions. We obtain a query that reflects the actual physical configuration of the database and is against stored collections only:

$$\begin{array}{l}
\{f.\text{title} \mid f \leftarrow \{\langle \text{title} = f_p.\text{title}, \\
\qquad\qquad \text{directors} = \{\!\{ d_p.\text{director} \mid d_p \leftarrow \text{Directors}, d_p.\text{film} = f_p.\# \}\!\}, \\
\qquad\qquad \text{cast} = [\langle \text{actor} = c_p.\text{actor}, \text{scenes} = c_p.\text{scenes} \rangle \mid c_p \leftarrow \text{Cast}, \\
\qquad\qquad\qquad\qquad c_p.\text{film} = f_p.\#] \rangle \mid f_p \leftarrow \text{Films}\}, \\
\qquad\qquad \mathit{some}\{\mathit{some}\{d = c.\text{actor} \mid c \leftarrow f.\text{cast}, c.\text{scenes} = 1\} \mid d \leftarrow f.\text{directors}\}\}
\end{array}$$

How are we supposed to deduce efficient implementations from a complex and deeply nested comprehension like this?

Since the nested subqueries directly correspond to the computation of potentially complex intermediary results, we aim for unnesting these by virtue of the equivalence

$$\mathcal{M}\{e \mid e_1, x \leftarrow E, y \leftarrow \mathcal{N}\{e' \mid x' \leftarrow E', e_2\}\} \quad = \quad \mathcal{M}\{e \mid e_1, x \leftarrow E, x' \leftarrow E', y \equiv e', e_2\}$$

which we introduced as the core of the comprehension normalization in Section 2.2. The comprehension normalization algorithm is given as a set of confluent rewrite rules—of which the above equivalence is a part—and serves as the basic heuristic in our translation process. Unnesting realizes a pipelined evaluation, and serves to avoid the build of the intermediary result y ranges over (we omit several of the unnesting steps here for brevity):

$$\begin{array}{ll}
\multicolumn{2}{l}{\{f.\text{title} \mid f_p \leftarrow \text{Films},} \\
& f \equiv \langle \text{title} = f_p.\text{title}, \\
& \quad \text{directors} = \{\!\{ d_p.\text{director} \mid d_p \leftarrow \text{Directors}, d_p.\text{film} = f_p.\# \}\!\}, \\
& \quad \text{cast} = [\langle \text{actor} = c_p.\text{actor}, \text{scenes} = c_p.\text{scenes}\rangle \mid c_p \leftarrow \text{Cast}, \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c_p.\text{film} = f_p.\#]\rangle, \\
& \mathit{some}\{\mathit{some}\{d = c.\text{actor} \mid c \leftarrow f.\text{cast}, c.\text{scenes} = 1\} \mid d \leftarrow f.\text{directors}\}\} \\
\multicolumn{2}{l}{\ldots\ (*)} \\
\equiv & \{f_p.\text{title} \mid f_p \leftarrow \text{Films}, \\
& \quad d_p \leftarrow \text{Directors}, d_p.\text{film} = f_p.\#, \\
& \quad \mathit{some}\{d_p.\text{director} = c_p.\text{actor} \mid c_p \leftarrow \text{Cast}, c_p.\text{film} = f_p.\#, c_p.\text{scenes} = 1\}\}
\end{array}$$

Unnest the *some*:

$$\begin{array}{ll}
\equiv & \{f_p.\text{title} \mid f_p \leftarrow \text{Films}, d_p \leftarrow \text{Directors}, c_p \leftarrow \text{Cast}, \\
& \quad d_p.\text{film} = f_p.\#, c_p.\text{film} = f_p.\#, c_p.\text{scenes} = 1, d_p.\text{director} = c_p.\text{actor}\}
\end{array}$$

As expected, the unnesting process constructs the characteristic normal form with all generators put first, followed by a conjunction of predicates (actually a predicate in CNF). The canonical query engine's (algebraic) interpretation of this expresssion, as for every comprehension in normal form, is that of a left-deep tree of cartesian products ($\times$):

$$\pi_{f_p.\text{title}}(\sigma_{d_p.\text{film}=f_p.\# \wedge c_p.\text{film}=f_p.\# \wedge c_p.\text{scenes}=1 \wedge d_p.\text{director}=c_p.\text{actor}}((\text{Films}_{f_p} \times \text{Directors}_{d_p}) \times \text{Cast}_{c_p}))$$

However, the normal form can serve as a basis for query optimization in System R style [SAC+88], in the sense that the exchange of two generators implements the reordering of join inputs and that placement of predicates next to generators that bind their variables transforms a cartesian product into a real join. Consequently, we are guaranteed not to end up with plans worse than System R style optimizers would produce, in our case

$$\begin{array}{l}
\{f_p.\text{title} \mid f_p \leftarrow \text{Films}, d_p \leftarrow \text{Directors}, f_p.\# = d_p.\text{film}, \\
\qquad\qquad c_p \leftarrow \text{Cast}, c_p.\text{scenes} = 1, \\
\qquad\qquad f_p.\# = c_p.\text{film}, d_p.\text{director} = c_p.\text{actor}\}
\end{array}$$

which is

$$\pi_{f_p.\text{title}}((\text{Films}_{f_p} \underset{f_p.\#=d_p.\text{film}}{\bowtie} \text{Directors}_{d_p}) \underset{f_p.\#=c_p.\text{film} \wedge d_p.\text{director}=c_p.\text{actor}}{\bowtie} (\sigma_{c_p.\text{scenes}=1}(\text{Cast}_{c_p})))$$

4.3 Detecting Access Patterns

We just saw that certain comprehension terms directly correspond to algebra operators. Let us consider this correspondence in its reverse direction. Then the join $E_1 \underset{p}{\bowtie} E_2$ is obviously computed by (let $\oplus$ denote the *merge* operation of a suitable record monoid):

$$E_1 \underset{p}{\bowtie} E_2 \quad = \quad \{x \oplus y \mid x \leftarrow E_1, y \leftarrow E_2, p\}$$

but non-standard, often significantly more efficient join operators have an equivalent calculus form too, e.g. the family of hierarchical joins [RRS93], the *semijoin* $\ltimes$, and the *nestjoin* Δ [Ste95] (see below).

The semijoin $E_1 \underset{p}{\ltimes} E_2$ is a join variant that delivers only those left operand objects that have at least one join partner with respect to the semijoin predicate p. Implementation can be efficient in terms of space (buffers or files that hold the intermediate result only carry E_1 objects) and time (as soon as any join partner is found for an E_1 object, it belongs to the result; no other E_2 object has to be touched):

$$E_1 \underset{p}{\ltimes} E_2 \quad = \quad \{x \mid x \leftarrow E_1, some\{p \mid y \leftarrow E_2\}\}$$

Note, that neither E_1, E_2, nor the join result need to be of type αset as in relational theory.

The above mentioned nestjoin operator Δ, as a final example, allows to join an object of the left operand E_1 with the whole *group* g of matching right operand (E_2) objects, which is exactly what is needed for the evaluation of queries with nesting in the `select`-clause:

$$E_1 \underset{x,y:f(x),p(x,y),c(x,y);g}{\Delta} E_2 \quad = \quad \{f(x) \oplus \langle g = \{c(x,y) \mid y \leftarrow E_2, p(x,y)\}\rangle \mid x \leftarrow E_1\}$$

See [SABdB94] for a detailed discussion of the nestjoin's characteristics. [CM93] achieves functionality similar to the nestjoin by introducing a *binary grouping* operator.

Now, revisiting the query rewriting above, we detect the characteristic **pattern** of a semijoin at $(*)$. By matching the semijoin's defining comprehension against the query we recognize a subquery that can be efficiently computed by a semijoin:

$$\begin{aligned}&\{d_p \mid d_p \leftarrow \text{Directors},\\ &\quad some\{d_p.\text{director} = c_p.\text{actor} \mid c_p \leftarrow \text{Cast}, c_p.\text{film} = d_p.\text{film}, c_p.\text{scenes} = 1\}\}\\ &\qquad = \quad \text{Directors} \underset{c_p.\text{actor}=d_p.\text{director} \wedge d_p.\text{film}=c_p.\text{film} \wedge c_p.\text{scenes}=1}{\ltimes} \text{Cast}\end{aligned}$$

Simple rewriting (predicate pushdown for $c_p.\text{scenes} = 1$) provides us with a query that takes advantage of the partitioned storage with respect to Cast. We can restrict Cast before joining and obtain (using the pattern for the algebraic selection $\sigma_p(E) = \{e \mid e \leftarrow E, p\}$):

$$\begin{aligned}&\{d_p \mid d_p \leftarrow \text{Directors},\\ &\quad some\{d_p.\text{director} = c_p.\text{actor} \mid c_p \leftarrow \text{Cast}, c_p.\text{scenes} = 1, c_p.\text{film} = d_p.\text{film}\}\}\\ &\qquad = \quad \text{Directors}_{d_p} \underset{c_p.\text{actor}=d_p.\text{director} \wedge d_p.\text{film}=c_p.\text{film}}{\ltimes} (\sigma_{c_p.\text{scenes}=1}(\text{Cast}_{c_p}))\end{aligned}$$

Since we do not want the optimization process to stop here, we have to find a way of remembering the subquery's implementation in further rewriting steps. Thus, we annotate the subquery with its algebraic equivalent by *replacing* the comprehension with the algebraic expression, obtaining a **hybrid** query notation, similar to the one proposed in [Nak90]:

$$\begin{aligned}\{f_p.\text{title} \mid\ & f_p \leftarrow \text{Films},\\ & d_p \leftarrow (\text{Directors}_{d_p} \underset{c_p.\text{actor}=d_p.\text{director} \wedge d_p.\text{film}=c_p.\text{film}}{\ltimes} (\sigma_{c_p.\text{scenes}=1}(\text{Cast}_{c_p}))),\\ & d_p.\text{film} = f_p.\#\}\end{aligned}$$

The join pattern matching proceedes and once again we detect that a semijoin is applicable since the query's result contains attributes stored in file Films only:

$$\text{Films}_{f_p} \underset{f_p.\#=d_p.\text{film}}{\ltimes} (\text{Directors}_{d_p} \underset{c_p.\text{actor}=d_p.\text{director} \wedge d_p.\text{film}=c_p.\text{film}}{\ltimes} (\sigma_{c_p.\text{scenes}=1}(\text{Cast}_{c_p})))$$

No further projection is necessary, the query requested all attributes in Films.

We managed to rewrite the query into an efficient semijoin-combination that also accounts for the partitioned storage schema of the film objects. The core strategy is to detect the patterns of efficient algebraic operations before exhaustive unnesting flattens out the query's structure until all information on (potentially uncorrelated) subqueries is lost. This is of even more importance since uncorrelated subqueries indicate the query engine that access can be physically localized in a partitioned storage design: During rewriting of the conceptual query the detection of the semijoin pattern in

$$\{f.\text{title} \mid f \leftarrow \text{films},\ d \leftarrow f.\text{directors},\ \mathit{some}\{d = c.\text{actor} \mid c \leftarrow f.\text{cast}, c.\text{scenes} = 1\}\ \}$$

would trigger the query engine to select a storage design with directors and cast stored in their own partitions, thus choosing a physical schema like the one we presumed.

Annotating parts of a query with its respective algebraic equivalent can be done in an iterative manner, replacing more and more of the calculus expressions until a pure algebraic representation is reached.

As a consequence, an optimizing query translator should map OQL to a *hybrid mix* of calculus and algebra, since some OQL expressions have an obvious algebraic character (e.g. the set operators `union`, `intersect`, and `except`, some selections, and of course joins), reducing the cost of pattern matching at the same time. Problems of value-construction, quantification and general purpose computations are best expressed in the calculus. Putting the algebra operators down to their calculus equivalents allows for a semantically clean interaction of the two formalisms.

Turning this approach into a complete hybrid query optimizing strategy means, firstly, to find the calculus equivalents for the algebra operators of the underlying database engine, in particular for those algorithms that promise an efficient evaluation of the problematic nesting cases we discovered. Special monoids like $\mathit{sorted}[f]$, whose *merge* operator performs a sorted set insert according to function f [Feg93], can represent algebra operators that rely on physical properties (like sorting) of their input, e.g. the *sort-merge-join*.

Secondly, the query language to monoid calculus mapping is performed. We then start unnesting following the normalization approach, aiming (1) to avoid the assembly of unnecessarily deep nested intermediate results and, more importantly, (2) to get hold of uncorrelated subqueries (in presence of a particular storage design) that match the pattern of efficient query engine algorithms.

The rewriting phase is complete when a pure annotated form is obtained. However, we cannot really ensure that the complete query execution plan can be carried out by the storage backend's primitives. Post-processing for those monoid operations that have no realization in the query engine must be scheduled. As mentioned before, this situation will improve with comprehension-tailored query processors.

5 Conclusion and Future Work

Facing the richer typed current object models, the monoid now plays the role that the set played for relational data models. Due to the fact that OQL is a language that can be mapped to the calculus rather straight-forward, we have now obtained a uniform framework for query translation, rewriting for optimization, and plan generation. Besides providing precise meaning for the query language with its diverse constructs, we gain an *operational semantics* for it. We have shown that it is indeed possible to bridge the gap between conceptual queries and efficient implementations for expressions given in the monoid calculus. We presented techniques for deducing the applicability of certain join types, annotating subqueries with their algebraic equivalents. Recent developments in the field, like bypass plans, fit smoothly into the picture. From our point of view, this proves the monoid comprehension calculus a valuable formalism for current and future query processing challenges.

Work in progress and future efforts concern several issues. We are currently underway developing an object algebra which is especially designed to interact with the monoid calculus in a seamless manner. Once completed, it will help to promote the hybrid approach we suggested in Section 4.3. There will be a many-to-many mapping of the algebra operators to the calculus that will be the input to the operator pattern matching process. [GS96] explores the degrees of freedom a reasonable physical database model for complex structured objects may feature while still being efficiently maintainable. This work employs the comprehension formalism as its storage definition language and will contribute to our efforts as well. The recent work of [Ste95] and several others on rule-based query optimization already proved to provide valuable input to these questions. Another field of interest is to use comprehensions as a starting point for the tight integration of functional programming languages—some of which already feature comprehensions as language constructs—and databases.

A prototype of an OQL query translator and rewriter is already operational, employing the sophisticated pattern matching facilities of the functional programming language SML.

Acknowledgements. We are grateful to Holger Riedel, Dieter Gluche, Andreas Heuer, Joachim Kröger, and Sophie Cluet for several discussions and valuable comments on a previous version of this paper.

References

[BLS+94] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. In *SIGMOD Record*, volume 23, pages 87–96, March 1994.

[Bun93] Peter Buneman. The Fast Fourier Transform as a Database Query. Technical Report MS-CIS-93-37/L&C 60, University of Pennsylvania, CIS Dept., March 1993.

[Cat93] Rick Cattell, editor. *The Object Database Standard: ODMG–93.* Morgan Kaufmann Publishers, 1993. Release 1.1.

[Cat95] The Object Database Standard: ODMG–93. Chapter 4, Object Query Language (Release 1.2), pages 65–69. Updates to Release 1.1, 1995.

[CM93] Sophie Cluet and Guido Moerkotte. Nested Queries in Object Bases. In *Proceedings of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, September 1993.

[Feg93] Leonidas Fegaras. Efficient Optimization of Iterative Queries. In *Proceedings 4th Int'l Workshop on Database Programming Languages*, pages 200–225, August 1993.

[Feg94] Leonidas Fegaras. A Uniform Calculus for Collection Types. Technical Report 94-030, Oregon Graduate Institute of Sciene & Technology, 1994.

[FM95a] Leonidas Fegaras and David Maier. An Algebraic Framework for Physical OODB Design. Technical report, Oregon Graduate Institute of Science & Technology, 1995.

[FM95b] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *1995 ACM SIGMOD International Conference on Management of Data*, May 1995.

[GS96] Dieter Gluche and Marc H. Scholl. Physical OODB Design Exploiting Replication. Technical Report in preparation, Department of Mathematics and Computer Science, Database Research Group, University of Constance, 1996.

[KMPS94] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing Disjunctive Queries with Expensive Predicates. In *Proceedings of the ACM SIGMOD Conference*, pages 336–347, Minneapolis, USA, 1994.

[Nak90] Ryohei Nakano. Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions. *ACM Transactions on Database Systems*, 15(4):518–557, December 1990.

[RRS93] Christian Rich, Arnon Rosenthal, and Marc H. Scholl. Reducing Duplicate Work in Relational Join(s): A Unified Approach. In *Proceedings of the Int'l Conference on Information Systems and Management of Data (CISMOD)*, pages 87–102, Delhi, India, October 1993.

[SABdB94] Hennie J. Steenhagen, Peter M.G. Apers, Henk M. Blanken, and Rolf A. de By. From Nested-Loop to Join Queries in OODB. In *Proceedings of the 20th VLDB Conference, Santiago, Chile*, pages 618–629, September 1994.

[SAC+88] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 82–93, 1988.

[Sch93] Marc H. Scholl. Physical Database Design for an Object-Oriented Database System. In J.C. Freytag, G. Vossen, and D.E. Maier, editors, *Query Processing for Advanced Database Applications*, chapter 14, pages 419–447. Morgan Kaufmann Publishers, 1993.

[SPMK95] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing Joins in Disjunctive Queries. In *Proceedings of the 21st Int'l Conference on Very Large Databases (VLDB)*, September 1995.

[Ste95] Hennie Steenhagen. *Optimization of Object Query Languages*. PhD thesis, Department of Computer Science, University of Twente, 1995.

[Tri91] Phil Trinder. Comprehenions, a Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*, pages 55–68, 1991.

[TSI94] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *Proceedings of the 20th Int'l Conference on Very Large Databases (VLDB)*, pages 367–378, Santiago, Chile, September 1994.

[TW89] Phil Trinder and Phil Wadler. Improving List Comprehension Database Queries. In *Proceedings of TENCON'89, Bombay, India*, pages 186–192, November 1989.

[Van92] Bennet Vance. Towards an Object-Oriented Query Algebra. Technical Report 91–008, Oregon Graduate Institute of Sciene & Technology, January 1992.

[Wad90] Phil Wadler. Comprehending Monads. In *Conference on Lisp and Functional Programming*, pages 61–78, June 1990.

[Won93] Limsoon Wong. Normal Forms and Conservative Properties for Query Languages over Collection Types. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pages 26–36, May 1993.

[WT91] David A. Watt and Phil Trinder. Towards a Theory of Bulk Types. Technical Report FIDE/91/26, Computer Science Department, University of Glasgow, 1991.

A Mapping OQL 1.2 to the Monoid Calculus

The (informal) proof that the monoid comprehension calculus is able to represent and compute any OQL query is open until now and we use this section of the appendix to fill this gap. We will do so by describing a mapping that is able to translate each OQL construct to a comprehension equivalent form. Since the comprehension calculus is a fully orthogonal language, we may then map any OQL query to a equivalent calculus expression by translating each syntactic construct of the query seperately and combine these pieces later.

Our language reference is the description of OQL's second release (OQL 1.2) which can be found in [Cat95].

A.1 Granularity of the Mapping

Query processing in object-oriented environments differs from, say, the relational case, in that operations on collection types (i.e. *set*, *bag*, *list*) may freely be intermixed with manipulation of scalar types, like arithmetics or calls to boolean functions (termed *general-purpose computations* in [Van92]). The efficient handling of the bulk operations that OQL can express is our main concern here. Operations on scalars are considered to be cheap and executable in constant time, whilst the size of a collection instance cannot be foreseen.

As a consequence, we will map constants of the built-in scalar types like *int*, *string*, or *bool* and operations on these (e.g. `+`, `*`) directly to the calculus with no further special treatment. Here, we assume that the query engine is able to perform these computations. The same is true for OQL's comparison operators `=`, `!=`, `<`, `<=`, `>=`, `>` if these get applied to scalars. However, this does not apply to the aggregation functions like `sum` or `avg`. Even though these express arithmetic operations, they aggregate the elements of a potentially huge collection and therefore deserve special attention.

Let us not proceed without noting that the monoid calculus *can* capture operations on scalars (cf. the discussion of the *int* and *bool* monoids in Section 2). The key to efficient query processing, however, is the optimized manipulation of bulk types.

The treatment of arrays in the monoid calculus suffers from the fact that there are no agreed upon primitives for the manipulation of indexable collections (arrays). We will represent arrays as lists of fixed length, as lists share the major characteristic of arrays with respect to query processing: the elements in these collections are accessed in order. As long as we have no satisfactory representation for operations like random array access by index, it will be the responsibility of the query engine to map these primitives to list accesses.

The following is influenced by the work of Leonidas Fegaras and David Maier [FM95a] on translating OQL 1.1 to the monoid calculus.

A.2 Mapping OQL 1.2 to the Monoid Calculus

In the sequel, let e_i and E_i denote arbitrary OQL queries that are correctly typed in the context they are used. We will not concern ourselves with typing issues here. In particular, let E_f be a filter, i.e. a query returning a boolean; x, x_i, and y represent variable names; the l_i denote valid labels.

A.2.1 Construction

`struct`.

$$\texttt{struct(}l_1\texttt{:}E_1\texttt{,}\ldots\texttt{,}l_n\texttt{:}E_n\texttt{)} \quad \mapsto \quad \langle l_1 = E_1, \ldots, l_n = E_n \rangle$$

`set`.

$$\texttt{set(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)} \quad \mapsto \quad \{E_1, \ldots, E_n\}$$

As already pointed out in Section 2, $\mathcal{M}\{E_1, \ldots E_n\}$ is a convenient notation for the explicit *merge*s over the E_i:

$$\mathit{merge}[\mathcal{M}](\mathit{merge}[\mathcal{M}](\ldots(\mathit{merge}[\mathcal{M}](\mathit{unit}[\mathcal{M}]E_1, \mathit{unit}[\mathcal{M}]E_2)\ldots), \mathit{unit}[\mathcal{M}]E_n)$$

$\mathcal{M}$ being *set*, *bag*, or *list*.

`bag`.

$$\texttt{bag(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)} \quad \mapsto \quad \{\!\{E_1, \ldots, E_n\}\!\}$$

`list`.

$$\texttt{list(}E_1\texttt{,}\ldots\texttt{,}E_n\texttt{)} \quad \mapsto$$
$$[E_1, \ldots, E_n]$$

A.2.2 `select-from-where` blocks

`select-from-where`.

$$\begin{array}{rl} \texttt{select} & E \\ \texttt{from} & E_1 \texttt{ as } x_1\texttt{, } \ldots \texttt{ , } E_n \texttt{ as } x_n \\ \texttt{where} & E_f \qquad\qquad\qquad \mapsto \end{array}$$
$$\{\!\{E \mid x_1 \leftarrow E_1, \ldots, x_n \leftarrow E_n, E_f\}\!\}$$

Omitting the optional `where`-clause is allowed and makes the predicate query E_f become the constant *true*.

`select-distinct-from-where`.

$$\begin{array}{rl} \texttt{select distinct} & E \\ \texttt{from} & E_1 \texttt{ as } x_1\texttt{, } \ldots \texttt{ , } E_n \texttt{ as } x_n \\ \texttt{where} & E_f \qquad\qquad\qquad \mapsto \end{array}$$
$$\{E \mid x_1 \leftarrow E_1, \ldots, x_n \leftarrow E_n, E_f\}$$

`select-from-group-by`.

$$\begin{array}{rl} \texttt{select} & E \\ \texttt{from} & E_1 \texttt{ as } x_1\texttt{, } \ldots \texttt{ , } E_r \texttt{ as } x_r \\ \texttt{group by} & l_1\texttt{:}e_1\texttt{, } \ldots \texttt{ , } l_n\texttt{:}e_n \qquad \mapsto \end{array}$$
$$\begin{array}{l} \{\!\{E \mid x \leftarrow \{\langle l_1 = e_1, \ldots, l_n = e_n, \mathit{partition}\rangle \mid x_1 \leftarrow E_1, \ldots, x_r \leftarrow E_r, \\ \qquad \mathit{partition} \equiv \\ \qquad \{\!\{\langle o_1, \ldots, o_r\rangle \mid o_1 \leftarrow E_1, \ldots, o_r \leftarrow E_r, \\ \qquad\qquad e_1(\overline{o}) = e_1(\overline{x}), \ldots, e_n(\overline{o}) = e_n(\overline{x})\}\!\}\}\}\!\} \end{array}$$

`select-from-group-by-having`.

$$\begin{array}{rl} \texttt{select} & E \\ \texttt{from} & E_1 \texttt{ as } x_1\texttt{, } \ldots \texttt{ , } E_r \texttt{ as } x_r \\ \texttt{group by} & l_1\texttt{:}e_1\texttt{, } \ldots \texttt{ , } l_n\texttt{:}e_n \\ \texttt{having} & E_f \qquad\qquad\qquad \mapsto \end{array}$$
$$\begin{array}{l} \{\!\{E \mid x \leftarrow \{\langle l_1 = e_1, \ldots, l_n = e_n, \mathit{partition}\rangle \mid x_1 \leftarrow E_1, \ldots, x_r \leftarrow E_r, \\ \qquad \mathit{partition} \equiv \\ \qquad \{\!\{\langle o_1, \ldots, o_r\rangle \mid o_1 \leftarrow E_1, \ldots, o_r \leftarrow E_r, \\ \qquad\qquad e_1(\overline{o}) = e_1(\overline{x}), \ldots, e_n(\overline{o}) = e_n(\overline{x})\}\!\}, \\ \qquad E_f\}\}\!\} \end{array}$$

`order by`. Let E be a `select-from` query of one of the types above.

$$E\texttt{order by } E_c \quad \mapsto$$
$$\mathit{sorted}[E_c]\{x \mid x \leftarrow E\}$$

where the sort criterion's (E_c) type features a partial ordering $\leq$ which is used to perform a sorted list insert of E's elements.

$\mathit{merge}[\mathit{sorted}[f]](x, y)$ could be defined as follows:

$$\mathit{merge}[\mathit{sorted}[f]](x, y) = \mathit{if}\ f(x) \leq f(y)\,\mathit{then}\ x \mathbin{+\!\!+} y\,\mathit{else}\ y \mathbin{+\!\!+} x$$

Sorting by more than one criterion is implemented analogously. The effect of specifying the modifier `desc` is achieved by using $\leq^{-1}$ instead of $\leq$ as the partial ordering. See [FM95a] for further comments on the $\mathit{sorted}[f]$ monoid.

A.2.3 Set operators

`union`.

$$E_1 \texttt{union}\ E_2 \quad \mapsto$$
$$merge[set](E_1, E_2)$$

`intersect`.

$$E_1 \texttt{intersect}\ E2 \quad \mapsto$$
$$\{x \mid x \leftarrow E_1, x \in E_2\}$$

Note, that the predicate $e \in E$ may be computed by $some\{x = e \mid x \leftarrow E\}$. See below.

`except`.

$$E_1 \texttt{except}\ E_2 \quad \mapsto$$
$$\{x \mid x \leftarrow E_1, all\{x \neq y \mid y \leftarrow E_2\}\}$$

In a hybrid framework (see Section 4), it is questionable if it is still beneficial to translate the above set operations into calculus expressions. An adequate object *algebra* will certainly feature $\cup$, $\cap$, and $\setminus$ directly. However, the explicit representation has its advantages. Simple rewriting, as in

$$\begin{aligned} & E_1 \texttt{intersect}\ E_2 \\ & \quad \mapsto \quad \{x \mid x \leftarrow E_1, x \in E_2\} \\ & \quad \equiv \quad \{x \mid x \leftarrow E_1, some\{x = y \mid y \leftarrow E_2\}\} \\ & \quad \equiv \quad E_1 \underset{x,y:x=y}{\ltimes} E_2 \end{aligned}$$

may reveal, most notable in interaction with other operators, efficient implementation alternatives.

A.2.4 Quantification, Membership Testing

`for all in`.

$$\texttt{for all}\ x \texttt{in}\ E_1 : E_2 \quad \mapsto$$
$$all\{E_2 \mid x \leftarrow E_1\}$$

`exists in`.

$$\texttt{exists}\ x \texttt{in}\ E_1 : E_2 \quad \mapsto$$
$$some\{E_2 \mid x \leftarrow E_1\}$$

E_2's result has to be of type *bool*.

`exists`.

$$\texttt{exists}(E) \quad \mapsto$$
$$some\{true \mid x \leftarrow E\}$$

Remarks in [Cat95] with respect to the evaluation of `exists` apply here, too: the optimizer has to detect that the complete evaluation of E is not necessary to answer the `exists` query.

`unique`.

$$\texttt{unique}(E) \quad \mapsto$$
$$sum\{1 \mid x \leftarrow E\} = 1$$

`any`. The keyword `some` may be used as a synonym.

$$E_1\ \theta \texttt{any}\ E_2 \quad \mapsto$$
$$some\{E_1\ \theta' x \mid x \leftarrow E_2\}$$

where $\theta \in \{\texttt{=}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>=}, \texttt{>}\}$ is mapped to $\theta' \in \{=, \neq, <, \leq, \geq, >\}$ as expected.

`all`.

$$E_1\ \theta \texttt{all}\ E_2 \quad \mapsto$$
$$all\{E_1\ \theta' x \mid x \leftarrow E_2\}$$

`in`.

$$E_1 \texttt{in}\ E_2 \quad \mapsto$$
$$some\{x = E_1 \mid x \leftarrow E_2\}$$

A.2.5 Aggregation

`sum`.

$$\begin{array}{c}\texttt{sum(}E\texttt{)} \quad \mapsto \\ sum\{x \mid x \leftarrow E\}\end{array}$$

`avg`.

$$\begin{array}{c}\texttt{avg(}E\texttt{)} \quad \mapsto \\ avg\{x \mid x \leftarrow E\}\end{array}$$

with the monoid $avg = (zero[avg], unit[avg], merge[avg])$ being defined by the three equations

$$\begin{aligned} zero[avg] &= \langle sum = 0, cnt = 0\rangle \\ unit[avg]a &= \langle sum = a, cnt = 1\rangle \\ merge[avg](\langle sum = a, cnt = n\rangle, \langle sum = b, cnt = m\rangle) &= \langle sum = a + b, cnt = n + m\rangle \end{aligned}$$

from which the average is derived immediately.

`count`.

$$\begin{array}{c}\texttt{count(}E\texttt{)} \quad \mapsto \\ sum\{1 \mid x \leftarrow E\}\end{array}$$

`max`.

$$\begin{array}{c}\texttt{max(}E\texttt{)} \quad \mapsto \\ max\{x \mid x \leftarrow E\}\end{array}$$

`min`.

$$\begin{array}{c}\texttt{min(}E\texttt{)} \quad \mapsto \\ min\{x \mid x \leftarrow E\}\end{array}$$

with *max* respectively *min* being the monoids introduced in Section 3.

A.2.6 Type Conversion

`flatten`. If the argument query E is of type $\alpha\ c_2\ c_1$ (c_i a type constructor), the result of `flatten(`E`)` is computed by

$$\begin{array}{c}\texttt{flatten(}E\texttt{)} \quad \mapsto \\ \mathcal{M}\{y \mid x \leftarrow E, y \leftarrow x\}\end{array}$$

where monoid $\mathcal{M}$ is determined by the table below:

		c_2		
		set	*bag*	*list*
	set	*set*	*set*	*set*
c_1	*bag*	*bag*	*bag*	*bag*
	list	*set*	*bag*	*list*

`listtoset`. Query E is expected to deliver a result of type α`list`.

$$\begin{array}{c}\texttt{listtoset(}E\texttt{)} \quad \mapsto \\ \{x \mid x \leftarrow E\}\end{array}$$

`distinct`. Dependent on the result type of E, we evaluate `distinct(`E`)` as follows

$$\begin{array}{c}\texttt{distinct(}E\texttt{)} \quad \mapsto \\ \mathcal{M}\{x \mid x \leftarrow E\}\end{array}$$

and choose $\mathcal{M}$ as *set* respective *oset* if E is of type αset or αbag respective $\alpha list$. Monoid *oset* represents lists without duplicates [FM95a], i.e. $merge[oset]$ is idempotent.

`element`. `element(`E`)` is expected to extract the sole member e of any singleton E and to raise an exception otherwise (i.e. E being $zero[\mathcal{M}]$ or containing two or more elements). Raising an exception is modeled by the commutative *single* monoid as returning a unique *null* value:

$$\begin{aligned} zero[single] &= null \\ unit[single]a &= a \\ merge[single](a, b) &= null \end{aligned}$$

Thus, we translate as shown below

$$\begin{array}{c} \texttt{element(}E\texttt{)} \quad \mapsto \\ single\{x \mid x \leftarrow E\} \end{array}$$

A.2.7 Boolean operations

The boolean connectives `and`/`or` have their calculus counterparts in $merge[all]$ and $merge[some]$, respectively. Note, that specifying the two predicates E_{f_1}, E_{f_2} as filters in a comprehension (in that order) is equivalent to evaluating (E_{f_1} *and-then* E_{f_2}).

Negating a filter via `not(`E_f`)` is safe, because variables do not get used before they are bound to a generator's domain. The translation of `not(`E_f`)` is therefore $\neg E_f$.

Constants *true* and *false* may be represented by the identity elemens $zero[all]$ and $zero[some]$, where this seems advantageous with respect to further rewriting.