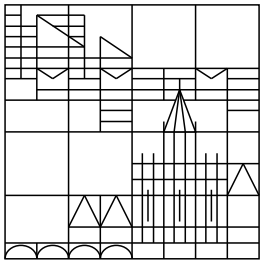

Universität Konstanz

Hybrid Strategies for Query Translation and Optimisation

Torsten Grust
Marc H. Scholl

Konstanzer Schriften in Mathematik und Informatik
Nr. 72, Oktober 1998

ISSN 1430–3558

Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften

Pastel

Persistent Application Systems, Technologies, Environments, and Languages (EP22552)

Research Theme 2, RT2.1

(Optimisation of ODMG/OQL Query Interfaces to PAS)

Deliverable RT2R1

Hybrid Strategies for Query Translation and Optimisation

INRIA, Konstanz U, Passau U, Pisa U, ETH Zürich

Abstract

The advanced data models for PAS that make these systems superior to their table-oriented antecedents (RDBMS) have an impact on the formalisms that are needed to capture these models and their appropiate query languages (eg. ODMG's OQL).

Queries that are nested to arbitrary depth, path expressions, and complex predicates pose challenges on the query translation process. The work package RT2.1 will identify generic (algebraic) operators that allow the efficient translation of such queries. These operators will account for the various bulk types the data models feature. Optimisation techniques have to be found, adapted, and validated.

PAS query languages allow to mix operations on bulk types and scalars (just like programming languages). Monad calculi treat bulk and scalar types in a uniform way, and allow for reasoning about arithmetics and general computation. This offers the perspective of a hybrid approach to query translation and optimisation, combining the power of algebra and calculus. A survey of the involved techniques and their interaction is the focus of this report.

1 Query Language Representations

"The limits of my language mean the limits of my world."
— Ludwig Wittgenstein (1922).

This report is concerned with the representation and optimisation of query languages for persistent application systems (PAS) for bulk data. Although the following dicussion mainly draws from the database area of query language research, its scope is meant to include the query (sub-)language for PAS in general.

It is the dominant characteristic of these query languages (also called *object language* from now on) to be *declarative*, that is 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. The PAS system itself maps a declarative query into a program which is then used to access the store for retrieval. Obvious advantages of such an approach include:

- Queries are considerably simpler to formulate correctly.
- Changes at the physical implementation level of the persistent store do not affect querying provided that the query mapping reflects these changes (physical data independence).
- If the PAS-internal query representation is amenable to equivalence preserving transformation, then the system may use query rewriting techniques in order to minimize query execution cost (e.g. response time).

The latter point in particular forbids the use of general purpose PAS programming languages for querying: equivalence theories are easier found and better understood for declarative languages than for procedural languages (if one can be found at all in the presence of, e.g., side-effects of language concepts like variable assignment).

Besides its amenability to rewriting, there are at least two further vital features of a suitable query language representation:

- The ability to capture the concepts (operations and types) of the object language completely.
- The suitability to serve as a starting point for a mapping from this representation to the primitives of the underlying persistent store.

Database research has come up with several proposals for declarative query languages each of which differ, among other aspects, in their degree of formality, expressivity, and potential for optimisation. Criteria like user friendliness and (user-level) syntax are of no concern here since we are going to discuss *internal* language representations. In what follows, the process of compiling a user-level query into its internal form will be of minor interest.

There are several dimensions along which we may compare query languages. Figure 1 emphasises three of these dimensions in order to structure the space of declarative query languages:

1.1 The Supported Type System

Query optimisation for relational database systems dominates the field of query language research by far. The only supported bulk type constructor is the *set (of tuple)* (or *relation*) constructor. Operators of relational query languages consume and produce set values.

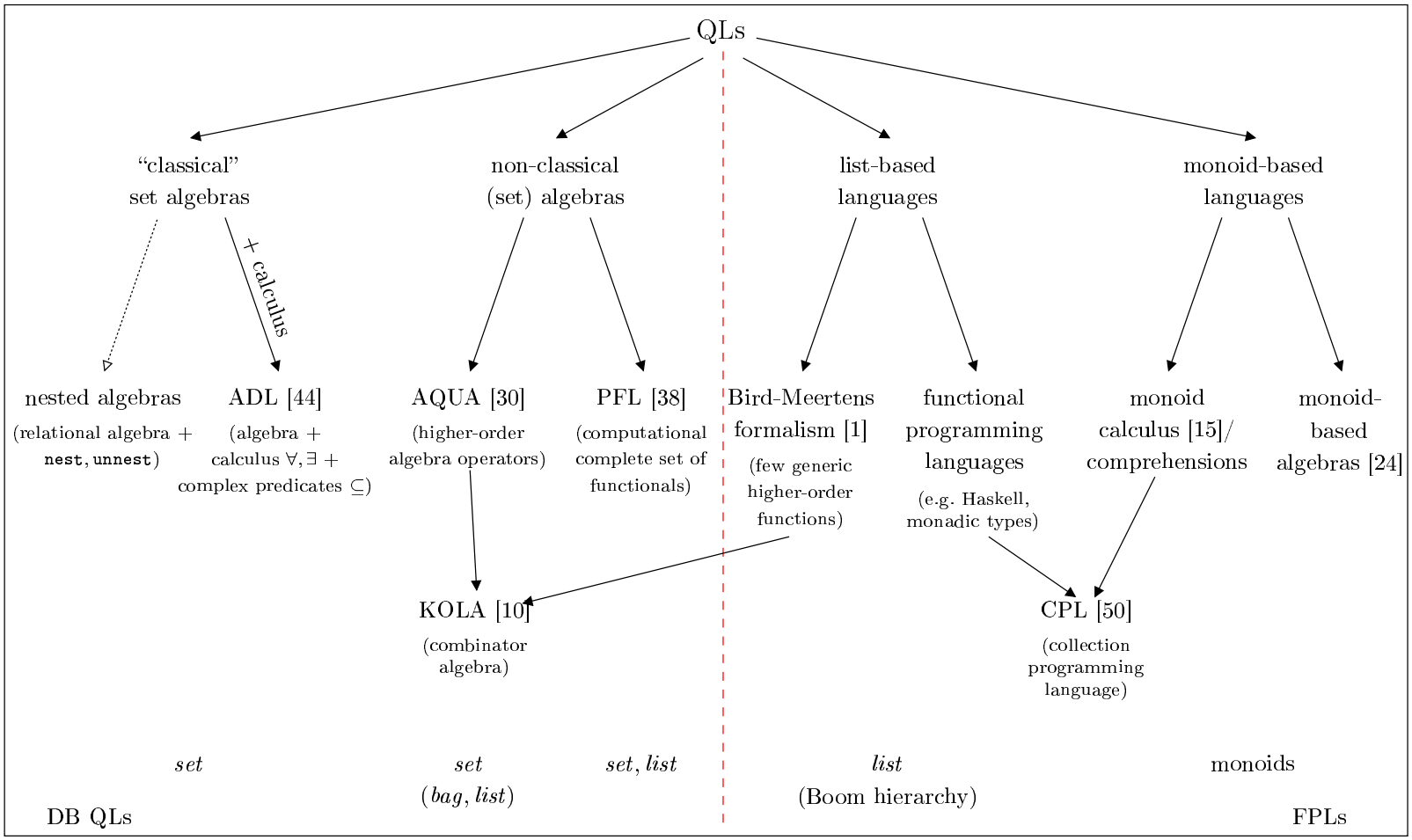

Figure 1: A space of declarative query languages.

Let us review non-first normal form (NF^2) relational systems [41] as an example of how the bulk type system affects the query language and its optimisation. Relational systems that violate the first normal form (1NF) restriction (whose type system allows for set-valued tuple components at arbitrary nesting depths) are accompanied by appropriate language extensions. Nesting in the type structure calls for query nesting capabilities in the corresponding language. Query optimisation for nested relational languages is not as well understood as for languages for 1NF relations. The dominant approach to this problem is to enrich 1NF algebras by *unnest* (μ) and *nest* (ν) operators: μ "flattens" a nested relation (with the cost of duplicating the formerly nested values) while ν forms a nested set of values that agree on a specified attribute value. Query processing and optimisation then proceeds in three steps: (1) use (sequences of) μ to transform the query input into 1NF form, (2) apply 1NF query optimisation techniques, (3) use ν to rebuild the potentially nested structure of the result. The efficiency of this approach clearly is problematic. Furthermore, algebraic rewriting is complicated by the relatively complex operators μ and ν (for which, in general, $\mu \neq \nu^{-1}$).

Solutions to these deficiencies include:

- Orthogonally nested algebras that directly allow to evaluate predicates on nests or projections on nested values. These languages reduce the ubiquitous need for unnesting prior to and nesting after query processing [42].
- Introduction of certain normal forms for relations (e.g., the partitioned normal form, PNF) that guarantee the losslessness of the ν operator.
- Incorporation of new algebraic operators that combine the effects of μ resp. ν and other algebraic operators in order to gain efficieny. This includes operators like a *nesting join*, $\triangle$ [43], the *binary grouping* Γ [12], or the *hierarchical join* of [40] which do not add expressive power to the algebras but are mainly needed for optimisation purposes. Such operators, however, enjoy few algebraic properties (w.r.t. to their interaction with other operators) making algebraic rewriting considerably hard.

In addition to arbitrary nesting, recent proposals for PAS data models include more bulk type constructors, in particular *bag* (*multiset*) and *list* [7]. Results from the relational query processing domain may or may not hold for these extended type systems. This complication is largely due to the algebraic differences between these collection type constructors (observe that the basic *value constructor* for *set*, set union $\cup$, is a commutative and idempotent function, while the respective constructor for *bag*, additive union $\uplus$, is commutative but not idempotent, and *list* concatentation $+\!\!+$ is neither; all of these are associative, however).

To appropriately manipulate values of richer type systems, the family of operators has to be adapted. In analogy to the relational algebra, primitives for bag and list manipulations have to be found. The interplay of these operators and their effect on values of different collection types have to be fixed. A language constructed in this way may be expected to be quite large (because of the need for operators like *list-set join*, *bag project*, etc.) and therefore difficult to handle. The EREQ project defined the AQUA data model and language [30] whose operators may operate on values of various bulk types. AQUA gains this generality by definining operators as *higher-order* functions that themselves take other functions as parameters. These functions may be different value

constructors (e.g. *set union* $\cup$ or *list concatenation* $+\!\!+$), thus turning one operator into its set-, bag-, or list-variant by parameterisation.

Query languages are bulk type oriented. The vast majority of work on query languages tends to neglect scalar operations like arithmetics, aggregation, or quantification. This is true although data warehouse queries and OLAP, applications which gained significant importance in recent past, heavily rely on exactly these operations. If the representation language knows about such operators at all, their interaction with bulk operators is often unclear. During the query rewriting phase, scalar operators are moved around as "black boxes" without algebraic properties (this is very much like the "unrevealed methods" problem in object-oriented database systems). As a result, execution plans typically possess a separate bulk oriented processing phase for which an optimizer can often find an efficient arrangement of bulk operators. The output of this phase is then passed on to an extra aggregation phase. An optimizer for a language that knows about the properties of scalar operators, however, would have the choice of folding these two phases into one, thus saving the cost of (1) storing the intermediate result of the first phase, and (2) performing the aggregation in a separate loop.

1.2 Formalism

Algebraic approaches to representation of declarative query languages dominate the field of research by far. The operators of query algebras are rather straightforward abstractions of the algorithms implemented by the underlying query engines. Join $\bowtie$, for example, is an abstraction that reduces join algorithms like *sort-merge join* or *hash join* to their algebraic properties (e.g. being commutative). As a consequence, it is a quite manageable task (which is nevertheless non-trivial if this is to be done efficiently) to map an expression tree of algebraic operators to a program that drives the query engine during the actual query execution. These programs are often represented as *physical algebras* as well.

The `select-from-where` block of the SQL and OQL families of query languages maps to π-σ-$\bowtie$ query trees and therefore is the best understood language construct. However, not all language constructs map to algebras that well. Aggregation, quantification, and (complex) predicates are more naturally represented by *calculus-style* languages since they directly include these concepts. Predicates with quantifiers like SQL's and OQL's `forall` resp. `exists` are easily represented and rewritten (e.g. brought into one of the several normal forms). Proposed algebras, however, lack a canonical representation of such predicates.

As an example, [11] enumerates five ways for an extended object-relational query algebra (with *anti-semijoin* $\overline{\rhd}$, binary grouping Γ, division $\div$, and set difference $\setminus$) to represent moderately complex queries featuring universal quantification. After examining 16 different special cases of variable bindings, the paper identifies $\overline{\rhd}$ as the most efficient choice of representation. We will see in Section 3 how a hybrid view of these queries leads to a concise and uniform treatment of all cases that achieves the same result with only two (calculus- and algebra-based) rewriting steps. The complexity of a purely algebraic treatment is explained in part by the inherently variable-less nature of algebras. Each of the above mentioned 16 cases lead to a very different algebraic representation of the only slightly modified original query.

Calculi come with further advantages, the existence of normal forms for calculus expressions being one of them. Normal forms help in recognising certain query patterns and thus may aid in reducing the size of optimizer rewrite rule

bases. We will also see that normalisation may also implement such complex query transformations as subquery unnesting.

Observations like the ones we just made led researchers to an incorporation of calculus-style language elements into query algebras (or, as e.g. in AMOS [13], to split the optimisation problem into subtasks which either rely on a calculus or algebra representation of queries).

The spectrum of formalisms includes

- pure, variable-free combinator algebras, e.g. KOLA [10, 8],
- algebras that allow for complex predicates (i.e. arbitrary disjunctions, conjunctions, and predicate formers like $\subseteq$),
- real hybrid languages providing a rich set of algebraic operators, complex predicates, and quantifiers; the input to an algebraic expression may be a calculus expression and vice versa (e.g. ADL [44, 43]),
- calculus-based languages.

Expressions of languages from different ends of the spectrum do significantly differ in style (not in expressivity, however). The following θ-$\bowtie$ is an instance of a hybrid expression (whose output is a set of $\langle x, y\rangle$ pairs; $\langle\cdot\rangle$ denotes the binary tuple constructor):

$$X \underset{x,y:\ x=2 \wedge x.A=y.A}{\bowtie} Y$$

The formulation of the conjunctive join predicate as well as the result construction makes use of *range variables* x and y whose intuitive semantics is to iterate over the collections X and Y respectively. As a consequence, the corresponding rewriting framework for a query language like the above has to deal with algebraic as well as predicate calculus rules. However, the expression explicitly represents the fact that we are dealing with a join query—which is a valuable hint to the plan generation phase.

The obvious algebraic optimisation strategy of "pushing down" the subpredicate $x = 2$ and rewriting into a selection on X in fact depends on a property of the *calculus* expression $x = 2 \wedge x.A = y.A$ (namely that variable y does not occur free in $x = 2$). The rewriting framework must provide a means for checking such properties. Observe that the property of "occuring free" is a notion introduced by the query representation formalism, not by the query expression itself. Query optimisation in such a framework therefore has to reason at the query *and* at the query representation level simultaneously. This might complicate matters.

The equivalent calculus expression

$$\{\langle x, y\rangle \mid x \leftarrow X, y \leftarrow Y, x = 2, x.A = y.A\}$$

makes the use of range variables and their binding explicit. The opportunity for optimisation is obvious and easily expressed. It does not come for free, however, to recognise that the query computes a join. Here, pattern matching or a similar effort is necessary.

Combinator[1] algebras at the other end of the spectrum are variable-free languages. KOLA [10] is a representative of these. Given the combinators (read: functions) π_i (extract the i-th component of a tuple), $\texttt{eq}$ (equality of two components of a pair), $\texttt{C}(f, x)$ with $\texttt{C}(f, x)y = f(x, y)$ (currying), and $\texttt{join}(p)\ (X, Y)$ (join of X and Y w.r.t. to predicate p which is applied to x, y-pairs) the above query reads

$$\texttt{join}((\texttt{C}(\texttt{eq}, 2) \circ \pi_1) \wedge (\texttt{eq} \circ \langle A, A\rangle))\ (X, Y)$$

It is a desirable feature of a combinator representation that the determination of bound resp.

1. A *combinator* is an expression of the λ-calculus in which no variable appears free.

free variables in queries is not necessary. Subqueries may be replaced and moved within the expression tree without concerns about variable renaming, variable capture and related problems [36]. The condition that we may push down the predicate $x = 2$ is rather expressed *structurally*. In *any* expression of the general form

$$\texttt{join}((p \circ \pi_1) \wedge q)\ (X, Y)$$

p may be pushed down the expression tree for X since p is applied after π_1 and thus cannot depend on the second join argument Y (an assurance justified by p's nature of being a combinator). It is a hard problem, however, to rewrite predicates into the revealing normal form $(p_i \circ \pi_i) \wedge (p_j \circ \pi_j) \wedge \ldots$ (*separated normal form*, SNF, in [9]). This is true for other rewriting techniques which are easily expressed in the presence of variables.

1.3 Root of Proposal

Declarative queries resemble programs written in a functional programming language (FPL). The predefined functions[2] (operators) of the query language are combined via *functional composition* to build a more or less complex query function. This composed function is then applied to a value (a named persistent root in the case of PAS, or database entry points, e.g. class extents or base relations, in the case of DBMS) to execute the query.

Such a view makes query processing subject to the equational theories (of the λ-calculus) found in the large body of literature on functional program transformation—esp. the work commonly known as *Bird-Meertens*' formalism which developed an extensive theory of program transformation techniques for an intentionally small family of generic combinators—and optimisation [1]. The obtained operational semantics is simple, but powerful. Computation proceeds by repeatedly reducing subexpressions (*redexes*) of a given query expression to normal form. Certain disciplines in performing these reductions, e.g. normal order reduction which reduces a function prior to its argument (if this is needed at all), help in making functional evaluation efficient. PFL, persistent functional language, takes these correspondences serious: PFL is a simple functional query language which has been developed into a computationally complete language while still being optimisable.

Most of the work in this field has been carried out in the context of the development of compilers and abstract machines for functional programming languages. Due to the type system of these languages the results tend to be *list*-biased, but this is not exclusively true. Many results are formulated for or may be carried over to more general type systems. In addition to *list*, the constructors *set*, *bag*, and *tree* (which together form a type hierarchy also known as the *Boom hierarchy* [5]) may orthogonally be used to build types. We have already emphasized the significance of these constructors in newer data model proposals. The latest release of the non-strict pure functional programming language Haskell has generalized some of its core construtcs (most notably *comprehensions*) to not work only on lists but on all *monadic* types, in which *set* and *bag* fit smoothly. Efforts to make the language more amenable to collection-oriented processing have recently made their way into the Haskell compilers [37] .

The query language CPL (collection programming language) [50] has been designed as a

2. We may include user-defined functions and methods here as long as they actually behave as functions: they have to preserve *referential integrity* and therefore are not allowed to cause side effects. This excludes *update methods*.

functional sublanguage with an particular emphasis on collection processing. CPL employs a particular algebraic collection abstraction, the *monoid*. We will pursue a similar language design based on the caterogical concept of *monads* (see Section 2).

A range of techniques developed by the FPL community has already been proven to be valuable in query language research. Notable among these techniques are

- the use of *higher-order functions* as query operators. Above we mentioned AQUA which achieves genericity of operators that way. Higher-order combinators `map` and `fold` cover generic recursive patterns of computation. The expressiveness of `fold` has actually been shown to be sufficient to cover ODMG's OQL. An optimiser for such a query representation may take advantage of the large body of knowledge on `fold` optimisations, e.g. *fold promotion* [24].

- Issues of *partial evaluation* of functional programs are relevant to query processing as well. The optimisation of query function applications for which some of the parameter are fixed (consider queries involving views) corresponds to the partial evaluation of a program whose static parameters are known.

- *Lazy evaluation*, a variant of normal order reduction in which the same subexpression is never reduced more than once, closely resembles the *demand-driven* operation of a pipeline of (physical) query operators: in an operator tree, nodes demand the production of input in a root-to-leafs manner, while child nodes proceed just as much with their computation as to be able to satisfy their parent's request. In such a system, *strictness analysis* reveales subqueries that have to be completely evaluated no matter what. These may be candidates for materialisation.

- Burstall/Darlington-like *unfold-fold transformations* [6] can be used to reveal the (recursive) structure of query language operators which in turn may be formulated in terms of "primitive" combinators like `map`. Query operators are no longer atomic building blocks of query expressions. The defining recursion equations of two or more operators may be "folded" which may result in a considerably more efficient function than the composition of these operators provides. *Deforestation* [49, 20, 32], a particular instance of this general technique, achieves the complete removal of intermediate data structures during query evaluation. A variant of deforestation provides the basic building block for [26] to develop transformations for the automatic derivation of pipelining execution plans from algebraic combinator queries.

Let us finally emphasize that the dashed line in Figure 1 does *not* mark a impenetrable border between database query languages and functional programming language research. Rather, in the light of the above, it denotes the permeable thin line between the two areas and should make us aware of the grey area in which our dicussion takes place.

Synopsis. We will proceed as follows. As the advanced orthogonal type systems of modern PAS have a major impact on query processing, we will organize our language framework around types, *not* operations. Section 2 elaborates this type-based language design. Efficiency considerations then motivate the step

from a type-based to an operation-based representation language in Section 3. The resulting hybrid query representation has beneficial impacts on query rewriting. This is shown in Section 3 as well. Section 4 describes the derivation of streaming (pipelining) programs from algebraic queries and completes the query processing framework. The derivation is based on program transformation techniques, *deforestation* in particular, and exemplifies how PAS query processing can draw benefit from knowledge established by the functional programming community. Section 5 concludes.

2 A Query Language Based On Types

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 [41]. 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 α *list* 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):

$$\begin{aligned}&[x_1, x_2, \ldots, x_n]\\ &\quad = \quad x_1 : (x_2 : (\cdots (x_n : [\,]) \cdots))\\ &\quad = \quad x_1 : x_2 : \cdots : x_n : [\,]\end{aligned}$$

The constructors `[]` and `(:)` are tightly connected to *list* in the sense that the algebra (α *list*, `[]`, `:`) is initial for the parameterized algebraic data type specification α *list* with signature (α *list*, *nil* $::\ \alpha$ *list*, *cons* $::\ \alpha \to \alpha$ *list* $\to$ α *list*). The principle extends to other algebras. Finitely many insertions into the empty set `{}` resp. into the empty bag `{|}` construct any finite set resp. bag. We will adopt this *insertion representation* for collections [46, 4] and denote the resulting algebra for type constructor τ by (α τ, $[\,]^\tau$, $\overset{\tau}{:}$) for the sake of notational uniformity, e.g. $[\,]^{set} = \{\}$ while $x \overset{set}{:} xs$

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.

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 $[x_1, \ldots, x_n]^{set}$ in what follows.

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

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

Definitions of *list append* ($+\!\!+$) and *additive union for bags* ($\uplus$) are derived likewise.

The above two equations define $\cup$ in terms of *structural recursion* over the constructors $[]^{set}$ and $(:^{set})$ (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, $(\alpha\ set, \{\}, \lambda e.\{e\}, \cup)$ is an initial algebra for the polymorphic set constructor in *union representation* with signature $(\alpha\ set, zero :: \alpha\ set, unit :: \alpha \to \alpha\ set, merge :: \alpha\ set \to \alpha\ set \to \alpha\ set)$. The closely related work of [15], for example, adopted the union representation while [4] exploits both viewpoints. Issues of expressiveness are examined by [46] and an in-depth comparison of insertion and union representation has been undertaken by [3]. 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 $sum = (num, 0, +)$ 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 $exists = (bool, \texttt{False}, ||)$, respectively. An existential quantifier is then readily obtained by defining the higher-order operator $(\exists)$ via structural recursion:

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

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 3.1 on query rewriting). This greatly helps to establish the necessary granularity of representation that makes all relevant query language constructs subject to inspection and transformation.

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.

2.1 Structural Recursion

Note that the function definitions (1) and (2) exhibit a common pattern of structural recursion. A large number of optimisations for expressions built by function composition, esp. *loop fusion* laws [16, 23, 14, 39], 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 (Figure 2).

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

`foldr` directly corresponds with the *structural recursion on the insertion representation*, or *"sri"*, of [2] and [46] in which the companion recursion scheme *"sru"* on the above mentioned union representation of algebras is introduced, too.

The effect of applying $\texttt{foldr}^{\tau}\ (\oplus)\ z$ to the collection $[x_1, x_2, \ldots, x_n]^{\tau}$ may also be understood by "visualizing" the application and its result as depicted in Figure 3.

$\texttt{foldr}^{\tau}$ folds $(\oplus)$ in between the collection elements by replacing the list constructors $\overset{\tau}{:}$ and $[]^{\tau}$ by $(\oplus)$ and z, respectively. $\texttt{foldr}^{\tau}$ is only conditionally well-defined which can be seen easily from (3). 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:

$$\begin{array}{rcl} y \overset{set}{:} (x \overset{set}{:} xs) & = & x \overset{set}{:} (y \overset{set}{:} xs) \\ x \overset{set}{:} (x \overset{set}{:} xs) & = & x \overset{set}{:} xs \end{array} \qquad (3)$$

and

$$y \overset{bag}{:} (x \overset{bag}{:} xs) \quad = \quad x \overset{bag}{:} (y \overset{bag}{:} xs)$$

The equations of (3) 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 $\texttt{foldr}^{\tau}\ (\oplus)\ z$ to be well-defined, $(\oplus)$ has to be commutative resp. idempotent whenever $(\overset{\tau}{:})$ is left-commutative resp. left-idempotent. This ensures the meaning of $\texttt{foldr}^{\tau}\ (\oplus)\ z$ 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 optimisation 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 [24] 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 4[5]. Records are internally represented as tu-

5. The primitive operators and the conditional are not special. If, for example, we model the booleans as the algebraic data type $bool = (bool, \texttt{False}, \texttt{True})$ and then define its associated $\texttt{foldr}^{bool}$ operator as

$$\begin{array}{rcl} \texttt{foldr}^{bool}\ f\ z\ \texttt{False} & = & z \\ \texttt{foldr}^{bool}\ f\ z\ \texttt{True} & = & f \end{array}$$

we have $\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 = \texttt{foldr}^{bool}\ e_2\ e_3\ e_1$ and $e_1\ \texttt{||}\ e_2 = \texttt{foldr}^{bool}\ \texttt{True}\ e_2\ e_1$. We prefer the familiar notation for these primitives in order to render expressions more readable.

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

Figure 2: Basic recursion combinator `foldr`.

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

Figure 3: Understanding the effect `foldr`.

ples.

2.2 Monad Comprehensions

Comprehensions have proven to be a very convenient target for the compilation of declarative query languages [35, 47, 50, 15, 24]. 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 [24].

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 $[\,x \mid x \leftarrow xs, p\ x\,]^{\tau}$ which operates similar to the relational selection $\sigma_p(xs)$ but is uniformly applicable to any data type τ. Note that for $\tau = \mathit{set}$ the relational calculus can be looked at as a specialisation 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*[6] [48]. 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 $(\mathit{zero}, \mathit{unit}, \mathit{map}, \mathit{join})$ from the algebraic data type τ with constructors $[]^{\tau}$ and $(\overset{\tau}{:})$ as follows:

$$
\begin{array}{lcl}
\mathit{zero}^{\tau} & = & \lambda x.\ []^{\tau} \\
\mathit{unit}^{\tau}\ x & = & x\ \overset{\tau}{:}\ []^{\tau} \\
\mathit{map}^{\tau}\ f & = & \texttt{foldr}^{\tau}\ (\lambda x\ xs.f\ x\ \overset{\tau}{:}\ xs)\ []^{\tau} \\
\mathit{join}^{\tau} & = & \texttt{foldr}^{\tau}\ (\overset{\tau}{+\!\!+})\ []^{\tau}
\end{array}
\qquad (4)
$$

where $(\overset{\tau}{+\!\!+})$ denotes the *union* operation (see (1)) for type τ. The required monad laws of [48] 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

6. If *zero* is missing, comprehensions are still sensibly defined, but the use of filter predicates is not permitted then.

$$
\begin{array}{lcll}
e & \to & v & \text{variables} \\
 & | & \texttt{c} & \text{constants} \\
 & | & (e_1, \ldots, e_n) & \text{tuples } (n \geq 1) \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \lambda vs.e & \text{abstraction} \\
 & | & \texttt{foldr}^{\tau}\ (\oplus)\ z & \text{reduction} \\
 & | & []^{\tau} \mid \overset{\tau}{:} & \text{constructors} \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{conditional} \\
 & | & (op)\ e_1 \ldots e_n & \text{primitives } (n \geq 0, op = \texttt{+}, \texttt{*}, ||, \ldots) \\
 & & & \\
\tau & \to & \mathit{set} \mid \mathit{bag} \mid \mathit{list} \mid \mathit{exists} \mid \mathit{all} & \text{algebraic data types} \\
 & | & \mathit{sum} \mid \mathit{prod} \mid \mathit{max} \mid \mathit{min} &
\end{array}
$$

Figure 4: Core representation language.

the so-called "Wadler identities"—to the above monad operations. Translation scheme $\mathcal{MC}$ in Figure 5 uses a simple recursion over the syntactic structure of the comprehenion qualifier list to implement the comprehension desugaring (in Figure 5, let p be an expression of result type *bool*, q, qs qualifiers, and e an arbitrary expression).

The above example comprehension is mapped to

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

which implements selection by mapping those elements to $[]^{\tau}$ that fail to satisfy the filter p (the $[]^{\tau}$ do not contribute to the result during the outer join^{τ} 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 (4)) 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 resp. left-idempotent, 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 $[\,x \mid x \leftarrow [1,2,3]^{\mathit{set}}\,]^{\mathit{list}}$, for example.

2.3 Monad Comprehensions as a Translation Target for OQL

As we cannot provide more than an intuition of the OQL to monad comprehension calculus mapping here, we refer to work reported in [24, 25] 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 [27]. In [21] 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 as show in Figure 6 (the

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

Figure 5: Mapping monad comprehensions to the language core.

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 7 summarizes the translation of further OQL clauses. The simplicity of $\mathcal{Q}$ is mainly due to its *uniformity* [46]: 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 1 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
```
(5)

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

$$
\texttt{[[}e\texttt{.sal} \mid e \leftarrow \mathit{emp}, d' \leftarrow \mathit{dept}, e\texttt{.dno} = d'\texttt{.no}, d\texttt{.no} = d'\texttt{.no}\texttt{]}^{\mathit{sum}} \mid d \leftarrow \mathit{dept}, d\texttt{.budg} < 10000\texttt{]}^{\mathit{bag}}
\tag{6}
$$

□

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 normalisation 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 Hybrid Query Rewriting

"The mutual interdependence of thought and word

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

Figure 6: Translating OQL's `select-from-where` block.

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

Figure 7: Translation of some OQL clauses to monad comprehensions.

illuminates clearly the truth that languages are not really means for representing already known truths but are rather instruments for discovering previously unrecognized ones."
— Wilhelm von Humboldt.

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 join variants like *nestjoin* $\triangle$ [44, 43], *semijoin* $\rhd$, or *antijoin* $\overline{\rhd}$).

The monad comprehension calculus provides hooks for optimisations 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.

3.1 Transformation of Monad Comprehension Programs

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 normalisation implements a number of query unnesting strategies that have been proposed in the literature.

The normalisation rules are given in Figure 8. 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 (9) and (10) 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 (9), 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 $p = \lambda x\ y.p\ y\ \&\&\ q\ x\ y$. 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 (9) implements a considerable generalisation of Kim's *type J* unnesting for nested SQL queries [28] in which q was fixed to be the equality test.

Furthermore, Kim's *type N* optimisation can be understood as an instance of Rule (10). The nesting in the `where`-clause of the following SQL query is readily removed by comprehension normalisation:

```
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 7) to the above query gives a comprehension which matches the left-hand side of Rule (10). Note how $\mathcal{Q}$ translates the element test into an existential quantification:

$$[\![f\ x \mid x \leftarrow xs, [\![g\ x = h\ y \mid y \leftarrow ys, p\ y]\!]^{exists}]\!]^{set} \underset{(10)}{\rightarrow} [\![f\ x \mid x \leftarrow xs, y \leftarrow ys, p\ y, g\ x = h\ y]\!]^{set}$$

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 optimisation 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 Combinators

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

$$[\, e \mid qs, x \leftarrow []^{\sigma}, qs' \,]^{\tau} \quad \rightarrow \quad []^{\tau} \tag{7}$$

$$[\, e \mid qs, x \leftarrow y \overset{\sigma}{:} ys, qs' \,]^{\tau} \quad \rightarrow \quad [\, e[y/x] \mid qs, qs'[y/x] \,]^{\tau} \overset{\tau}{+\!\!+} [\, e \mid qs, x \leftarrow ys, qs' \,]^{\tau} \tag{8}$$

$$[\, e \mid qs, x \leftarrow [\, e' \mid qs'' \,]^{\sigma}, qs' \,]^{\tau} \quad \rightarrow \quad [\, e[e'/x] \mid qs, qs'', qs'[e'/x] \,]^{\tau} \tag{9}$$

$$[\, e \mid qs, [\, p \mid qs'' \,]^{exists}, qs' \,]^{\tau} \quad \rightarrow \quad [\, e \mid qs, qs'', p, qs' \,]^{\tau} \quad (\tau \text{ left-idempotent}) \tag{10}$$

Figure 8: Monad comprehension normalisation algorithm.

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 optimisation 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 optimisation 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 optimisation 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 [24].

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).

Figure 9 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 [12, 43, 40].

It is actually only a small step from combinator definitions to the associated translation

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

Figure 9: Monad comprehension-based definitions of algebraic combinators.

rules they induce, so that we only list few of them in Figure 10. Successive applications of these rewriting rules will translate a comprehension expression into an equivalent combinator program. In Figure 10, 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 (15) realize deconstruction of the tuple-valued result produced by the `join` (let `fst` $(x,y) = x$ and `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 (5) resp. its monad comprehension equivalent (6). By interleaving the translation—driven by the rules of Figure 10—with rewriting steps on the calculus level, we obtain an efficient combinator expression in only five rewriting steps (Figure 11, 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 [46], 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).

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

Example 2 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:

```
select distinct  x
           from  xs as x
          where  exists y in ys:
                 exists z in zs:
                        q x z and p y z
```
(16)

Due to $q\ x\ z$ `&&` $p\ y\ z$ being dependent on the free variable x we are stuck after one application of the existential unnesting Rule (10). 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 (Figure 12). 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`.

$$[\, e \mid qs, x \leftarrow xs :: \sigma, p\ x, qs' \,]^{\tau} \quad \rightarrow \quad [\, e \mid qs, x \leftarrow \mathtt{filter}^{\sigma} p\ xs, qs' \,]^{\tau} \tag{11}$$

$$[\, f\ e \mid qs \,]^{\tau} \quad \rightarrow \quad \mathtt{map}^{\tau}\ f\ [\, e \mid qs \,]^{\tau} \tag{12}$$

$$[\, e \mid qs \,]^{agg} \quad \rightarrow \quad agg^{\tau}\ \mathtt{id}\ [\, e \mid qs \,]^{\tau} \tag{13}$$

$$[\, e \mid qs, x \leftarrow xs :: \sigma, [\, p \mid y \leftarrow ys :: \sigma' \,]^{exists}, qs' \,]^{\tau} \quad \rightarrow \quad [\, e \mid qs, x \leftarrow \mathtt{semijoin}^{\sigma\sigma'} p\ xs\ ys, qs' \,]^{\tau} \tag{14}$$

$$[\, e \mid qs, x \leftarrow xs :: \sigma, y \leftarrow ys :: \sigma', p\ x\ y, qs' \,]^{\tau} \quad \rightarrow \quad [\, e[\mathtt{fst}\ v/x][\mathtt{snd}\ v/y] \mid qs, v \leftarrow \mathtt{join}^{\sigma\sigma'} p\ (\lambda x\ y.(x,y))\ xs\ ys, qs'[\mathtt{fst}\ v/x][\mathtt{snd}\ v/y] \,]^{\tau} \tag{15}$$

Figure 10: Combinator pattern matching.

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

Figure 11: Derivation of a combinator representation for the "low budget" query.

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 monad 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 (16), 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: $[\,q\ \&\&\ p \mid x \leftarrow xs\,]^{\tau} = q\ \&\&\ [\,p \mid x \leftarrow xs\,]^{\tau}$, $\tau \in \{exists, all\}$. The optimizer can infer an efficient `semijoin`-based plan for query (16) in the presence of these additional rules *and* the combinator patterns of Figure 10. The resulting combinator query is depicted in Figure 13. □

To conclude this section on query translation, let us briefly review the transformation of a class of 16 queries discussed in [11]. All of

$$
\begin{array}{ll}
 & [\,x \mid x \leftarrow xs :: \sigma, [\,[\,q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid z \leftarrow zs :: \sigma'']^{exists} \mid y \leftarrow ys :: \sigma']^{exists}\,]^{set} \\
\underset{(10)}{=} & [\,x \mid x \leftarrow xs, y \leftarrow ys, [\,q\ x\ z\ \texttt{\&\&}\ p\ y\ z \mid z \leftarrow zs]^{exists}\,]^{set} \\
\underset{(\texttt{cross})}{=} & [\,\texttt{fst}\ e \mid e \leftarrow \texttt{cross}^{\sigma\sigma'}\ xs\ ys, [\,q(\texttt{fst}\ e)z\ \texttt{\&\&}\ p(\texttt{snd}\ e)z \mid z \leftarrow zs]^{exists}\,]^{set} \\
\underset{(\texttt{semijoin})}{=} & [\,\texttt{fst}\ e \mid e \leftarrow \texttt{semijoin}^{\sigma\sigma''}\ (\lambda e\ z.q(\texttt{fst}\ e)z\ \texttt{\&\&}\ p(\texttt{snd}\ e)z) \\
 & \qquad\qquad (\texttt{cross}^{\sigma\sigma'}\ xs\ ys)\ zs]^{set} \\
\underset{(\texttt{map})}{=} & \texttt{map}^{set}\texttt{fst}\ (\texttt{semijoin}^{\sigma\sigma''}\ (\lambda e\ z.q(\texttt{fst}\ e)z\ \texttt{\&\&}\ p(\texttt{snd}\ e)z) \\
 & \qquad\qquad (\texttt{cross}^{\sigma\sigma'}\ xs\ ys)\ zs)
\end{array}
$$

Figure 12: Tupling resolves the variable scoping conflict.

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

Figure 13: Quantifier exchange and descoping improve the translation.

these are treated alike in our representation. Here, we will concentrate on the most interesting case.

Example 3 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 [34, 43]. 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* (Figure 14). If the subquery is closed we can fire Rule (11) which introduces a `filter` to implement the selection w.r.t. p. This case is depicted in the first chain of trans-

formations 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 9) in both cases. Application of the **antijoin** rule already completes both transformations.

The derived **antijoin** expressions are identical to those that have been identified as the most efficient alternatives by [11]. □

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 optimisation 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 optimisation context.

4 Streaming Programs for Combinator Queries

The widely accepted approach of designing a query algebra as a set of combinators facilitates algebraic query optimisation. 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: [22] 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 [31] 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] [49], i.e. transformed so that they allocate no intermediate data structures. Deforestation is

7. Deforestation removes intermediate data structures, which Wadler collectively refers to as trees. Hence the name *deforestation*.

Subquery closed:

$$
\begin{array}{ll}
& [\,x \mid x \leftarrow xs :: bag, [\,q\ x\ y \mid y \leftarrow ys :: bag, p\ y\,]^{all}\ ]^{bag} \\
\underset{(\texttt{filter})}{=} & [\,x \mid x \leftarrow xs, [\,q\ x\ y \mid y \leftarrow \texttt{filter}^{bag}\ p\ ys\,]^{all}\ ]^{bag} \\
\underset{(\texttt{antijoin})}{=} & \texttt{antijoin}^{bag\,bag}\ q\ xs\ (\texttt{filter}^{bag}\ p\ ys)
\end{array}
$$

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

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

Figure 14: Rewriting universal quantification.

a specific instance of the general *unfold-fold* program transformation strategy [6]: 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 ($\texttt{foldr}^{\tau}$) 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 15 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 [1, 23, 33]. Most importantly for our purposes, however, *cheap deforestation* (or `foldr-build` deforestation) is possible for `foldr`-based programs [20, 29, 19]. 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 optimisation of *listful programs*. However, the principle nicely adapts to the monadic type system [26].

4.1 Cheap Deforestation

How would a streaming program for the combinator query $\texttt{map}^{\tau}\ f\ (\texttt{filter}^{\tau}\ p\ s)$ look like? Instead of allocating the temporary result of the $\texttt{filter}^{\tau} p$ pass over s, a streaming plan would scan s, and *immediately* apply f but

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

Figure 15: Some combinators expressed via structural recursion.

only to those objects x that satisfy p (and otherwise just drop x). The corresponding program does not allocate intermediate results and would read

$$
\begin{array}{l}
\texttt{foldr}^{\tau}\ (\lambda x\ xs. \\
\quad \texttt{if}\ p\ x\ \texttt{then}\ f\ x\ \overset{\tau}{:}\ xs\ \texttt{else}\ xs)\ \ []^{\tau}\ \ s
\end{array}
\tag{17}
$$

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 $\texttt{foldr}^{\tau}$ comes into play. Recall from (3) that $\texttt{foldr}^{\tau}\ (\oplus)\ z$ traverses its argument of type $\alpha\ \tau$ and replaces constructor $(\overset{\tau}{:})$ by $(\oplus)$ resp. $[]^{\tau}$ 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 $\texttt{foldr}^{\tau}$ application at all. Cheap deforestation does exactly this for $\tau = \mathit{list}$ [20].

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 $\beta-$*abstraction* of g w.r.t. the constructors it uses: $\lambda c\ n.g[c/\overset{\tau}{:}][n/\ []^{\tau}]$. The result of applying $\texttt{foldr}^{\tau}\ (\oplus)\ z$ to g may now readily computed at plan generation time:

$$
\begin{array}{l}
\texttt{foldr}^{\tau}\ (\oplus)\ z\ g \\
\quad \underset{(\star)}{=} \quad (\lambda c\ n.g[c/\overset{\tau}{:}][n/\ []^{\tau}])\ (\oplus)\ z \\
\quad \underset{\beta}{=} \quad g[\oplus/\overset{\tau}{:}][z/\ []^{\tau}]
\end{array}
\tag{18}
$$

Note that the rightmost expression is a streaming program for the composition $\texttt{foldr}^{\tau}\ (\oplus)\ z\ g$: instead of constructing a temporary result using $(\overset{\tau}{:})$ and $[]^{\tau}$, 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 $[]^{\tau}$ to produce its result since we need to replace *all* constructors to implement streaming correctly. This condition is met by all combinators in Figure 15. 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 [20]. If we define $\texttt{build}^{\tau}\ g = g\ (\overset{\tau}{:})\ []^{\tau}$ (here, the type abstraction allows us to supply correctly typed constructor parameters to g), we can render $(\star)$ more concisely

as

$$\texttt{foldr}^{\tau}\ (\oplus)\ z\ (\texttt{build}^{\tau}\ g') \quad = \quad g'\ (\oplus)\ z \tag{19}$$

(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 definitions (Figure 15). We additionally reexpress the combinators by **build** as shown in Figure 16 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 (19).

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 4 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

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

The deforestation rule is applied once which produces

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

Finally, unfolding the outer **build** gives the desired streaming program we have shown before (17)

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

□

Example 5 To complete the translation of the "low budget" query (5), let us deforest its combinator equivalent which we have derived in Section 2.3 (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

$$\begin{array}{l}\texttt{foldr}^{bag}\ (\lambda s\ ss.\texttt{foldr}^{bag}\ (+)\ 0\ s\ \overset{bag}{:}\ ss)\ []^{bag} \\ \quad (\texttt{build}^{bag}\ (\lambda c\ n.\texttt{foldr}^{bag}\ (\lambda d\ ds. \\ \quad\quad \texttt{build}^{bag}\ (\lambda c_1\ n_1.\texttt{foldr}^{bag}\ (\lambda e\ es. \\ \quad\quad\quad \texttt{if}\ e.\texttt{dno} = d.\texttt{no}\ \texttt{then}\ e.\texttt{sal}\ \texttt{`}c_1\texttt{`}\ es \\ \quad\quad\quad \texttt{else}\ es)\ \texttt{n}_1\ emp)\ \texttt{`}c\texttt{`}\ ds) \\ \quad n\ \ (\texttt{build}^{bag}\ (\lambda c_2\ n_2. \\ \quad\quad\quad \texttt{foldr}^{bag}\ (\lambda b\ bs. \\ \quad\quad\quad\quad \texttt{if}\ b.\texttt{budg} < 10000\ \texttt{then}\ b\ \texttt{`}c_2\texttt{`}\ bs \\ \quad\quad\quad\quad \texttt{else}\ bs)\ n_2\ dept))))\end{array}$$

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

$$\begin{array}{l}\texttt{foldr}^{bag}\ (\lambda d\ ds. \\ \quad \texttt{foldr}^{bag}\ (+)\ 0\ (\texttt{build}^{bag}\ (\lambda c_1\ n_1. \\ \quad\quad \texttt{foldr}^{bag}\ (\lambda e\ es. \\ \quad\quad\quad \texttt{if}\ e.\texttt{dno} = d.\texttt{no}\ \texttt{then}\ e.\texttt{sal}\ \texttt{`}c_1\texttt{`}\ es \\ \quad\quad\quad \texttt{else}\ es)\ n_1\ emp))\ \overset{bag}{:}\ ds) \\ \quad []^{bag}\ \ (\texttt{build}^{bag}\ (\lambda c_2\ n_2. \\ \quad\quad\quad \texttt{foldr}^{bag}\ (\lambda b\ bs. \\ \quad\quad\quad\quad \texttt{if}\ b.\texttt{budg} < 10000\ \texttt{then}\ b\ \texttt{`}c_2\texttt{`}\ bs \\ \quad\quad\quad\quad \texttt{else}\ bs)\ n_2\ dept))\end{array}$$

This opens the possibility to finally merge the aggregate into the grouping phase to give the resulting streaming plan in which aggregation,

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

Figure 16: `foldr-build` forms of some combinators

join, and grouping have been fused completely:

$$
\begin{array}{l}
\texttt{foldr}^{bag}\,(\lambda d\ ds. \\
\quad \texttt{if}\ d.\texttt{budg} < 10000 \\
\quad\ \texttt{then}\ (\texttt{foldr}^{bag}\,(\lambda e\ es. \\
\quad\quad \texttt{if}\ e.\texttt{dno} = d.\texttt{no}\ \texttt{then}\ e.\texttt{sal} + es \\
\quad\quad \texttt{else}\ es)\ 0\ emp)\ \mathbin{(\!\!:\!\!)}^{bag}\ ds \\
\quad \texttt{else}\ ds)\ []^{bag}\ dept
\end{array}
$$

□

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 [19]. A suitable generalisation of this observation to monadic types leads to the modified comprehension translation scheme $\mathcal{MC}^*$ in Figure 17. $\mathcal{MC}^*$ 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 *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.

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 $\mathbin{(\!\!:\!\!)}^{set}$ 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

$$\texttt{build}^{set}\,g \quad \stackrel{\rightarrow}{=} \quad \texttt{dupelim(build}^{list}\,g\texttt{)}$$

(read in the direction of the arrow, this allows to "push duplicate elimination out" of the plan

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

Figure 17: Alternative translation scheme for monad comprehensions.

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 memoisation which disrupts the fully stream-based execution of queries.

The loop fusion techniques for the derivation of iterative programs from relational queries developed in [17, 18] 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., [39] 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, [39] accompanies specific rules with rather complex provisos (e.g. strictness or distributivity of functions) that cannot easily be asserted, let alone be checked syntactically [3]. Query deforestation is able to deforest the *unsafe programs* (queries in which a nested subquery traverses partial results computed by an outer query) of [14]. 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.

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 resp. 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. Transforming a user-level query into a PAS storage access program 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. The short cuts $\mathcal{MC} \circ \mathcal{Q}$ resp. $\mathit{deforestation} \circ \mathcal{MC}^* \circ \mathcal{Q}$—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 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 here. 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 [45] which we have found to be expressible by tupling [25].

References

[1] Richard S. Bird. An Introduction to the Theory of Lists. In Manfred Broy, editor, *Logic of Programming and Calculi of Design*, volume 36 of *NATO ASI Series*, pages 5–42. Springer Verlag, April 1987.

[2] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally Embedded Query Languages. In *Proc. of the Int'l Conference on Database Theory (ICDT)*, pages 140–154, Berlin, Germany, October 1992.

[3] Val Breazu-Tannen and Ramesh Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In *Proc. of th 18th Int'l Colloquium on Automata, Languages and Programming*, pages 60–75, Madrid, Spain, July 1991.

[4] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1):3–48, 1995.

[5] Alexander Bunkenburg. The Boom Hierarchy. In Kevin Hammond and O'Donnell John T., editors, *Proc. of the Workshop on Functional Programming*, Workshops in Computing, Ayr, Scotland, 1993. Springer Verlag.

[6] Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[7] Rick G. Cattell and Douglas K. Barry, editors. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, California, 1997. Release 2.0.

[8] Mitch Cherniack. Translating Queries into Combinators. Technical report, Department of Computer Science, Brown University Database Group, 1996.

[9] Mitch Cherniack and Stan Zdonik. Changing the Rules: Transformation for Rule-Based Optimizers. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, Seattle, Washington, June 1998.

[10] Mitch Cherniack and Stanley B. Zdonik. Rule Languages and Internal Algebras for Rule-Based Optimizers. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 401–412, June 1996.

[11] Jens Claussen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing Queries with Universal Quantification in Object-Oriented Databases. In *Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB)*, Athens, Greece, August 1997.

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

[13] Gustav Fahl and Tore Risch. Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4):261–281, November 1997.

[14] Leonidas Fegaras. Efficient Optimization of Iterative Queries. In *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, pages 200–225, August 1993.

[15] Leonidas Fegaras and David Maier. Towards an Effective Calculus for Object Query Languages. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, May 1995.

[16] Johann Christoph Freytag and Nathan Goodman. Rule-Based Translation of Relational Queries into Iterative Programs. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 206–214, Washington, D.C., USA, May 1986.

[17] Johann Christoph Freytag and Nathan Goodman. Translating Aggregate Queries into Iterative Programs. In *Proc. of the 12th Int'l Conference on Very Large Data Bases (VLDB)*, pages 138–146, Kyoto, Japan, August 1986.

[18] Johann Christoph Freytag and Nathan Goodman. On the Translation of Relational Queries into Iterative Programs. *ACM Transactions on Database Systems*, 14(1):1–27, March 1989.

[19] Andrew John Gill. *Cheap Deforestation for Non-strict Functional Languages*. Ph.D. dissertation, Department of Computing Science, University of Glasgow, January 1996.

[20] Andy Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Functional Programming & Computer Architecture*, April 1993.

[21] Dieter Gluche, Torsten Grust, Christof Mainberger, and Marc H. Scholl. Incremental Updates for Materialized OQL

Views. In François Bry, Raghu Ramakrishnan, and Kotagiri Ramamohanarao, editors, *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD'97)*, number 1341 in Lecture Notes in Computer Science (LNCS), pages 52–66, Montreux, Switzerland, December 1997. Springer Verlag.

[22] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 102–111, Atlantic City, New Jersey, USA, 1990.

[23] Malcolm Grant. Homomorphisms and Promotability. In Jan L.Ã. van de Snepscheut, editor, *Proc. of the Int'l Conference on Mathematics of Program Construction*, number 375 in Lecture Notes in Computer Science (LNCS), pages 335–347, Groningen, The Netherlands, June 1989. Springer Verlag.

[24] Torsten Grust, Joachim Kröger, Dieter Gluche, Andreas Heuer, and Marc H. Scholl. Query Evaluation in CROQUE—Calculus and Algebra Coincide. In Carol Small, Paul Douglas, Roger Johnson, Peter King, and Nigel Martin, editors, *Proc. of the 15th British National Conference on Databases (BNCOD15)*, number 1271 in Lecture Notes in Computer Science (LNCS), pages 84–100, London, Birkbeck College, July 1997. Springer Verlag.

[25] Torsten Grust and Marc H. Scholl. 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, March 1996.

[26] Torsten Grust and Marc H. Scholl. Query Deforestation. Technical Report 68/1998, Faculty of Mathematics and Computer Science, Database Research Group, University of Konstanz, June 1998.

[27] Graham J. L. Kemp, Jesus J. Iriarte, and Peter M. D. Gray. Efficient Access to FDM Objects Stored in a Relational Database. In *Proc. of the 12th British National Conference on Databases (BNCOD)*, pages 170–186, Guildford, UK, 1994.

[28] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[29] John Launchbury and Tim Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, La Jolla, California, June 1995.

[30] Theodore W. Leung, Gail Mitchell, Bharathi Subramanian, Stanley B. Zdonik, and other. The AQUA Data Model and Algebra. In *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, September 1993.

[31] Daniel F. Lieuwen and David J. DeWitt. A Transformation Based Approach to Optimizing Loops in Database Programming Languages. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 91–100, San Diego, California, June 1992.

[32] Simon David Marlow. *Deforestation for Higher-Order Functional Programs*. Ph.D. dissertation, Department of Computing Science, University of Glasgow, September 1995.

[33] Erik Meijer, Marten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, number 523 in Lecture Notes in Computer Science (LNCS), pages 124–144, Cambridge, USA, 1991. Springer Verlag.

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

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

[36] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1987.

[37] Simon L. Peyton Jones. Bulk Types with Class. Glasgow Workshop on Functional Programming, July 1996.

[38] Alexandra Poulovassilis and Carol Small. Algebraic Query Optimisation for Database Programming Languages. *The VLDB Journal*, 5(2):119–132, 1996.

[39] Alexandra Poulovassilis and Carol Small. Formal Foundations for Optimising Aggregation Functions in Database Programming Languages. In *Proc. of the 6th Int'l Workshop on Database Programming Languages (DBPL)*, Estes Park, Colorado, USA, 1997.

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

[41] Hans-Jörg Schek and Marc H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

[42] Marc H. Scholl. Extensions to the Relational Data Model. In Peri Loucopoulos and Roberto Zicari, editors, *Advances in Conceptual Modelling and CASE: An Integrated View of Information Systems Development*, pages 163–182. John Wiley & Sons, New York, 1992.

[43] Hennie J. Steenhagen. *Optimization of Object Query Languages*. Ph.D. dissertation, Department of Computer Science, University of Twente, 1995.

[44] Hennie J. Steenhagen, Peter M.G. Apers, Henk M. Blanken, and Rolf A. de By. From Nested-Loop to Join Queries in OODB. In *Proc. of the 20th Int'l Conference on Very Large Databases (VLDB)*, pages 618–629, Santiago, Chile, September 1994.

[45] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing Joins in Disjunctive Queries. In *Proc. of the 21st Int'l Conference on Very Large Databases (VLDB)*, pages 228–238, September 1995.

[46] Dan Suciu and Limsoon Wong. On Two Forms of Strutural Recursion. In Georg Gottlob and Moshe Y. Vardi, editors, *Proc. of the 5th Int'l Conference on Database Theory (ICDT)*, number 893 in Lecture Notes in Computer Science

(LNCS), pages 111–124, Prague, Czech Republic, January 1995. Springer Verlag.

[47] Phil Trinder. Comprehensions, a Query Notation for DBPLs. In *Proc. of the 3rd Int'l Workshop on Database Programming Languages (DBPL)*, pages 55–68, Nafplion, Greece, 1991.

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

[49] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[50] Limsoon Wong. *Querying Nested Collections*. Ph.D. dissertation, University of Pennsylvania, Philadelphia, August 1994.