First publ. in : Journal of Intelligent Information Systems ; 12 (1999), 3. - S. 191-218
DOI : 10.1023/A:1008705026446

How to Comprehend Queries Functionally

TORSTEN GRUST torsten.grust@uni-konstanz.de
MARC H. SCHOLL marc.scholl@uni-konstanz.de
Department of Computer and Information Science, University of Konstanz, D-78457 Konstanz, Germany

Editors: Peter M.D. Gray, Peter J.H. King and Larry Kerschberg

Abstract. Compilers and optimizers for declarative query languages use some form of intermediate language to represent user-level queries. The advent of compositional query languages for orthogonal type systems (e.g., OQL) calls for internal query representations beyond extensions of relational algebra. This work adopts a view of query processing which is greatly influenced by ideas from the functional programming domain. A uniform formal framework is presented which covers all query translation phases, including user-level query language compilation, query optimization, and execution plan generation. We pursue the type-based design—based on initial algebras—of a core functional language which is then developed into an intermediate representation that fits the needs of advanced query processing. Based on the principle of structural recursion we extend the language by monad comprehensions (which provide us with a calculus-style sublanguage that proves to be useful during the optimization of nested queries) and combinators (abstractions of the query operators implemented by the underlying target query engine). Due to its functional nature, the language is susceptible to program transformation techniques that were developed by the functional programming as well as the functional data model communities. We show how database query processing can substantially benefit from these techniques.

Keywords: databases, query optimization, structural recursion, monad comprehensions, program transformation

1. Intermediate query language representations

This work focuses on the internal representation and optimization of query languages for databases supporting complex data models. It is one dominant characteristic of these query languages to be *declarative*, i.e., they exclusively include constructs that allow to describe *what* to retrieve from the persistent store. The language is not concerned with the (procedural) description of *how* the store has to be accessed in order to efficiently answer a given query. A query compiler and optimizer maps declarative queries into programs which are then used to efficiently access the store for retrieval.

The choice of an adequate intermediate query representation has a major impact on the overall query compilation and optimization process. Once fixed, the intermediate language determines which class of queries can be represented at all, which parts of the representation may be analyzed or remain "black boxes" in later stages, and which equivalence preserving transformations may be employed during the optimization phase. To be more precise, this is what an adequate intermediate language is expected to provide, at least:

(*Ex*) Capability of expressing any user query (this affects query operators as well as the type system).

(*Abs***)** Abstraction from user-level queries while providing a granularity of representation that makes all relevant query language constructs subject to inspection and transformation.

(*Eq***)** A well understood equational theory that decides the equivalence preservation of expression transformations.

(*Map***)** A suitable starting point for a mapping to the access primitives of the underlying persistent store.

Algebraic approaches to the representation of declarative query languages dominate the field of research by far. The query algebra operators are rather straightforward abstractions of the algorithms implemented by the underlying query engine. This asserts desideratum (*Map*). Relational join $\bowtie$, for example, is an abstraction that reduces join algorithms like *sort-merge join* or *hash join* to their basic algebraic properties (e.g., being commutative in their two inputs). These properties are then exploited during the algebraic optimization stage of the query compiler. Relational algebraic query optimization is a deeply investigated and well understood area of research for which (*Eq*) has clearly been established.

Relational query languages, SQL in particular, have been designed to fit the abstractions provided by relational algebra: it is a quite manageable task to map the SQL `select-from-where` block to an intermediate algebraic σ-π-$\bowtie$ representation (which asserts (*Ex*) for this subset of SQL).

However, SQL—and this is even more true for recent query language proposals like OQL (Cattell and Barry, 1997)—includes concepts which are not inspired by bulk-oriented set processing and therefore not easily mapped into an algebraic equivalent: most algebras fail to provide canonical forms for quantification, aggregation, arithmetics, and the orthogonal nesting of these query constructs (note that these are language concepts which especially play a role in data warehouse and OLAP queries). This mismatch between the query language and its intermediate representation makes SQL compilation and optimization a non-straightforward and sometimes error-prone process, a prominent example being the infamous `count` bug of (Kim, 1982) in which nested queries involving aggregate functions were transformed in order to obtain a "canonical [algebraic] n-relation query". Additionally, SQL and relational algebra suffer from a type system mismatch: while SQL is a language predominantly defined on *bags*, relational algebra processes sets.

The interaction of extra-relational concepts with the relational bulk processing is often unclear. During the query rewrite phase, these operators are therefore moved around as "black boxes" whose algebraic properties remain unexploited. As a result, store access programs typically exhibit a separate bulk-processing phase for which an optimizer can find an efficient arrangement of operators. The output of this phase is then passed on to an extra aggregation phase which is opaque to the query optimizer. These partitioned access plans inhibit a global optimization so that (*Map*) is not fully met. An optimizer for a language in which these concepts may be naturally represented would have the choice of folding these two phases into one, thus saving the cost of (1) storing the intermediate result, and (2) performing the aggregation in a separate loop.

Furthermore, predicates are often mere annotations to algebraic operators and not part of the core language which renders them as atomic and therefore inaccessible to (*Eq*) (query optimization frameworks nevertheless provide means for predicate manipulation, but these are not expressible in the intermediate language itself and therefore considered *ad-hoc* in our context). Consequently, desiderata (*Ex*) as well as (*Abs*) are not met by relational algebra if the *full* query language is taken as a standard.

Numerous solutions to these problems have been proposed. To assert (*Map*) for fully orthogonal languages, nesting and grouping abstractions like *nestjoin* $\triangle$ (Steenhagen, 1995), *binary grouping* Γ (Cluet and Moerkotte, 1993), or hierarchical joins (Rich et al., 1993) were introduced which do not add expressive power to the algebras but are mainly useful for optimization purposes. These operators enjoy few algebraic properties, however, which considerably complicates the transformation of subexpressions in which they occur.

Query calculi provide a means of reestablishing (*Ex*) and (*Abs*) since complex predicates and quantifiers are canonically expressed (often normal forms are derivable) and subject to transformation in such representations. Calculus expression are not easily mapped into efficient access programs (often implemented as expressions over a *physical algebra*), though.

Finally, the advanced type systems of OQL-like query languages presumably have the greatest impact on the design of an adequate intermediate language. Bulk type constructors like *bag* and *list* may be applied orthogonally to construct arbitrarily nested complex values. Results from the relational query processing domain may or may not hold for these extended type systems. A class of language proposals that try to overcome this difficulty (Buneman et al., 1995; Vance, 1992; Wong, 1994; Fegaras and Maier, 1995) have been organized around *types*, not *operations*: the domain of a type τ is understood as an algebra itself. The intermediate language includes the operations (or *constructors* in the important case of initial algebras) of this algebra in order to be able to manipulate complex values of type τ. This view leads to a considerably different approach to intermediate languages and we will elaborate this idea in Section 2.

1.1. Functional sublanguages as intermediate languages

It is the main claim of this article that *functional languages* and the related program transformation theory provide an adequate environment (ensuring (*Ex*) through (*Map*)) for the compilation, optimization, and derivation of access programs for orthogonal query languages with advanced type systems.

To form an intermediate query language, the expressiveness of a complete functional language is not needed. Query languages provide rather uniform idioms of data access so that a small set of *combinators*[1] suffices to represent queries. We have found a large number of (relational and object) algebra operators introduced by related work to be instances of these combinators. This lets us benefit from already developed transformation strategies based on these operators. We will extend this combinator sublanguage with syntactic sugar, *comprehensions*, to obtain the intermediate representation we will use throughout this article. Comprehensions are easily reduced to an equivalent combinator expression (by means of a simple translation scheme sometimes referred to as the “Wadler identities” (Wadler, 1990a)) so that we are free to mix and match the representation forms.

However, full functional languages have been successfully used as intermediate query languages (Poulovassilis and King, 1990; Poulovassilis and Small, 1996). Although the authors define complete database programming languages they do not sacrifice criterion (*Eq*): functional languages are usually defined by a mapping to a core language based on the λ-calculus which comes with an extensive equational theory.

Adopting the functional style, we will form complex queries by functional composition of higher order combinators. As long as typing rules are obeyed, combinators may be freely composed which naturally fits the orthogonality of recent user query language proposals. The use of combinators leads to a *collectionful style* of programming, which is a well understood class of programs in the field of functional languages. The access program derivation phase will benefit from this knowledge (see below).

The idea of a type-based language design goes well with our aim of using a functional intermediate language. Functional languages naturally operate over recursively defined algebraic types by means of *structural recursion*. We will define collection type constructors *set*, *bag*, and *list* so that the collection type's domain together with its associated constructor functions form initial algebras. The initiality then guarantees the well-definedness of a restricted, yet expressive, form of structural recursion over these types (Goguen et al., 1978). Many program transformations hold for expressions of such algebras in general, so that rewriting rules may abstract from the actual collection type they are valid for.[2]

Additionally, aggregation, quantification, and arithmetics are naturally understood in terms of initial algebras. Since comprehensions are uniformly definable over these algebras we will obtain an expressive calculus-like sublanguage, the *comprehension calculus* (Fegaras and Maier, 1995; Buneman et al., 1994; Wong, 1994; Wadler, 1990a). It will provide us with tools to canonically represent and effectively reason about complex predicates, including quantifiers, as well as collection processing.

This functional view makes query processing subject to the extensive body on program transformation—especially the work commonly known as the *Bird-Meertens*' *formalism*, or *squiggol* (Bird, 1987), which developed a theory of program transformation techniques for an intensionally small family of generic combinators. The results tend to be *list*-biased, but this is no inherent limitation. Many transformations can be generalized and adapted to be valid for other algebraic type constructors.

The access program derivation stage of the query compiler will be based on a specific program transformation, *deforestation*, which strives for removal of intermediate data structures a typical combinator query allocates during execution. This finally leads to a stream-based (or pipelined) query execution which avoids the I/O of temporary data. The derivation of pipelining programs from algebraic queries has been an area which has mainly been tackled on the implementation level only, e.g., by defining stream-based implementation disciplines for query operators (Graefe, 1993). A notable exception is the work of (Freytag and Goodman, 1986a, 1989) which exploits program transformations to derive iterative programs for relational queries. We will show that *cheap deforestation* (Gill et al., 1993; Gill, 1996) covers and extends these ideas, leads to results faster, and avoids inherent complications of the former approaches.

The different stages of the query compilation process will provide the skeleton for the rest of this article. Section 2 will introduce the type system and the *insertion representation* of collection construction which we will use in the sequel. Comprehensions and basic combinators complete the intermediate language and are presented in this section, too. ODMG's OQL will be used to exemplify the compilation of an actual user-level query language but this is not principal to the method. The heuristic optimization of expressions of this language is discussed in Section 3. Section 4 then employs a generalization of cheap deforestation to derive pipelining programs from combinator queries. Conclusions and sketches of further work on how the functional approach to query compilation might evolve are finally given in Section 5. We will use a Haskell-style (Hudak et al., 1992) notation for the sake of readability.

2. A type-based query representation

Given a specific type τ, which functions are necessary to operate on values of τ? The design of a language over τ should be *complete* in the sense that any value of type τ may be constructed using the language's functions. At the same time the language should avoid to include "junk", i.e., functions that may be defined by composition of more basic building blocks.

Earlier we said that the types of our intermediate language will be organized as algebras. For algebras, the vague intuition of the previous paragraph amounts to the algebra being *initial*: the algebra only contains elements that can be built from its *constructors* and given *constants*, and only those equalities provable between constructed elements hold. Most importantly, initiality implies that these algebras support a form of *structural recursion* over their constructors.[3] These are the main observations which lead to the design of the core of our intermediate representation: the language includes the constructors of all supported algebraic types and their associated structural recursion operators. We could thus say that the language has a *type-based design*.

(This is in contrast to an *operation-based* design in which the user-level query operators determine the intermediate language to a large extent. A typical example is the extension of relational algebra with *nest* (ν) and *unnest* (μ) to an algebra for non-first normal form relations (Schek and Scholl, 1986). It is not obvious that the nested relational algebra with ν and μ is complete (in fact it is not) or includes "junk" in the above sense. Operators ν and μ will indeed be derivable in our language.)

Finite values of the polymorphic type *list* α may be built by finitely many applications of two constructor functions to elements of type α: the constant (or 0-ary function) *empty list* $[\,]$ and *list construction* (:) which is commonly pronounced *cons*. For example (constructor (:) is defined to be right-associative so that parentheses may be omitted):

$$[x_1, x_2, \ldots, x_n] = x_1 : (x_2 : (\cdots (x_n : [\,]) \cdots)) = x_1 : x_2 : \cdots : x_n : [\,]$$

The constructors $[\,]$ and $(:)$ are tightly connected to *list* in the sense that the algebra ($\mathit{list}\,\alpha$, $[\,]$, $:$) is initial for the parameterized algebraic data type specification *list* α with signature $(\mathit{list}\,\alpha, \mathit{nil} :: \mathit{list}\,\alpha, \mathit{cons} :: \alpha \to \mathit{list}\,\alpha \to \mathit{list}\,\alpha)$. The principle extends to other algebras. Finitely many insertions into the empty set $\{\}$ or into the empty bag $\{\!\{\}\!\}$ construct any finite set or bag respectively. We will adopt this *insertion representation* for collections (Suciu and Wong, 1995; Buneman et al., 1995) and denote the resulting algebra for type constructor τ by $(\tau\ \alpha,\ \texttt{[]}^{\tau},\ \overset{\tau}{:})$ for the sake of notational uniformity, e.g., $\texttt{[]}^{set} = \{\}$ while $x \overset{set}{:} xs$ inserts $x :: \alpha$ into the set xs (read symbol :: as "of type"). Similarly, the set $\{x_1, \ldots, x_n\}$ will often be written as $\texttt{[}x_1, \ldots, x_n\texttt{]}^{set}$ in what follows.

The initiality of the algebras gives us the assertion that the constructors $\texttt{[]}^{\tau}$ and $(\overset{\tau}{:})$ are already sufficient to operate over values of type $\tau\ \alpha$. In particular, there is no need for *set union* $\cup$ since this operation is already derivable from $\texttt{[]}^{set}$ and $(\overset{set}{:})$. This is not a deep insight at all:[4]

$$\begin{array}{ll} (\cup) & ::\ set\ \alpha \to set\ \alpha \to set\ \alpha \\ \texttt{[]}^{set} \cup ys & = ys \\ (x \overset{set}{:} xs) \cup ys & = x \overset{set}{:} (xs \cup ys) \end{array} \tag{1}$$

Definitions of *list append* (++) and *additive union for bags* (⊎) are derived likewise.

The above two equations define $\cup$ in terms of *structural recursion* over the constructors $\texttt{[]}^{set}$ and $(\overset{set}{:})$ (which, again, is well-defined because of initiality and therefore justified). The number of constructors determines the number of cases that a function defined by structural recursion has to consider. This is the main reason why we chose the insertion representation for algebraic data types in this article. Indeed, $(set\ \alpha,\ \{\},\ \lambda e.\ \{e\}, \cup)$ is an initial algebra for the polymorphic set constructor in *union representation* with signature $(set\ \alpha, zero :: set\ \alpha, unit :: \alpha \to set\ \alpha, merge :: set\ \alpha \to set\ \alpha \to set\ \alpha)$. The closely related work of Fegaras and Maier (1995), for example, adopted the union representation while Buneman et al. (1995) exploits both viewpoints. Issues of expressiveness are examined by Suciu and Wong (1995) and an in-depth comparison of insertion and union representation has been undertaken by Breazu-Tannen and Subrahmanyam (1991). We prefer the insertion presentation because it gets by with two constructors instead of three.

Aggregation and quantification may be uniformly understood in this model as well. To define the summation aggregate `sum` we employ the algebra $sum = (num,\ \texttt{0},\ \texttt{+})$ which uses the domain of the basic (built-in) numeric type *num* as its carrier. The maximum aggregate is represented by $max = (num,\ -\infty_{num}, \texttt{max})$ where `max` computes the maximum of two elements of type *num*. Universal and existential quantifiers are implemented with the help of $all = (bool, \texttt{True}, \texttt{\&\&})$ and $exists = (bool, \texttt{False}, \texttt{||})$, respectively. An existential quantifier is then readily obtained by defining the higher-order operator $(\exists)$ via structural recursion:

$$\begin{array}{ll} \exists^{\tau} & ::\ (\alpha \to bool) \to \tau\ \alpha \to bool \\ \exists^{\tau}\ p\ \texttt{[]}^{\tau} & = \texttt{False} \\ \exists^{\tau}\ p\ (x \overset{\tau}{:} xs) & = p\ x\ \texttt{||}\ \exists^{\tau}\ p\ xs \end{array} \tag{2}$$

The quantifier carries the collection type τ it operates on as an annotation, a device that will be useful in later stages (see Section 4). A type checker could infer the annotation automatically. For the time being the annotations may be understood as type abstractions in the sense of second-order λ-calculus.

The initial algebra approach provides us with a language that can uniformly represent computations over values of different collection types but also gives a canonical representation of aggregation and quantification. Unlike in the relational algebra, these concepts are therefore no longer “black boxes” to the intermediate representation but are accessible to equational reasoning (examples of which are given in Section 3 on query rewriting). This greatly helps to establish (*Abs*).

2.1. Structural recursion

Note that the function definitions (1) and (2) exhibit a common pattern of structural recursion. A large number of optimizations for expressions built by function composition, esp. *loop fusion* laws (Freytag and Goodman, 1986a; Grant, 1989; Fegaras, 1993; Poulovassilis and Small, 1997), can be derived by observing that the involved functions are defined by this recursion scheme. The access program derivation phase of Section 4 heavily relies on this observation. It is thus sensible to express the structural recursion scheme by an extra higher-order combinator, *fold right* or `foldr`, to make the use of structural recursion explicit to the query compiler:

$$\begin{array}{ll} \mathtt{foldr}^{\tau} & :: (\alpha \to \beta \to \beta) \to \beta \to \tau\ \alpha \to \beta \\ \mathtt{foldr}^{\tau}\ (\oplus)\ z\ \mathtt{[]}^{\tau} & = z \\ \mathtt{foldr}^{\tau}\ (\oplus)\ z\ (x \overset{\tau}{:} xs) & = x \oplus (\mathtt{foldr}^{\tau}\ (\oplus)\ z\ xs) \end{array} \tag{3}$$

Using `foldr`, the set union $xs \cup ys$ is expressed as $\mathtt{foldr}^{set}\ (\lambda x\ xs.x \overset{set}{:} xs) ys\ xs$. $\mathtt{foldr}^{\tau}\ (\lambda x\ xs.p\ x \mid\mid xs)\ \mathtt{False}$ implements $\exists^{\tau}\ p$.

`foldr` directly corresponds with the *structural recursion on the insertion representation*, or “*sri*”, of (Breazu-Tannen et al., 1992) and (Suciu and Wong, 1995) in which the companion recursion scheme “*sru*” on the above mentioned union representation of algebras is introduced, too.

The effect of applying $\mathtt{foldr}^{\tau}\ (\oplus)\ z$ to the collection $\mathtt{[}x_1, x_2, \ldots, x_n\mathtt{]}^{\tau}$ may also be understood by “visualizing” the application and its result as follows:

$$\begin{array}{lccccccccc} \mathtt{foldr}^{\tau}(\oplus)\ z\ (& x_1 & \overset{\tau}{:} & (x_2 & \overset{\tau}{:} & (\cdots(& x_n & \overset{\tau}{:} & \mathtt{[]}^{\tau} &)\cdots))) \\ & \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \\ \qquad\qquad = (& x_1 & \oplus & (x_2 & \oplus & (\cdots(& x_n & \oplus & z &)\cdots))) \end{array} \tag{4}$$

$\mathtt{foldr}^{\tau}$ folds $(\oplus)$ in between the collection elements by replacing the list constructors $\overset{\tau}{:}$ and $\mathtt{[]}^{\tau}$ by $(\oplus)$ and z, respectively. $\mathtt{foldr}^{\tau}$ is only conditionally well-defined which can be seen easily from (4). The type constructors *set* and *bag*, as they were given above, are not yet completely specified. We require $(\overset{bag}{:})$ and $(\overset{set}{:})$ to be *left-commutative*, the latter also *left-idempotent*, and thus extend the algebraic data type specifications to reflect these requirements:

$$\begin{array}{l} y \overset{set}{:} (x \overset{set}{:} xs) = x \overset{set}{:} (y \overset{set}{:} xs) \\ x \overset{set}{:} (x \overset{set}{:} xs) = x \overset{set}{:} xs \end{array} \quad \text{and} \quad y \overset{bag}{:} (x \overset{bag}{:} xs) = x \overset{bag}{:} (y \overset{bag}{:} xs) \tag{5}$$

$$
\begin{array}{lcll}
e & \rightarrow & v & \text{variables} \\
 & | & \mathtt{c} & \text{constants} \\
 & | & (e_1, \ldots, e_n) & \text{tuples } (n \geq 1) \\
 & | & e_1 \; e_2 & \text{application} \\
 & | & \lambda vs.e & \text{abstraction} \\
 & | & \mathtt{foldr}^{\tau} \; (\oplus) \; z & \text{reduction} \\
 & | & \mathtt{[]}^{\tau} \mid \overset{\tau}{:} & \text{constructors} \\
 & | & \mathtt{if} \; e_1 \; \mathtt{then} \; e_2 \; \mathtt{else} \; e_3 & \text{conditional} \\
 & | & \mathtt{let} \; v = e_1 \; \mathtt{in} \; e_2 & \text{local bindings} \\
 & | & (op) \; e_1 \ldots e_n & \text{primitives } (n \geq 0, op = \mathtt{+}, \mathtt{*}, \mid\mid, \ldots)
\end{array}
$$

$$
\begin{array}{lcl}
\tau & \rightarrow & set \mid bag \mid list \mid exists \mid all \quad \text{algebraic data types} \\
 & | & sum \mid prod \mid max \mid min
\end{array}
$$

Figure 1. Core representation language.

The equations of (5) establish the provable equalities between elements of the algebras *set* and *bag* which have been constructed by potentially different constructor expressions. Thus, in order for $\mathtt{foldr}^{\tau}$ $(\oplus)$ z to be well-defined, $(\oplus)$ has to be commutative or idempotent whenever $(\overset{\tau}{:})$ is left-commutative or left-idempotent respectively. This ensures the meaning of $\mathtt{foldr}^{\tau}$ $(\oplus)$ z to be independent of the actual construction of its argument.

It is shown below that the *comprehension calculus* sublanguage of our intermediate representation merely provides convenient syntactic sugar for (nested) applications of `foldr`. The same is true for the algebraic combinators which are introduced—for optimization purposes only—in the next section. This implies that a complete query compiler could be solely based on a language organized around `foldr`. A variant of this idea has been worked out in (Grust et al., 1997) to obtain an optimizing compiler for OQL. For this reason and to summarize how far we have got until now, an overview of the core intermediate query language is given in figure 1.[5] Records are internally represented as tuples.

2.2. Comprehensions

Comprehensions have proven to be a very convenient target for the compilation of declarative query languages (Paton and Gray, 1990; Trinder, 1991; Wong, 1994; Fegaras and Maier, 1995; Grust et al., 1997). Just like user-level query languages, comprehensions are equipped with a notion of variables, variable bindings, and orthogonal nesting. While the compilation of a query language like OQL into a variable-free algebraic equivalent can be complicated, the language is easily translated into *monad comprehensions* via a syntactic mapping (Grust et al., 1997).

In the comprehension $[e \mid q_1, \ldots, q_n]^{\tau}$ the *qualifiers* q_i are either *generators* $v \leftarrow q$ or *filters* (expressions of result type *bool*) p. A generator $q_i = v \leftarrow q$ sequentially binds variable v to elements of its *range* q; v is bound in $q_{i+1}, \ldots, q_n$ and e. The binding of v is propagated until a filter evaluates to `False` under the binding. The *head expression*

e is evaluated under all bindings that pass all qualifiers and the evaluation results are then accumulated by $(\overset{\tau}{:})$. As a first example, consider the comprehension $\texttt{[}x \mid x \leftarrow xs,\ p\ x\texttt{]}^\tau$ which operates similar to the relational selection $\sigma_p(xs)$ but is uniformly applicable to any data type τ. Note that for $\tau = set$ the relational calculus can be looked at as a specialization of the monad comprehension calculus. A number of examples for comprehensions are given in Section 2.3 which sketches the translation of OQL into monad comprehensions.

Comprehensions are definable over any type exhibiting the properties of a *monad with zero* (Wadler, 1990a). The algebraic data types that form the basis of our intermediate language indeed induce monad instances. Using the notation of Wadler's article, we can derive a monad (*zero*, *unit*, *map*, *join*) from the algebraic data type τ with constructors $\texttt{[]}^\tau$ and $(\overset{\tau}{:})$ as follows:

$$\begin{aligned} zero^\tau &= \lambda x.\,\texttt{[]}^\tau \\ unit^\tau x &= x \overset{\tau}{:} \texttt{[]}^\tau \\ map^\tau f &= \texttt{foldr}^\tau\ (\lambda x\ xs.\, f\ x \overset{\tau}{:} xs)\ \texttt{[]}^\tau \\ join^\tau &= \texttt{foldr}^\tau\ (\overset{\tau}{+\!\!+})\ \texttt{[]}^\tau \end{aligned} \tag{6}$$

where $(\overset{\tau}{+\!\!+})$ denotes the *union* operation (see (1)) for type τ. The required monad laws of (Wadler, 1990a) are easily shown to be fulfilled by rather straightforward proofs based on structural recursion.

The exact semantics of a monad comprehension can be defined by reducing it—by means of the so-called "Wadler identities"—to the above monad operations. Translation scheme $\mathcal{MC}$ uses a simple recursion over the syntactic structure of the comprehenion qualifier list to implement the comprehension desugaring (let p be an expression of result type *bool*, q, *qs* qualifiers, and e an arbitrary expression):

$$\begin{aligned} \mathcal{MC}\ \texttt{[}e \mid \texttt{]}^\tau &= unit^\tau(\mathcal{MC}\ e) \\ \mathcal{MC}\ \texttt{[}e \mid x \leftarrow q :: \sigma\texttt{]}^\tau &= \texttt{foldr}^\tau(\overset{\tau}{:})\texttt{[]}^\tau(map^\sigma(\lambda x.\mathcal{MC}\ e)\ (\mathcal{MC}\ q)) \\ \mathcal{MC}\ \texttt{[}e \mid p\texttt{]}^\tau &= \texttt{if}\ \mathcal{MC}\ p\ \texttt{then}\ unit^\tau(\mathcal{MC}\ e)\ \texttt{else}\ zero^\tau() \\ \mathcal{MC}\ \texttt{[}e \mid q, qs\texttt{]}^\tau &= join^\tau(\mathcal{MC}\ \texttt{[}\mathcal{MC}\ \texttt{[}e \mid qs\texttt{]}^\tau \mid q\texttt{]}^\tau) \\ \mathcal{MC}\ e &= e \end{aligned} \tag{7}$$

The above example comprehension is mapped to

$$\mathcal{MC}\ \texttt{[}x \mid x \leftarrow xs,\ p\ x\texttt{]}^\tau = join^\tau(map^\tau(\lambda x.\texttt{if}\ p\ x\ \texttt{then}\ unit^\tau\ x\ \texttt{else}\ zero^\tau())\ xs)$$

which implements selection by mapping those elements to $\texttt{[]}^\tau$ that fail to satisfy the filter p (the $\texttt{[]}^\tau$ do not contribute to the result during the outer $join^\tau$ because they are the identity of $\overset{\tau}{+\!\!+}$ as we have seen before).

Since $\mathcal{MC}$ reduces a generator $v \leftarrow q$ occurring in a comprehension over monad τ to a map^σ (which in turn reduces to $\texttt{foldr}^\sigma$, see (6)) over the generator domain $q :: \sigma$ we have to impose the well-definedness condition for $\texttt{foldr}^\tau$ on monad comprehensions, too: whenever $(\overset{\sigma}{:})$ is left-commutative, left-idempotent, or both, we require $(\overset{\tau}{:})$ to have at least the same properties. This rules out the non-deterministic conversion of a set into a list, as in $\texttt{[}x \mid x \leftarrow \texttt{[}1, 2, 3\texttt{]}^{set}\texttt{]}^{list}$, for example.

2.2.1. Related work. In (Fegaras and Maier, 1995; Fegaras, 1994), a comprehension notation is derived for any *monoid*, i.e., an algebraic structure that comes equipped with an associative operator $+\!\!+$ having the distinguished element *zero* as its left and right identity (an additional *unit* function lifting elements to singleton collections is required if a monoid is supposed to model a collection type). As monoids are just the algebras for a monad τ enriched by $\overset{\tau}{+\!\!+}$ and $zero^{\tau}$, *monoid comprehensions* constitute a specific instance of monad comprehensions for a monad with $\overset{\tau}{+\!\!+}$ and $zero^{\tau}$. We have seen that the monads τ induced by the collection type constructors *list*, *bag*, and *set* may be naturally enriched by $\overset{\tau}{+\!\!+}$ and $zero^{\tau}$.

Monad comprehensions, however, are already sensibly defined for any non-enriched monad ($unit^{\tau}, map^{\tau}, join^{\tau}$),[6] and the use of monads goes well beyond the modelling of collection type constructors. Ohori (1990) defines a *state transformer monad ST* which is used to transform a database programming language for objects with identity (modelled as mutable references) and destructive updates into a referentially transparent language being more amenable to optimization. References are implemented as the contents of a store (of type *S*) on which operations (*state transformers*) like reference creation (`new`), assignment (`asgn`), and read access (`deref`) are implemented. The referentially transparent program is required to pass the state of the store between function invocations. The *ST* monad takes care of the correct state passing and a computation expressed as a monad comprehension over *ST* may act *as if* a mutable store would be available to the program.

Monad *ST* maps any computation resulting in a value of type α onto a function that receives the state of the store as its input and emits the new state as well as its original result: $ST\ \alpha = S \to (\alpha, S)$. Viewed from the outside, such a function behaves like a computation having access to a mutable store. Functions $unit^{ST}$, map^{ST}, and $join^{ST}$ implement the state passing as follows:

$$\begin{array}{ll}
unit^{ST} & :: \alpha \to ST\ \alpha \\
unit^{ST}\ x & = \lambda s.(x, s) \\
map^{ST} & :: (\alpha \to \beta) \to ST\ \alpha \to ST\ \beta \\
map^{ST}\ f\ s' & = \lambda s.\mathtt{let}\ (x, s'') = s'\ s\ \mathtt{in}\ (f\ x, s'') \\
join^{ST} & :: ST\ (ST\ \alpha) \to ST\ \alpha \\
join^{ST}\ s' & = \lambda s.\mathtt{let}\ (x, s'') = s'\ s\ \mathtt{in}\ x\ s''
\end{array} \tag{8}$$

While $unit^{ST}$ simply returns its argument and does not modify the store, state transforming computations can be sequenced by means of $join^{ST}$. Comprehensions over the thus defined monad *ST* provide a succinct notation for computations interacting with a store. In the comprehension

$$[\,p \mid x \leftarrow \mathtt{new}\ 0, y \leftarrow \mathtt{new}\ 1, z \leftarrow \mathtt{deref}\ x, p \leftarrow \mathtt{asgn}\ y\ z\,]^{ST}$$

a generator $v \leftarrow q$ applies the state transformer q to the current state s, yielding a new state s' which is implicitly passed down the qualifier list as the new current state while v gets bound to the computation's result value. After evaluation of the comprehension above, the reference associated with y is assigned the same value as reference x.

2.3. *Mapping OQL to monad comprehensions*

As we cannot provide more than an intuition of the OQL to monad comprehension calculus mapping here, we refer to work reported in (Grust et al., 1997; Grust and Scholl, 1996) where translations for the whole set of OQL clauses were developed. A mapping in the reverse direction (from list comprehensions to SQL) is discussed in (Kemp et al., 1994). In (Gluche et al., 1997) we devised an incremental maintenance algorithm for OQL views which could be conveniently formulated at the monad comprehension level.

The principal OQL construct, the `select-from-where` block, closely resembles a comprehension: the `from` clause translates into a sequence of generators, while predicates in the `where` clause introduce filters. Finally, the `select` clause corresponds to the comprehension's head expression. The core of the OQL mapping $\mathcal{Q}$ thus reads

$$\mathcal{Q}\left(\begin{array}{l}\texttt{select}\ e\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1\,,\ \ldots\,,e_n\ \texttt{as}\ x_n\\ \ \ \texttt{where}\ p\end{array}\right) = [\,\mathcal{Q}\ e \mid x_1 \leftarrow \mathcal{Q}\ e_1, \ldots, x_n \leftarrow \mathcal{Q}\ e_n,\ \mathcal{Q}\ p\,]^{bag} \tag{9}$$

(the x_i appear free in e and p). Use of the `distinct` modifier would trigger the choice of *set* instead of *bag* as the result monad. Figure 2 summarizes the translation of further OQL clauses. The simplicity of $\mathcal{Q}$ is mainly due to its *uniformity* (Suciu and Wong, 1995): a query clause e may be compiled independently from subqueries e_i occurring in it. During the translation of e the e_i are treated as free variables that may be instantiated later to complete the translation.

Example. As a final example consider the following query involving aggregation, selection, join, and grouping (sum up the employees' salaries in those departments which are low on budget):

```
    select sum (e.sal)
      from emp as e, dept as d
     where e.dno = d.no and d.budg < 10000
  group by d.no
```
(10)

$$\begin{array}{lcl}
\mathcal{Q}(\texttt{forall}\ x\ \texttt{in}\ e : p) & = & [\,(\mathcal{Q}\ p)\ x \mid x \leftarrow \mathcal{Q}\ e]^{all}\\
\mathcal{Q}(\texttt{exists}(e)) & = & [\,\texttt{True} \mid x \leftarrow \mathcal{Q}\ e]^{exists}\\
\mathcal{Q}(e_1\ \texttt{in}\ e_2) & = & [\,\mathcal{Q}\ e_1 = x \mid x \leftarrow \mathcal{Q}\ e_2]^{exists}\\
\mathcal{Q}(\texttt{max}(e)) & = & [\,x \mid x \leftarrow \mathcal{Q}\ e]^{max}\\
\mathcal{Q}(e_1\ \texttt{intersect}\ e_2) & = & [\,x \mid x \leftarrow \mathcal{Q}\ e_1, [\,x = y \mid y \leftarrow \mathcal{Q}\ e_2]^{exists}\ ]^{set}\\
\mathcal{Q}(\texttt{flatten}(e)) & = & [\,y \mid x \leftarrow \mathcal{Q}\ e, y \leftarrow x]^{\tau} \qquad (\tau \in \{set, bag, list\})\\
\mathcal{Q}(\texttt{listtoset}(e)) & = & [\,x \mid x \leftarrow \mathcal{Q}\ e]^{set}
\end{array}$$

Figure 2. Translation of some OQL clauses to monad comprehensions.

$\mathcal{Q}$ emits a nested comprehension as the translation result for this query (the comparison $d\,\texttt{.no = }d'\texttt{.no}$ serves to group the employees of one department):

$$[[e\texttt{.sal} \mid e \leftarrow emp, d' \leftarrow dept, \\ e\texttt{.dno} = d'\texttt{.no}, d\texttt{.no} = d'\texttt{.no}]^{sum} \mid d \leftarrow dept, d\texttt{.budg} < \texttt{10000}]^{bag} \tag{11}$$

Although monad comprehensions can be considered as mere syntactic sugar, they give us all the benefits of a calculus-based query representation: for every monad calculus expression, a *normal form* is derivable. The calculus normalization implements a general form of subquery nesting. Calculus as well as user-level query language obey the same variable scoping rules which additionally supports the formulation of simple unnesting rules. Quantifiers have a canonical calculus representation. This will greatly help in rewriting predicates containing quantified variables. The next section will elaborate on these issues.

Let us close this section by noting that the composition $\mathcal{MC} \circ \mathcal{Q}$ already provides a basic query compiler for OQL. The demands on the query execution engine are minimal: it is sufficient to implement the algebraic data type constructors as well as structural recursion via `foldr`. As $\mathcal{MC}$ translates monad comprehensions into nested `foldr`s the query engine will use *nested-loop processing* to execute queries. This is admittedly naive and cannot compete with an algebraically optimized plan but nevertheless provides the tools for a prototypical yet complete OQL compiler. The following two sections will make up for this naivety.

3. Deriving combinator queries from comprehensions

Deriving efficient access programs from monad comprehensions is not as obvious as for query algebras. The design of algebra operators as abstractions of the algorithms implemented by the target query engine induces an one-to-many mapping of query operators to algorithms. This is not true for comprehensions. Access programs for such queries execute, in principle, nested loops. This is due to the translation scheme $\mathcal{MC}$ which establishes a canonical one-to-one mapping from comprehensions to potentially deeply nested applications of the combinator `foldr`. Thus we are facing two problems at this point:

1. How can we reduce the nesting level of a monad comprehension expression?
2. There exists no obvious correlation between comprehensions and the more advanced query engine operators (e.g., efficient variants of join like *nestjoin* $\triangle$ (Steenhagen et al., 1994; Steenhagen, 1995), *semijoin* $\triangleright$, or *antijoin* $\overline{\triangleright}$).

The monad comprehension calculus provides hooks for optimizations to tackle both difficulties, fortunately. Section 3.1 introduces a comprehension-level rewriting that accomplishes a very general form of query unnesting. Our type-based language will also be geared towards a more operation-based representation in Section 3.2.

$$
\begin{array}{rcll}
[\,e \mid qs, x \leftarrow [\,]^{\sigma}, qs'\,]^{\tau} & \rightarrow & [\,]^{\tau} & \text{(12a)} \\
[\,e \mid qs, x \leftarrow y \overset{\sigma}{:} ys, qs'\,]^{\tau} & \rightarrow & [\,e[y/x] \mid qs, qs'[y/x]\,]^{\tau} \overset{\tau}{+\!\!+} & \text{(12b)} \\
 & & [\,e \mid qs, x \leftarrow ys, qs'\,]^{\tau} & \\
[\,e \mid qs, x \leftarrow [\,e' \mid qs''\,]^{\sigma}, qs'\,]^{\tau} & \rightarrow & [\,e[e'/x] \mid qs, qs'', qs'[e'/x]\,]^{\tau} & \text{(12c)} \\
[\,e \mid qs, [\,p \mid qs''\,]^{exists}, qs'\,]^{\tau} & \rightarrow & [\,e \mid qs, qs'', p, qs'\,]^{\tau} & \text{(12d)} \\
 & & (\tau \text{ left-idempotent}) &
\end{array}
$$

Figure 3. Monad comprehension normalization algorithm.

3.1. Comprehension unnesting

A comprehension calculus expression can be put into a *normal form* by the repeated application of a small and confluent set of rewriting rules. As a side-effect, the normalization implements a number of query unnesting strategies that have been proposed in the literature.

The normalization rules are given in figure 3. Variables qs, qs', qs'' denote possibly empty sequences of qualifiers, while σ, τ are monadic types chosen to ensure the well-definedness (see Section 2.2) of the depicted comprehensions. $e[y/x]$ denotes e with all free occurences of x replaced by y.

The rules are to be applied in the direction of the arrows. Equivalence is preserved which can be verified by, e.g., applying $\mathcal{MC}$ to the left- and right-hand sides followed by proofs based on the structural recursion scheme realized by `foldr`. Once the validity of the rules has been established, however, it is much more convenient to reason at the comprehension calculus level. Rules (12c) and (12d) remove one level of nesting in the qualifier list of a comprehension. After rule application, the qualifiers (generators and filters) in qs'' appear at the top-level which opens the possibility for the introduction of joins. Consider the nested comprehension

$$[\,f\ x\ x' \mid x \leftarrow xs, x' \leftarrow [\,y \mid y \leftarrow ys, p\ y\,]^{list}, q\ x\ x'\,]^{set}$$

which, by application of Rule (12c), is normalized into the equivalent expression

$$[\,f\ x\ y \mid x \leftarrow xs, y \leftarrow ys, p\ y, q\ x\ y\,]^{set}$$

Note that the normalized comprehension is equivalent to the join $xs \bowtie_p ys$ with $p = \lambda x\ y. p\ y\ \texttt{\&\&}\ q\ x\ y$. Based on the *qualifier exchange rule* $[\,e \mid qs, q_1, q_2, qs'\,]^{\tau} = [\,e \mid qs, q_2, q_1, qs'\,]^{\tau}$ (given that variables bound in q_1 do not occur free in q_2), which constitutes the calculus analogy of selection push-down and join-order transformations, we could further optimize the resulting normal form. Interestingly, Rule (12c) implements a considerable generalization of Kim's *type J* unnesting for nested SQL queries (Kim, 1982) in which q was fixed to be the equality test.

Furthermore, Kim's *type N* optimization can be understood as an instance of Rule (12d). The nesting in the `where`-clause of the following SQL query is readily removed by comprehension normalization:

```
select distinct f x
           from xs as x
          where g x in (select h y
                         from ys as y
                        where p y)
```

Application of $\mathcal{Q}$ (figure 2) to the above query gives a comprehension which matches the left-hand side of Rule (12d). Note how $\mathcal{Q}$ translates the element test into an existential quantification:

$$\begin{aligned} & [\,f\ x \mid x \leftarrow xs,\ [g\ x = h\ y \mid y \leftarrow ys,\ p\ y]^{exists}\,]^{set} \\ \underset{(12d)}{\rightarrow}\ & [\,f\ x \mid x \leftarrow xs,\ y \leftarrow ys,\ p\ y,\ g\ x = h\ y]^{set} \end{aligned}$$

The right-hand side is now equivalent to a join query and does not repeatedly compute the element test as the original version did:

```
select distinct f x
           from xs as x , ys as y
          where p y and g x = h y
```

Unnesting provides an important preprocessing step for the next optimization phase which we will discuss in the following section: normalized (thus unnested) comprehensions are especially amenable to be transformed into different sorts of joins which, in turn, are efficiently supported by the query engine. In addition, normalized comprehensions produce fewer intermediate results than their non-normalized equivalents. This is an issue that will be examined more closely in Section 4 on the generation of *stream-based* access programs.

3.2. Combinator patterns

Access programs for typical database query engines are expressions over an algebra of *physical operators*. For our purposes, a *logical algebra* provides the appropriate level of abstraction of these physical operators. A logical algebra operator hides the implementation details of its (often many) physical counterparts and reduces them to their abstract logical properties (e.g., commutativity). Unlike for comprehensions, this close correlation gives hints on how a logical operator may be efficiently implemented by an equivalent access program.

Algebraic optimization reorders operators, introduces new ones, or collapses a composition of operators into one. Because we want to algebraically optimize queries, we will represent logical operators as *combinators*, i.e., *closed* expressions of our language. Combinator compositions are easily rearranged because there are no variable interdependencies (unlike in the comprehension calculus) between combinators. It can be beneficial, however, to look into interdependencies between combinators and to uncover their inner control structure when algebraic optimization has done its job. This can help to establish an efficient pipelined execution of combinator queries. Section 4 investigates this point more closely.

In what follows, our two major aims are to map comprehensions into a combinator representation and to show how this mapping process and algebraic optimization can be interleaved with comprehension calculus rewriting. Query transformations may thus be formulated using the representation (calculus or combinator algebra) in which they are most easily expressed (Grust et al., 1997).

We will define the combinators by putting them down to equivalent monad comprehensions. This ensures a semantically clean interaction of calculus and combinator algebra. The combinator definitions provide *patterns* which, once identified, will trigger the replacement of the matched calculus subexpression with the appropriate combinator. The replacement process proceeds until a pure combinator representation has been derived. We are guaranteed to succeed because any calculus expression at least has its canonical `foldr` representation (which we will use as a last resort). Note that, once identified and replaced by an equivalent combinator, the query engine is no longer forced to execute sequential scans and nested loops in order to evaluate a monad comprehension expression. Internally, the combinators may and should be implemented with, e.g., the help of indices. This does not affect our discussion as long as the combinators *act like* their defining monad comprehension when viewed from the outside. Among other equivalences, the qualifier exchange rule provides the means to reorder filters and joins so that query evaluation may benefit from the presence of indices.

Figure 4 lists the monad comprehension equivalents for the algebraic combinators that we will refer to in the sequel. Many more can be defined similarly. Because we will encounter *curried* (i.e., partial) applications of the combinators we prefer the prefix form instead of the "classical" infix operator symbols shown in parentheses.

Aside from `nestjoin` the combinators are pretty standard and should explain themselves. Operator `nestjoin` combines join and grouping: for each object *x* in *s* a group of objects *y* of *t* w.r.t. predicate *p* is determined. Variants of `nestjoin` have already been proven to be especially useful in query processors for complex object models (Cluet and Moerkotte, 1993; Steenhagen, 1995; Rich et al., 1993).

It is actually only a small step from combinator definitions to the associated translation rules they induce, so that we only list few of them in figure 5. Successive applications of these rewriting rules will translate a comprehension expression into an equivalent combinator

$$
\begin{array}{lcll}
\mathtt{map}^{\tau}\ f\ s & = & [\, f\ x \mid x \leftarrow s]^{\tau} & (\pi) \\
\mathtt{filter}^{\tau}\ p\ s & = & [\, x \mid x \leftarrow s, p\ x]^{\tau} & (\sigma) \\
\mathtt{cross}^{\tau\sigma}\ s\ t & = & [\, (x,y) \mid x \leftarrow s, y \leftarrow t]^{\tau} & (\times) \\
\mathtt{join}^{\tau\sigma}\ p\ f\ s\ t & = & [\, f\ x\ y \mid x \leftarrow s, y \leftarrow t, p\ x\ y]^{\tau} & (\bowtie) \\
\mathtt{semijoin}^{\tau\sigma}\ p\ s\ t & = & [\, x \mid x \leftarrow s, [\, p\ x\ y \mid y \leftarrow t]^{exists}\ ]^{\tau} & (\triangleright) \\
\mathtt{antijoin}^{\tau\sigma}\ p\ s\ t & = & [\, x \mid x \leftarrow s, [\, \neg p\ x\ y \mid y \leftarrow t]^{all}\ ]^{\tau} & (\overline{\triangleright}) \\
\mathtt{nestjoin}^{\tau\sigma}\ p\ f\ s\ t & = & [\, [\, f\ x\ y \mid y \leftarrow t, p\ x\ y]^{\sigma} \mid x \leftarrow s]^{\tau} & (\triangle) \\
agg^{\tau}\ f\ s & = & [\, f\ x \mid x \leftarrow s]^{agg} & (agg \in \{\mathtt{max}, \mathtt{min}, \mathtt{exists}, \mathtt{all}, \mathtt{sum}\})
\end{array}
$$

Figure 4. Monad comprehension-based definitions of algebraic combinators.

$$
\begin{array}{rcll}
\mathtt{[}e \mid qs, x \leftarrow xs :: \sigma, p\ x, qs'\mathtt{]}^{\tau} & \rightarrow & \mathtt{[}e \mid qs, x \leftarrow \mathtt{filter}^{\sigma} p\ xs, qs'\mathtt{]}^{\tau} & \text{(13a)} \\
\mathtt{[}f\ e \mid qs\mathtt{]}^{\tau} & \rightarrow & \mathtt{map}^{\tau}\ f\ \mathtt{[}e \mid qs\mathtt{]}^{\tau} & \text{(13b)} \\
\mathtt{[}e \mid qs\mathtt{]}^{agg} & \rightarrow & agg^{\tau}\ \mathtt{id}\ \mathtt{[}e \mid qs\mathtt{]}^{\tau} & \text{(13c)} \\
\mathtt{[}e \mid qs, x \leftarrow xs :: \sigma, & \rightarrow & \mathtt{[}e \mid qs, x \leftarrow \mathtt{semijoin}^{\sigma\sigma'} p\ xs\ ys, qs'\mathtt{]}^{\tau} & \text{(13d)} \\
\quad \mathtt{[}p \mid y \leftarrow ys :: \sigma'\mathtt{]}^{exists}\ , qs'\mathtt{]}^{\tau} & & & \\
\mathtt{[}e \mid qs, x \leftarrow xs :: \sigma, & \rightarrow & \mathtt{[}e[\mathtt{fst}\ v/x][\mathtt{snd}\ v/y] \mid qs, & \text{(13e)} \\
y \leftarrow ys :: \sigma', p\ x\ y, qs'\mathtt{]}^{\tau} & & \quad v \leftarrow \mathtt{join}^{\sigma\sigma'} p\ \mathtt{(}\lambda x\ y.\mathtt{(}x, y\mathtt{))}\ xs\ ys, & \\
 & & \quad qs'[\mathtt{fst}\ v/x][\mathtt{snd}\ v/y]\mathtt{]}^{\tau} &
\end{array}
$$

Figure 5. Combinator pattern matching.

program. In figure 5, some collection variables have been annotated with their respective monadic types to help in deducing the result type after rewriting (the types are indeed inferable). The replacements in Rule (13e) realize deconstruction of the tuple-valued result produced by the `join` (let $\mathtt{fst}\ (x, y) = x$ and $\mathtt{snd}\ (x, y) = y$). The introduction of pattern matching capabilities for comprehension generators, which we have not done for simplicity reasons, make these replacements obsolete.

It is instructive to trace the transformation of the OQL query (10) and its monad comprehension equivalent (11). By interleaving the translation—driven by the rules of figure 5—with rewriting steps on the calculus level, we obtain an efficient combinator expression in only five rewriting steps (subexpressions that trigger the next rewriting step have been marked grey). The resulting combinator expression exploits `nestjoin` to implement the join as well as the grouping. The groups computed by the `nestjoin` are subsequently aggregated by `map sum` employing the *higher-order* nature of the combinator algebra. We believe, in correspondence with (Suciu and Wong, 1995), that operators of higher order are crucial for the efficient compilation of compositional query languages like OQL (OQL's `select`-clause closely corresponds to the higher-order `map` combinator).

$$
\begin{array}{cl}
 & \mathtt{[[}e\mathtt{.sal} \mid e \leftarrow emp :: bag, d' \leftarrow dept :: bag, e\mathtt{.dno} = d'\mathtt{.no}, d\mathtt{.no} = d'\mathtt{.no]}^{sum} \mid \\
 & \qquad d \leftarrow dept :: bag, d\mathtt{.budg} < \mathtt{10000]}^{bag} \\
\underset{\mathtt{(filter)}}{=} & \mathtt{[[}e\mathtt{.sal} \mid e \leftarrow emp, d' \leftarrow dept, e\mathtt{.dno} = d'\mathtt{.no}, d\mathtt{.no} = d'\mathtt{.no]}^{sum} \mid \\
 & \qquad d \leftarrow \mathtt{filter}^{bag}\mathtt{(}\lambda b.b\mathtt{.budg} < \mathtt{10000)}\ dept\mathtt{]}^{bag} \\
\underset{(calc)}{=} & \mathtt{[[}e\mathtt{.sal} \mid e \leftarrow emp, e\mathtt{.dno} = d\mathtt{.no]}^{sum} \mid \\
 & \qquad d \leftarrow \mathtt{filter}^{bag}\mathtt{(}\lambda b.b\mathtt{.budg} < \mathtt{10000)}\ dept\mathtt{]}^{bag} \\
\underset{(agg)}{=} & \mathtt{[sum}^{bag}\mathtt{id\ [}e\mathtt{.sal} \mid e \leftarrow emp, e\mathtt{.dno} = d\mathtt{.no]}^{bag} \mid \\
 & \qquad\qquad d \leftarrow \mathtt{filter}^{bag}\mathtt{(}\lambda b.b\mathtt{.budg} < \mathtt{10000)}\ dept\mathtt{]}^{bag} \\
\underset{\mathtt{(map)}}{=} & \mathtt{map}^{bag}\mathtt{(sum}^{bag}\mathtt{id)\ [[}e\mathtt{.sal} \mid e \leftarrow emp, e\mathtt{.dno} = d\mathtt{.no]}^{bag} \mid \\
 & \qquad\qquad\qquad d \leftarrow \mathtt{filter}^{bag}\mathtt{(}\lambda b.b\mathtt{.budg} < \mathtt{10000)}\ dept\mathtt{]}^{bag} \\
\underset{\mathtt{(nestjoin)}}{=} & \mathtt{map}^{bag}\mathtt{(sum}^{bag}\mathtt{id)(nestjoin}^{bag\,bag}\mathtt{(}\lambda e\ d.e\mathtt{.dno} = d\mathtt{.no)\ (}\lambda e\ d.e\mathtt{.sal)} \\
 & \qquad\qquad \mathtt{(filter}^{bag}\mathtt{(}\lambda b.b\mathtt{.budg} < \mathtt{10000)}\ dept\mathtt{)}\ emp\mathtt{)}
\end{array}
$$

The beneficial interference of calculus- and combinator-level rewriting will become even more apparent during the study of the following two translation examples.

Example. The `where`-clause of the OQL query below involves two nested existential quantifiers. Note that both quantifier formulas are not closed (x appears free in predicates p and q) which makes the translation into an efficient combinator equivalent a somewhat hard task:

$$\begin{array}{l} \texttt{select distinct}\ x \\ \qquad\qquad \texttt{from}\ xs\ \texttt{as}\ x \\ \qquad\qquad \texttt{where exists}\ y\ \texttt{in}\ ys\texttt{: exists}\ z\ \texttt{in}\ zs\texttt{:}\ q\ x\ z\ \texttt{and}\ p\ y\ z \end{array} \tag{14}$$

Due to $q\ x\ z\ \texttt{\&\&}\ p\ y\ z$ being dependent on the free variable x we are stuck after one application of the existential unnesting Rule (12d). This forces us to "bundle" xs and ys by a rather inefficient `cross` product (a program transformation technique commonly referred to as *tupling*) before we can apply the `semijoin` combinator to implement the remaining existential quantifier. The tupling of xs and ys additionally causes the introduction of the outer `map` to extract the first component of the tuples generated by `cross`:

$$\begin{array}{cl} \multicolumn{2}{l}{[x \mid x \leftarrow xs :: \sigma,\ [[q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid z \leftarrow zs :: \sigma'']^{exists} \mid y \leftarrow ys :: \sigma']^{exists}]^{set}} \\ \underset{(12d)}{=} & [x \mid x \leftarrow xs, y \leftarrow ys,\ [q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid z \leftarrow zs]^{exists}]^{set} \\ \underset{(\texttt{cross})}{=} & [\texttt{fst}\ e \mid e \leftarrow \texttt{cross}^{\sigma\sigma'}\ xs\ ys,\ [q\,(\texttt{fst}\ e)z\ \texttt{\&\&}\ p\,(\texttt{snd}\ e)z \mid z \leftarrow zs]^{exists}]^{set} \\ \underset{(\texttt{semijoin})}{=} & [\texttt{fst}\ e \mid e \leftarrow \texttt{semijoin}^{\sigma\sigma''}\,(\lambda e\ z.q\,(\texttt{fst}\ e)z\ \texttt{\&\&}\ p\,(\texttt{snd}\ e)z) \\ & \qquad\qquad\qquad\qquad (\texttt{cross}^{\sigma\sigma'}\ xs\ ys)\ zs]^{set} \\ \underset{(\texttt{map})}{=} & \texttt{map}^{set}\texttt{fst}\ (\texttt{semijoin}^{\sigma\sigma''}\,(\lambda e\ z.q\,(\texttt{fst}\ e)z\ \texttt{\&\&}\ p\,(\texttt{snd}\ e)z) \\ & \qquad\qquad\qquad\qquad (\texttt{cross}^{\sigma\sigma'}\ xs\ ys)\ zs) \end{array}$$

However, the variable interdependency problem in this query is easily resolved by two rewritings which are naturally expressed in the calculus representation. The first performs the exchange of the generator domains in a nested comprehension over any commutative aggregate τ: $[[p \mid z \leftarrow zs]^\tau \mid y \leftarrow ys]^\tau = [[p \mid y \leftarrow ys]^\tau \mid z \leftarrow zs]^\tau$. In our second attempt to translate query (14), we specialize this rule by fixing τ to be *exists* and obtain a *quantifier exchange* rule this way. The second transformation, *descoping*, moves a predicate q out of the scope of a quantifier if the quantifier's range variable does not occur free in q: $[q \overset{\sigma}{:} p \mid x \leftarrow xs]^\tau = q \overset{\sigma}{:} [p \mid x \leftarrow xs]^\tau$, *for* $\tau = exists$, $\sigma = all$ or *vice-versa*. The optimizer can infer an efficient `semijoin`-based plan for query (14) in the presence of these additional rules *and* the combinator patterns of figure 5:

$$\begin{array}{cl} \multicolumn{2}{l}{[x \mid x \leftarrow xs\ \ :: \sigma,\ [[q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid z \leftarrow zs :: \sigma'']^{exists} \mid y \leftarrow ys :: \sigma']^{exists}]^{set}} \\ \underset{\text{(quant. ex.)}}{=} & [x \mid x \leftarrow xs,\ [[q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid y \leftarrow ys]^{exists} \mid z \leftarrow zs]^{exists}]^{set} \end{array}$$

$$\underset{\text{(descope)}}{=} \quad [x \mid x \leftarrow xs,\ [q\ x\ z\ \&\&\ [p\ y\ z \mid y \leftarrow ys]^{exists} \mid z \leftarrow zs]^{exists}]^{set}$$

$$\underset{\text{(descope)}}{=} \quad [x \mid x \leftarrow xs,\ [q\ x\ z \mid z \leftarrow zs,\ [p\ y\ z \mid y \leftarrow ys]^{exists}]^{exists}]^{set}$$

$$\underset{(2\,\times\,\texttt{semijoin})}{=} \quad \texttt{semijoin}^{set\sigma''} q\ xs\ (\texttt{semijoin}^{\sigma''\sigma'}\ p\ zs\ ys)$$

To conclude this section on query translation, let us briefly review the transformation of a class of 16 queries discussed in Claussen et al. (1997). All of these are treated alike in our representation. Here, we will concentrate on the most interesting class.

Example. Queries of the generic form shown below are considered hard to translate for purely algebraic optimizers mainly because there exists no canonical translation "recipe" for universal quantification in the `where` clause:

```
select x
  from xs as x
 where for all y in (select y
                       from ys as y
                      where p y) : q x y
```

Possible algebraic equivalents involve *set difference*, *relational division*, or a combination of grouping and counting to implement the universal quantifier. The derivation of these algebraic forms is tedious, however, and the resulting expressions are judged to be too complex to be useful during subsequent rewriting phases (Nakano, 1990; Steenhagen, 1995). The case is even more complex if the nested subquery is not closed (e.g., if the quantifier predicate $p\ y$ is replaced by $p\ x\ y$). This renders the use of *division* impossible.

The universal quantifier is no obstacle in our approach though. By means of $\mathcal{Q}$, the quantifier is translated into a comprehension over monad *all*. If the subquery is closed we can fire Rule (13a) which introduces a `filter` to implement the selection w.r.t. p. This case is depicted in the first chain of transformations below. Otherwise (i.e., if the subquery is not closed) we apply a variant of the descoping rule explained above in the reverse direction: predicate p is moved *into* the scope of the universal quantifier. The comprehensions now match the typical `antijoin` pattern ($(\overline{\triangleright})$ in figure 4) in both cases. Application of the `antijoin` rule already completes both transformations.
Subquery closed:

$$[x \mid x \leftarrow\ xs :: bag,\ [q\ x\ y \mid y \leftarrow ys :: bag,\ p\ y]^{all}]^{bag}$$

$$\underset{(\texttt{filter})}{=} \quad [x \mid x \leftarrow xs,\ [q\ x\ y \mid y \leftarrow \texttt{filter}^{bag} p\ ys]^{all}]^{bag}$$

$$\underset{(\texttt{antijoin})}{=} \quad \texttt{antijoin}^{bag\,bag}(\neg q)\ xs\ (\texttt{filter}^{bag} p\ ys)$$

Subquery not closed (references variable x of enclosing scope):

$$\begin{aligned}
&[x \mid x \leftarrow\ xs :: bag,\ [q\ x\ y \mid y \leftarrow ys :: bag,\ p\ x\ y]^{all}]^{bag} \\
\underset{\text{(inscope)}}{=}\quad &[x \mid x \leftarrow xs,\ [q\ x\ y \mid\mid \neg p\ x\ y \mid y \leftarrow ys]^{all}]^{bag} \\
\underset{\texttt{(antijoin)}}{=}\quad &\texttt{antijoin}^{bag\,bag}\,(\lambda x\ y.\neg q\ x\ y\ \&\&\ p\ x\ y)\ xs\ ys
\end{aligned}$$

The derived `antijoin` expressions are identical to those that have been identified as the most efficient alternatives by Claussen et al. (1997).

Note that the transformations were purely driven by pattern matching as well as basic predicate rewriting and did not involve some sort of “eureka” step.

We have found the combination of monad calculus-based and combinator (or algebraic) rewriting to cover, extend, and generalize most of the proposed algebraic optimization frameworks. The flexibility of the representation gives us the option to at least *simulate* related ideas. Additionally, monad comprehensions and combinators are concepts that are susceptible to program transformation techniques (like *deforestation*) developed by the functional programming community. This provides another valuable source of knowledge that can and should be exploited in the query optimization context.

4. Deforestation of combinator queries

The widely accepted approach of designing a query algebra as a set of combinators facilitates algebraic query optimization. Combinators may be orthogonally combined and easily rearranged due to the absence of interdependencies between operators. From the viewpoint of the *query execution engine*, however, combinator algebras come with an inherent cost: during query execution, temporary results need to be communicated between operators because these are individually designed to consume their input as a whole, and subsequently produce an intermediate result. The cost of the I/O of temporary results may very well dominate the overall query execution cost. These observations led to the development of query processors that try to operate in a *streaming* (or *pipelined*) mode. Query execution benefits from streaming since objects are addressed and loaded from the persistent storage only once. Execution is in-memory and no intermediate writes to the secondary storage and subsequent reads for materialized intermediate results occur.

Approaches to the derivation of streaming programs for combinator queries have been predominantly devised on the implementation-level only: Graefe (1990) proposed to reimplement operators in a streaming style. Combinators consume their input on demand (lazily) and element-wise by interacting via a simple *end-of-stream*? and *next* call interface. The work of Lieuwen and DeWitt (1992) developed a specific set of source-to-source transformations to optimize the flow of values through access programs written in an imperative implementation language.

The key to the idea we will develop here is the observation that combinator style queries coincide very closely with *listful programs*, a term coined by the functional programming community. A listful program expresses a complex list manipulation by composition of generic combinators, each generating an intermediate result list. Programs of this style may be *deforested*[7] (Wadler, 1990b), i.e., transformed so that they allocate no intermediate data structures. Deforestation is a specific instance of the general *unfold-fold* program transformation strategy (Burstall and Darlington, 1977): combinators are replaced by their defining expressions (*unfolded*) with the aim of fusing them with the definitions of neighbouring combinators. The strategy then tries to *fold* the fused program back into combinator form, a step that involves complex pattern matching and, for some cases, can be done only semiautomatically.

The type-based foundation and uniform representation of our intermediate language allows us to take a different route. As we have remarked in Section 2.1 already, structural recursion ($\mathtt{foldr}^{\tau}$) provides the principal way to express functions over values of an initial algebraic data type τ. All combinators introduced in the last section may indeed be re-expressed by `foldr` directly. Figure 6 summarizes the `foldr`-based definitions of some combinators (some are omitted for the sake of brevity).

The transformation of programs that are solely built from basic combinators like `foldr` is a widely investigated area of functional programming (Bird, 1987; Grant, 1989; Meijer et al., 1991). Most importantly for our purposes, however, *cheap deforestation* (or `foldr-build` deforestation) is possible for `foldr`-based programs (Gill et al., 1993; Launchbury and Sheard, 1995; Gill, 1996). Cheap deforestation is an one-step transformation that does not involve a *fold* step. It relies on the observation that a list built from the constructors `(:)` and `[]` which is subsequently consumed by a `foldr` may actually be reduced *during* its construction. This matches exactly the scenario we face during the processing of combinator queries. The derived `foldr`-based queries may then be used as a template to derive an actual typically imperative storage access program (which is a manageable task due to the simple *linear* recursion scheme represented by `foldr`).

Cheap deforestation has been developed as a technique for the optimization of *listful programs*. However, the principle nicely adapts to the monadic type system (Grust and Scholl, 1998).

$$
\begin{array}{lcl}
\mathtt{map}^{\tau}\ f\ s & = & \mathtt{foldr}^{\tau}(\lambda x\ xs.f\ x \overset{\tau}{:} xs)\ \mathtt{[]}^{\tau}\ s \\
\mathtt{filter}^{\tau}\ p\ s & = & \mathtt{foldr}^{\tau}(\lambda x\ xs.\mathtt{if}\ p\ x\ \mathtt{then}\ x \overset{\tau}{:} xs\ \mathtt{else}\ xs)\ \mathtt{[]}^{\tau}\ s \\
\mathtt{join}^{\tau\sigma}\ p\ f\ s\ t & = & \mathtt{foldr}^{\tau}(\lambda x\ xs.\mathtt{foldr}^{\sigma}(\lambda y\ ys. \\
 & & \ \mathtt{if}\ p\ x\ y\ \mathtt{then}\ f\ x\ y \overset{\tau}{:} ys\ \mathtt{else}\ ys)\ xs\ t)\ \mathtt{[]}^{\tau}\ s \\
\mathtt{semijoin}^{\tau\sigma}\ p\ s\ t & = & \mathtt{foldr}^{\tau}(\lambda x\ xs. \\
 & & \ \mathtt{if}\ \mathtt{exists}^{\sigma}\ (p\ x)\ t\ \mathtt{then}\ x \overset{\tau}{:} xs\ \mathtt{else}\ xs)\ \mathtt{[]}^{\tau}\ s \\
\mathtt{nestjoin}^{\tau\sigma}\ p\ f\ s\ t & = & \mathtt{foldr}^{\tau}(\lambda x\ xs. \\
 & & \ (\mathtt{foldr}^{\sigma}(\lambda y\ ys. \\
 & & \ \ \mathtt{if}\ p\ x\ y\ \mathtt{then}\ f\ x\ y \overset{\sigma}{:} ys\ \mathtt{else}\ ys)\ \mathtt{[]}^{\sigma}\ t) \overset{\tau}{:} xs)\ \mathtt{[]}^{\tau}\ s \\
agg^{\tau}\ f\ s & = & \mathtt{foldr}^{\tau}(\lambda x\ xs.f\ x \overset{agg}{:} xs)\ \ \mathtt{[]}^{agg}\ s
\end{array}
$$

Figure 6. Some combinators expressed via structural recursion.

4.1. *Cheap deforestation*

How would a streaming program for the combinator query $\mathtt{map}^{\tau}\, f\ (\mathtt{filter}^{\tau}\, p\ s)$ look like? Instead of allocating the temporary result of the $\mathtt{filter}^{\tau}\, p$ pass over s, a streaming plan would scan s, and *immediately* apply f but only to those objects x that satisfy p (and otherwise just drop x). The corresponding program does not allocate intermediate results and would read

$$\mathtt{foldr}^{\tau}\,(\lambda x\ xs.\mathtt{if}\ p\ x\ \mathtt{then}\ f\ x \overset{\tau}{:} xs\ \mathtt{else}\ xs)\ \mathtt{[]}^{\tau}\ s \tag{15}$$

Having detected this particular chance for streaming, we could supply the plan generator with a rewrite rule to realize this specific transformation. However, how are we supposed to deduce a streaming program from a composition of arbitrary combinators?

This is where the regular recursion pattern expressed by $\mathtt{foldr}^{\tau}$ comes into play. Recall from (4) that $\mathtt{foldr}^{\tau}\,(\oplus)\ z$ traverses its argument of type $\tau\ \alpha$ and replaces constructor $(\overset{\tau}{:})$ by $(\oplus)$ and $\mathtt{[]}^{\tau}$ by z as it goes. If this replacement could be done at *plan generation time*, i.e., *not* during query execution but *statically*, we could get rid of that $\mathtt{foldr}^{\tau}$ application at all. Cheap deforestation does exactly this for $\tau = \mathit{list}$ (Gill et al., 1993).

To implement this idea we need to gain a handle of all constructor occurences in a core language expression, g say. We can achieve this by compile-time *β-abstraction* of g w.r.t. the constructors it uses: $\lambda c\ n.g[c/\overset{\tau}{:}][n/\mathtt{[]}^{\tau}]$. The result of applying $\mathtt{foldr}^{\tau}\,(\oplus)\ z$ to g may now readily computed at plan generation time:

$$\mathtt{foldr}^{\tau}\,(\oplus)\ z\ g \underset{(\star)}{=} (\lambda c\ n.g[c/\overset{\tau}{:}][n/\mathtt{[]}^{\tau}])\ (\oplus)\ z \underset{\beta}{=} g[\oplus/\overset{\tau}{:}][z/\mathtt{[]}^{\tau}] \tag{16}$$

Note that the rightmost expression is a streaming program for the composition $\mathtt{foldr}^{\tau}\ (\oplus)\ z\ g$: instead of constructing a temporary result using $(\overset{\tau}{:})$ and $\mathtt{[]}^{\tau}$, g uses $\oplus$ and z to reduce its input during consumption.

The correctness of the deforestation step at $(\star)$ clearly depends on the prerequisite that g exclusively uses $(\overset{\tau}{:})$ and $\mathtt{[]}^{\tau}$ to produce its result since we need to replace *all* constructors to implement streaming correctly. This condition is met by all combinators in figure 6. Note that we may safely replace $\oplus$ for $(\overset{\tau}{:})$ in g without sacrificing well-definedness since $\oplus$ is left-idempotent/left-commutative whenever $(\overset{\tau}{:})$ is (provided that the left-hand expression is well-defined).

Before we turn to some examples of combinator deforestation, let us adopt the `build` notation introduced in (Gill et al., 1993). If we define $\mathtt{build}^{\tau} g = g\ (\overset{\tau}{:})\ \mathtt{[]}^{\tau}$ (here, the type abstraction allows us to supply correctly typed constructor parameters to g), we can render $(\star)$ more concisely as

$$\mathtt{foldr}^{\tau}\,(\oplus)\ z\ (\mathtt{build}^{\tau} g') = g'\ (\oplus)\ z \tag{17}$$

(where g' has been derived by β-abstraction of g as described above). This `foldr`-`build` cancellation is the only transformation cheap deforestation relies on. No *fold* step is required.

The application of cheap deforestation to the streaming plan problem is immediate: we use β-abstraction to factor constructor occurences out of the `foldr`-based combinator

$$
\begin{aligned}
\texttt{map}^{\tau}\ f\ s &= \texttt{build}^{\tau}\,(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.f\ x\ `c`\ xs)\ n\ s)\\
\texttt{filter}^{\tau}\ p\ s &= \texttt{build}^{\tau}\,(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.\\
&\quad \texttt{if}\ p\ x\ \texttt{then}\ x\ `c`\ xs\ \texttt{else}\ xs)\ n\ s)\\
\texttt{join}^{\tau\sigma}\ p\ f\ s\ t &= \texttt{build}^{\tau}\,(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.\texttt{foldr}^{\sigma}\,(\lambda y\ ys.\\
&\quad \texttt{if}\ p\ x\ y\ \texttt{then}\ f\ x\ y\ `c`\ ys\ \texttt{else}\ ys)\ xs\ t)\ n\ s)\\
\texttt{semijoin}^{\tau\sigma}\ p\ s\ t &= \texttt{build}^{\tau}\,(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.\\
&\quad \texttt{if}\ \texttt{exists}^{\sigma}\ (p\ x)\ t\ \texttt{then}\ x\ `c`\ xs\ \texttt{else}\ xs)\ n\ s)\\
\texttt{nestjoin}^{\tau\sigma}\ p\ f\ s\ t &= \texttt{build}^{\tau}\,(\lambda c_1\ n_1.\texttt{foldr}^{\tau}\,(\lambda x\ xs.\\
&\quad (\texttt{build}^{\sigma}\,(\lambda c_2\ n_2.\texttt{foldr}^{\sigma}\,(\lambda y\ ys.\\
&\quad\ \texttt{if}\ p\ x\ y\ \texttt{then}\ f\ x\ y\ `c_2`\ ys\ \texttt{else}\ ys)\ n_2\ t)\ `c_1`\ xs)\ n_1\ s))\\
agg^{\tau}\ f\ s &= \texttt{build}^{agg}(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.f\ x\ `c`\ xs)\ n\ s)
\end{aligned}
$$

Figure 7. `foldr`-`build` forms of some combinators.

definitions (figure 6). We additionally reexpress the combinators by `build` as shown in figure 7 to prepare them for the fusion with adjacent combinators. Note that `nestjoin` produces its result using the constructors of both τ and σ so that `build` has to be applied twice to fully abstract the constructor occurences away. Combinator queries (i.e., compositions of combinators) will thus consist of `foldr`-`build` pairs which we will try to cancel by Rule (17).

Two examples might help to reveal the potentials of query deforestation. The unfolded combinator queries look rather convoluted but their structure—alternating applications of `build` and `foldr`—is quite regular. A `foldr`-`build` pair is marked grey if it triggers the next deforestation step.

Example. Turning back to our initial example query, $\texttt{map}^{\tau}\ f\ (\texttt{filter}^{\tau}\ p\ s)$, deforestation derives a streaming plan as follows. The combinator definitions are unfolded to give

$$
\begin{aligned}
&\texttt{build}^{\tau}\,(\lambda c\ n.\texttt{foldr}^{\tau}\,(\lambda x\ xs.\,f\ x\ `c`\ xs)\ n\\
&\ (\texttt{build}^{\tau}\,(\lambda c_1\ n_1.\texttt{foldr}^{\tau}\,(\lambda y\ ys\ .\ \texttt{if}\ \ p\ y\ \texttt{then}\ y\ `c_1`\ ys\ \texttt{else}\ ys)\ n_1\ s)))
\end{aligned}
$$

The deforestation rule is applied once which produces

$$
\begin{aligned}
&\texttt{build}^{\tau}\,(\lambda c\ n.\\
&\ \ \texttt{foldr}^{\tau}\,(\lambda y\ ys.\texttt{if}\ \ p\ y\ \texttt{then}\ \ f\ y`c`ys\ \texttt{else}\ ys)\ n\ s)
\end{aligned}
$$

Finally, unfolding the outer `build` gives the desired streaming program we have shown before (15)

$$
\texttt{foldr}^{\tau}\,(\lambda y\ ys.\texttt{if}\ \ p\ y\ \texttt{then}\ \ f\ y\ \overset{\tau}{:}\ ys\ \texttt{else}\ ys)\ \ \texttt{[]}^{\tau}\ s
$$

Example. To complete the translation of the "low budget" query (10), let us deforest its combinator equivalent which we have derived in Section 3.2 (some steps have been omitted to save space). Query deforestation comes up with a streaming program that computes the `sum` aggregation during the grouping phase so that the groups need never be allocated. After combinator unfolding we have

```
foldr^bag (λs ss.foldr^bag (+) 0 s :^bag ss) []^bag
 (build^bag (λc n.foldr^bag (λd ds.
  build^bag (λc_1 n_1.foldr^bag (λe es.
   if e.dno = d.no then e.sal ‘c_1‘ es else es) n_1 emp) ‘c‘ ds)
  n (build^bag (λc_2 n_2.
   foldr^bag (λb bs.if b.budg < 10000 then b ‘c_2‘ bs else bs) n_2 dept))))
```

Deforestation of `map` results in the `sum` and `nestjoin` being adjacent:

```
foldr^bag (λd ds.
 foldr^bag (+) 0 (build^bag (λc_1 n_1.
 foldr^bag (λe es.if e.dno = d.no then e.sal ‘c_1‘ es else es) n_1 emp)) :^bag ds)
 []^bag (build^bag (λc_2 n_2.
  foldr^bag (λb bs.if b.budg < 10000 then b ‘c_2‘ bs else bs) n_2 dept))
```

This opens the possibility to finally merge the aggregate into the grouping phase to give the resulting streaming plan in which aggregation, join, and grouping have been fused completely:

```
foldr^bag (λd ds.
 if d.budg < 10000
  then (foldr^bag (λe es.if e.dno = d.no then e.sal + es else es) 0 emp) :^bag ds
  else ds) []^bag dept
```

4.1.1. Deforestation of comprehensions. Since a comprehension over type τ is essentially defined by a mapping to the monadic operations of τ, which in turn are implementable by structural recursion (Section 2.2), a direct deforestation of comprehensions should be possible. List comprehensions indeed deforest well (Gill, 1996). A suitable generalization of this observation to monadic types leads to the modified comprehension translation scheme $\mathcal{MC}^*$ in (18). $\mathcal{MC}^*$ abstracts from the specific monadic type τ using $\texttt{build}^\tau$ and then calls the helper function $\mathcal{C}$ to translate the comprehension into structural recursion over its generator domains.

This interaction of monad comprehensions and deforestation opens up another interesting alternative to query compilation: the composition $\mathit{deforestation} \circ \mathcal{MC}^* \circ \mathcal{Q}$ provides a rather efficient short cut to query language implementation in this functional framework. Again, just like for $\mathcal{MC} \circ \mathcal{Q}$ (Section 2.3), a restricted `foldr`-based query engine would be sufficient to execute the resulting plans.

$$\begin{aligned}
\mathcal{MC}^{*}\ \mathtt{[}e \mid qs\mathtt{]}^{\tau} &= \mathtt{build}^{\tau}\,\mathtt{(}\lambda c\ n.\mathcal{C}\ \mathtt{[}e \mid qs\mathtt{]}\ c\ n\mathtt{)} \\
\mathcal{MC}^{*}\ e &= e \\
\\
\mathcal{C}\ \mathtt{[}e \mid \mathtt{]}\ c\ n &= c\ (\mathcal{MC}^{*}\ e)\ n \\
\mathcal{C}\ \mathtt{[}e \mid p, qs\mathtt{]}\ c\ n &= \mathtt{if}\ \mathcal{MC}^{*}\ p\ \mathtt{then}\ \mathcal{C}\ \mathtt{[}e \mid qs\mathtt{]}\ c\ n\ \mathtt{else}\ n \\
\mathcal{C}\ \mathtt{[}e \mid x \leftarrow q :: \sigma, qs\mathtt{]}\ c\ n &= \mathtt{foldr}^{\sigma}\,\mathtt{(}\lambda x\ xs.\mathcal{C}\ \mathtt{[}e \mid qs\mathtt{]}\ c\ xs\mathtt{)}\ n\ q
\end{aligned} \tag{18}$$

A final note on an implementation aspect is appropriate here. Actual implementations of query engines are predominantly based on list processing. This is mainly due to the apparent cost of *duplicate elimination* inherent in $(\overset{set}{:})$ and actually enables the engine to reason about sorting orders of the processed data streams. These query engines typically employ a separate `dupelim` combinator to enforce *set* semantics if needed. Such systems are naturally represented in our framework as `dupelim` can be expressed in terms of structural recursion as well. The query optimizer then has the option to delay duplicate elimination based on the equivalence

$$\mathtt{build}^{set} g \overset{\rightarrow}{=} \mathtt{dupelim(build}^{list} g\mathtt{)}$$

(read in the direction of the arrow, this allows to “push duplicate elimination out” of the plan and thus enable efficient list-based processing of g). This rule is of particular significance in the streaming program context since duplicate elimination typically involves sorting or memoization which disrupts the fully stream-based execution of queries.

The loop fusion techniques for the derivation of iterative programs from relational queries developed in (Freytag and Goodman, 1986b, 1989) turn out to be specific instances of the deforestation step. At the same time, deforestation appears to be simpler. The implementation of query deforestation merely involves the exhaustive application of a single syntactic transformation. This is in contrast to, e.g., (Poulovassilis and Small, 1997) where extensive sets of rewriting rules are derived from a generic `fold` fusion rule. Such rule sets imply the need for a more or less sophisticated rule application strategy. Additionally, Poulovassilis and Small (1997) accompanies specific rules with rather complex provisos (e.g., strictness or distributivity of functions) that cannot easily be asserted, let alone be checked syntactically (Breazu-Tannen and Subrahmanyam, 1991). Query deforestation is able to deforest the *unsafe programs* (queries in which a nested subquery traverses partial results computed by an outer query) of Fegaras (1993). Instances of this class of programs, elements of which are aggregate queries that compute running sums or list reversal, are not susceptible to the *fold promotion theorem* exploited in the above cited work and thus cannot be fused with adjacent expressions.

5. Conclusions and further work

The consistent comprehension of queries as functional programs has been the major driving force behind the present work and it is our main claim that database query optimization can derive immense benefit from taking this view.

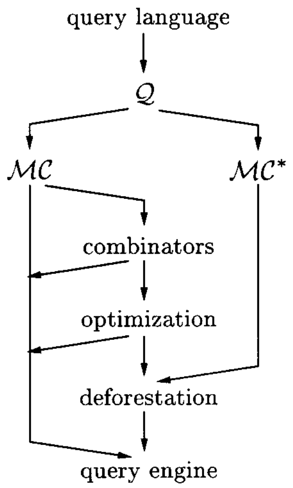

Figure 8. Alternative paths for query compilation.

It turned out that the principle of structural recursion was the most important design choice. On the one hand, it influenced the type-based design of the intermediate representation. Making the constructors of the algebraic data types available in the language gave a finer granularity of abstraction than operation-based languages could provide. The initiality of the underlying algebras additionally ensured completeness of this type-oriented language core. In a world of initial algebras, structural recursion coveniently provides the principal skeleton after which proofs of the correctness of query transformations can be structured.

On the other hand, we have shown structural recursion itself to be able to express the basic idioms of iteration, aggregation, and quantification which are at the heart of database query languages. Two different forms of syntactic sugar, monad comprehensions and combinators, both of which can be put down to the basic recursion combinator `foldr`, established connecting links to the user-level query language and the target query engines.

It is a unique feature of this approach that a single uniform formal framework embraces all stages of the query compilation process (figure 8). Traveling the paths in figure 8 top-down involves a shift of the syntactic viewpoint (from comprehensions to combinators) but the underlying formal basis is never changed (unlike in, e.g., the *logical vs. physical algebra* approaches). Especially the plan generation phase constitutes an area that has mainly been tackled on the implementation level so far. The derivation of streaming programs from comprehension and combinator expressions is only a first step. We are convinced that further plan generation strategies may be understood in terms of this model and thus become subject to formal reasoning, too. Since certain tree-like data types may be grasped by the monadic approach, it is conceivable to model index lookups by structural recursion on the index data structures (e.g., in *b-trees*). Deforestation could then be used to efficiently interleave lookups and query execution. These tree data types could also provide the means for semi-structured data and query processing. This is an area that remains to be explored.

Different levels of sophistication are equally expressible in this framework, because the query translation process itself is completely compositional. The distinct translation phases may be arbitrarily composed to select a specific path in figure 8. Both outer paths—which correspond to the short cuts $\mathcal{MC} \circ \mathcal{Q}$ and *deforestation* $\circ\, \mathcal{MC}^* \circ \mathcal{Q}$—involve syntactic transformations only (and thus give rise to rapidly prototyped implementations of query languages due to the minimal demands on the query engine) while the combinator mapping and optimization in the inner path involves query unnesting and algebraic rewriting. In addition to the techniques discussed here, the extensive body of knowledge of optimization methods should clearly be exploited on this path. This involves encoding new algebraic operators by structural recursion (by analogy with, e.g., `nestjoin`) as well as the representation of advanced query processing strategies like the *bypass* technique (Steinbrunn et al., 1995) which we have found to be expressible by tupling (Grust and Scholl, 1996).

Notes

1. A combinator is an expression of the λ-calculus which contains no occurences of a free variable.
2. The complete Boom hierarchy of types may be understood in terms of likewise defined algebraic data types.
3. This is a consequence of the fact that the term algebra modulo provable equalities of an algebraic data type specification is initial. Since initial algebras are isomorphic, reasoning about constructed terms is essentially the same as reasoning about the elements of the algebra themselves.
4. As in Haskell, we will write $(\oplus)$ for the prefix application of infix operator $\oplus$, and 'c' if a binary function c is used as an infix operator.
5. The primitive operators and the conditional are not special. If, for example, we model the booleans as the algebraic data type $bool = (bool, \mathtt{False}, \mathtt{True})$ and then define its associated $\mathtt{foldr}^{bool}$ operator as

$$\mathtt{foldr}^{bool}\ f\ z\ \mathtt{False} = z$$
$$\mathtt{foldr}^{bool}\ f\ z\ \mathtt{True} = f$$

   we have $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 = \mathtt{foldr}^{bool}\ e_2\ e_3\ e_1$ and $e_1\ \mathtt{||}\ e_2 = \mathtt{foldr}^{bool}\ \mathtt{True}\ e_2\ e_1$. We prefer the familiar notation for these primitives in order to render expressions more readable.
6. Note that, in categorical terms, τ is an endofunctor in the category of sets and total functions whose morphism mapping is given by map^{τ}. As we want to avoid to delve into the categorical background here, you may safely continue to perceive τ as a type constructor.
7. Deforestation removes intermediate data structures, which Wadler collectively refers to as trees. Hence the name *deforestation*.

References

Bird, R.S. (1987). An Introduction to the Theory of Lists. In M. Broy (Ed.), *Logic of Programming and Calculi of Design*. NATO ASI Series. Springer Verlag, vol. 36, pp. 5–42.

Breazu-Tannen, V., Buneman, P., and Wong, L. (1992). Naturally embedded query languages. *Proc. of the Int'l Conference on Database Theory (ICDT)* (pp. 140–154). Berlin, Germany.

Breazu-Tannen, V. and Subrahmanyam, R. (1991). Logical and computational aspects of programming with sets/bags/lists. *Proc. of the 18th Int'l Colloquium on Automata, Languages and Programming* (pp. 60–75). Madrid, Spain.

Buneman, P., Libkin, L., Suciu, D., Tannen, V., and Wong, L. (1994). Comprehension Syntax. *ACM SIGMOD Record*, 23, 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(1), 3–48.

Burstall, R.M. and Darlington, J. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1), 44–67.

Cattell, R.G. and Barry, D.K. (Eds.) (1997). *The Object Database Standard: ODMG 2.0*, San Francisco, CA: Morgan Kaufmann. Release 2.0.

Claussen, J., Kemper, A., Moerkotte, G., and Peithner, K. (1997). Optimizing queries with universal quantification in object-oriented databases. *Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB)*. Athens, Greece.

Cluet, S. and Moerkotte, G. (1993). Nested queries in object bases. *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)*.

Fegaras, L. (1993). Efficient optimization of iterative queries. *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)* (pp. 200–225).

Fegaras, L. (1994). A Uniform Calculus for Collection Types. Technical Report 94-030, Oregon Graduate Institute of Science & Technology.

Fegaras, L. and Maier, D. (1995). Towards an effective calculus for object query languages. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*.

Freytag, J.C. and Goodman, N. (1986a). Rule-based translation of relational queries into iterative programs. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data* (pp. 206–214). Washington, DC, USA.

Freytag, J.C. and Goodman, N. (1986b). Translating aggregate queries into iterative programs. *Proc. of the 12th Int'l Conference on Very Large Data Bases (VLDB)* (pp. 138–146). Kyoto, Japan.

Freytag, J.C. and Goodman, N. (1989). On the Translation of Relational Queries into Iterative Programs. *ACM Transactions on Database Systems*, 14(1), 1–27.

Gill, A.J. (1996). Cheap Deforestation for Non-Strict Functional Languages. Ph.D. dissertation, Department of Computing Science, University of Glasgow.

Gill, A.J., Launchbury, J., and Peyton-Jones, S.L. (1993). A Short Cut to Deforestation. *Functional Programming & Computer Architecture* (pp. 223–232).

Gluche, D., Grust, T., Mainberger, C., and Scholl, M.H. (1997). Incremental updates for materialized OQL views. *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD'97)* (pp. 52–66). Montreux, Switzerland.

Goguen, J.A., Thatcher, J.W., and Wagner, E.G. (1978). An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R.T. Yeh (Ed.), *Current Trends in Programming Methodology*. Prentice Hall, vol. 4, chap. 5, pp. 81–149.

Graefe, G. (1990). Encapsulation of parallelisms in the volcano query processing system. *Proc. of the ACM SIGMOD Int'l Conference on Management of Data* (pp. 102–111). Atlantic City, NJ, USA.

Graefe, G. (1993). Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 73–170.

Grant, M. (1989). Homomorphisms and promotability. *Proc. of the Int'l Conference on Mathematics of Program Construction* (pp. 335–347). Groningen, The Netherlands.

Grust, T., Kröger, J., Gluche, D., Heuer, A., and Scholl, M.H. (1997). Query evaluation in CROQUE—Calculus and algebra coincide. *Proc. of the 15th British National Conference on Databases (BNCOD)* (pp. 84–100). London: Birkbeck College.

Grust, T. and Scholl, M.H. (1996). Translating OQL into Monoid Comprehensions—Stuck with Nested Loops? Technical Report 3/1996, Faculty of Mathematics and Computer Science, Database Research Group, University of Konstanz.

Grust, T. and Scholl, M.H. (1998). Query Deforestation. Technical Report 68/1998, Faculty of Mathematics and Computer Science, Database Research Group, University of Konstanz.

Hudak, P., Peyton Jones, S.L., Wadler, P. et al. (1992). Report on the Programming Language Haskell. *ACM SIGPLAN Notices*, 27(5).

Kemp, G.J.L., Iriarte, J.J., and Gray, P.M.D. (1994). Efficient access to FDM objects stored in a relational database. *Proc. of the 12th British National Conference on Databases (BNCOD)* (pp. 170–186). Guildford, UK.

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

Launchbury, J. and Sheard, T. (1995). Warm fusion: Deriving build-catas from recursive definitions. *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*. La Jolla, CA.

Lieuwen, D.F. and DeWitt, D.J. (1992). A transformation based approach to optimizing loops in database programming languages. *Proc. of the ACM SIGMOD Int'l Conference of Management of Data* (pp. 91–100). San Diego, CA.

Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes, and barbed wire. *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*. Number 523 in Lecture Notes in Computer Science (LNCS) (pp. 124–144). Cambridge, USA: Springer Verlag.

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

Ohori, A. (1990). Representing Object Identity in a Pure Functional Language. In P.C. Kanellakis and S. Abiteboul (Eds.), *Proc. of the 3rd Int'l Conference on Database Theory (ICDT)*. Number 470 in Lecture Notes in Computer Science (LNCS) (pp. 41–55). Paris, France: Springer Verlag.

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

Poulovassilis, A. and King, P. (1990). Extending the functional data model to computational completeness. *Proc. of the 2nd Int'l Conference on Extending Database Technology (EDBT)*, number 416 in Lecture Notes in Computer Science (LNCS) (pp. 75–91). Venice, Italy: Springer Verlag.

Poulovassilis, A. and Small, C. (1996). Algebraic Query Optimisation for Database Programming Languages. *The VLDB Journal*, 5(2), 119–132.

Poulovassilis, A. and Small, C. (1997). Formal foundations for optimising aggregation functions in database programming languages. *Proc. of the 6th Int'l Workshop on Database Programming Languages (DBPL)*. Estes Park, Colorado, USA.

Rich, C., Rosenthal, A., and Scholl, M.H. (1993). Reducing duplicate work in relational join(s): A unified approach. *Proc. of the Int'l Conference on Information Systems and Management of Data (CISMOD)* (pp. 87–102). Delhi, India.

Schek, H.-J. and Scholl, M.H. (1986). The relational model with relation-valued attributes. *Information Systems*, 11(2), 137–147.

Steenhagen, H.J. (1995). Optimization of Object Query Languages. Ph.D. dissertation, Department of Computer Science, University of Twente.

Steenhagen, H.J., Apers, P.M., Blanken, H.M., and de By, R.A. (1994). From nested-loop to join queries in OODB. *Proc. of the 20th Int'l Conference on Very Large Databases (VLDB)* (pp. 618–629). Santiago, Chile.

Steinbrunn, M., Peithner, K., Moerkotte, G., and Kemper, A. (1995). Bypassing joins in disjunctive queries. *Proc. of the 21st Int'l Conference on Very Large Databases (VLDB)* (pp. 228–238).

Suciu, D. and Wong, L. (1995). On two forms of strutural recursion. *Proc. of the 5th Int'l Conference on Database Theory (ICDT)* (pp. 111–124). Prague, Czech Republic.

Trinder, P. (1991). Comprehensions, a query notation for DBPLs. *Proc. of the 3rd Int'l Workshop on Database Programming Languages (DBPL)* (pp. 55–68). Nafplion, Greece.

Vance, B. (1992). Towards an Object-Oriented Query Algebra. Technical Report 91-008, Oregon Graduate Institute of Science & Technology.

Wadler, P. (1990a). Comprehending monads. *Conference on Lisp and Functional Programming* (pp. 61–78).

Wadler, P. (1990b). Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73, 231–248.

Wong, L. (1994). Querying Nested Collections. Ph.D. dissertation, University of Pennsylvania, Philadelphia.