

# Comprehending Queries

Torsten Grust

University of Konstanz  
Department of Computer and Information Science  
Database Systems Research Group  
Torsten.Grust@uni-konstanz.de

**1** There are no compelling reasons why database-internal query representations have to be designated by *operators*. This thesis describes a world in which *datatypes* determine the comprehension of queries. In this world, a datatype is characterized by its algebra of value constructors. These algebras are principal. Query operators are secondary in the sense that they simply box (recursive) programs that describe how to form a query result by application of datatype constructors. Often, operators will be unboxed to inspect and possibly rewrite these programs. Query optimization then means to deal with the transformation of programs.

The predominant role of the constructor algebras suggests that this model understands queries as mappings between such algebras. The key observation that makes the whole approach viable is that (a) *homomorphic* mappings are expressive enough to cover declarative user query languages like OQL or recent SQL dialects, and, at the same time, (b) a single program form suffices to express homomorphisms between constructor algebras. Reliance on a single combining form, *catamorphisms*, renders the query programs susceptible to *Constructive Algorithmics*, an effective and extensive algebraic theory of program transformations.

The complete text of this thesis may be downloaded by following the reference [Gru99].

## 1 Database Queries and Homomorphisms

**2** As sketched in the abstract, the consistent comprehension of queries as mapping between datatypes, *i. e.* their algebras of value constructors, provides the guide rail throughout the entire text.

The thesis rediscovers well-known query optimization knowledge on sometimes unusual paths that are more practicable to follow for an optimizer, though. Solutions previously proposed by others can be simplified and generalized mainly due to the clear account of the structure of queries that the monad comprehension calculus—thanks to its density—provides. The calculus effectively supports query optimization in the presence of grouping, various forms of nesting, aggregates, and quantifiers. Although built on top of abstract concepts like homomorphisms and monads, this query model is specific enough to grasp implementation issues, such as the generation of stream-based (pipelined) query execution plans, whose treatment has traditionally been delayed until query runtime.

It is the main objective of the thesis to show that catamorphisms and monad comprehensions enable a comprehension of queries that is *effective* and easily *exploitable* inside a query optimizer.

**3** How would you go about and try to comprehend database queries?

Open the cover and start to dismantle a database system. Then unbox the query engine and disassemble its parts. You end up with a set of query operators that may be combined in various but well-defined ways, the engine's operator algebra. The algebra tells you how the operators fit together and you start to play and combine operators to form queries. But as you reach for a *join* operator you hear clatter. In fact, all operators clatter as you shake them. Apparently you cannot develop a complete comprehension of the query engine if you do not further unbox the inner workings of the operators.—This text is an exploration of what you will discover as soon as you break an operator's case.

**4** In the course of this exploration we will soon realize that there is a single principle action that is pervasive inside all operators: the *construction* of values, which we will represent by the function symbol *cons*. Once we unfold the operators and inspect their definitions, we will find these operators to merely provide structure—a program—that controls application(s) of *cons*.

We will encounter construction in various instantiations, *e. g.*, as insertion of an element into a collection or incremental computation of an aggregate value, but these instantiations share so many properties that we will often do

without telling them apart. Unlike other query models, we do not let collection types play an exceptional role. This sets the scene for a comprehension of queries which acknowledges the presence of query constructs other than bulk operations.

The predominance of *cons* motivates the starting point of our exploration. From the start, we let the construction of values dominate our understanding of queries and then work our way bottom-up. To effectively reason about construction we will exploit algebras of value constructors. In fact, these are the primary algebras we will work with and it is the programs that build terms over these algebras that cause the clatter you have heard. As operators merely box such terms we can study the action of operators by actually examining how they construct values. The resulting operator algebra, however, is secondary in this text.

**5** Unboxing the operators gives us a rather fine control over value construction and we will see how a query optimizer can benefit from this control. At the same time, unboxing also implies that we have to deal with what we find inside: programs. Query analysis and transformation in this model amounts to analyze and transform programs. Throughout the entire text we will make good use of techniques native to the program transformation domain and establish well-known as well as invent novel query transformations this way.

To base a query optimizer on these techniques means that we have to be restrictive about the program forms we may admit. Only then we can assure that the optimizer can operate as a program transformation system free of the need for external guidance or *Eureka steps*.

Here, the algebras of value constructors provide a point of reference. The only program forms we will admit are those that mimic the structure (of the recursive type) of the values they analyze and construct, *i. e.*, those that perform structural recursion. This restrictive discipline will render programs as *homomorphic mappings* between algebras of value constructors. It is worthwhile to dwell on this thought a little longer.

If a query  $h$  is a mapping between types  $A$  and  $B$ , how can we go about and represent  $h$  inside a computer or, more specifically, a query engine? Under the proviso that  $A$  is finite we could exploit a lookup table to encode  $h$ . But what if  $A$  is infinite (as are the domains we define query languages over)? Then, the first thing we need is a finite (recursive) description  $F$  of the elements in  $A$ . (The algebra  $\alpha : FA \rightarrow A$  describes how the elements of  $A$  are actually constructed.) Internally, the query will thus be a mapping of type  $FA \rightarrow B$ . Second, for the encoding of  $h$  to be finite, too, it has to track the description  $F$  of  $A$ , which is nothing else than the hand-waving way of stating that  $h$  is a

homomorphism. This describes the basic understanding of queries in this text already fairly well. To get a bearing on this process, the overall picture of the involved mappings is

$$\begin{array}{ccc}
 FA & \xrightarrow{\alpha} & A \\
 \downarrow Fh & & \downarrow h \\
 FB & \xrightarrow{\beta} & B
 \end{array}$$

Intuitively,  $h$  meets these restrictions, if both paths from  $FA$  to  $B$  denote the same function. The thesis trades this intuition for precise statements about query programs using the language of category theory. Basic categorical vocabulary suffices, however. We perceive category theory as the vehicle not the cargo of this work.

**6** The restrictions we impose on query programs may seem rather rigid at first sight but actually they are not. The expressive power of these programs is sufficient to cover orthogonal query languages for complex value databases, like OQL or newer dialects of SQL. Everything can be reduced to a single recursive program form, the *catamorphism*, which provides all the control structure we need. At the same time, this restriction can lead to new insights into compositions of programs—and thus complex queries—due to the properties catamorphisms exhibit.

## 2 The Monad Comprehension Calculus

**7** To narrow the gap between user level query syntax and catamorphisms, we will exploit a calculus, the *monad comprehension calculus*, as a mediator between the two worlds. For now, think of monads [Mog91] as algebras that offer just the right measure of structure to interpret the constructs of a query calculus. In a nutshell, monads are algebras exhibiting exactly the structure that is needed to support the interpretation of a query calculus, the *monad comprehension calculus* [Wad90]. Built on top of the abstract monad notion, the calculus maps a variety of query constructs (*e. g.*, bulk operations, aggregates, and quantifiers) to few syntactic forms. The uniformity of the calculus facilitates the analysis and transformation, especially the normalization, of its expressions. Few but generic calculus rewriting rules suffice to implement query transformations that would otherwise require extensive rule sets. Monad comprehensions provide the query representation of choice throughout major portions of the text as they are accessible to the human eye as well as an

effective way to manipulate queries inside an optimizer. Once we remove the calculus' syntactic sugar, however, we realize that we are still operating with catamorphisms.

**8** In some sense, this text used monads in the role that sets play in the relational calculus. We consider it a feature not a bug of the monad notion that it comes with just enough internal structure that is needed to interpret a query calculus. The resulting monad comprehension calculus is poor with respect to the variety of syntactic forms it offers but this ultimately led to a discipline in query compilation that extracted the core structure inherent to a query. No obfuscation caused by syntactic sugar (of which approaches that rewrite user level syntax, *e. g.*, OQL or SQL, suffer) remained.

Being completely parametric in the monad an expression of the calculus is evaluated in, the number of different query forms we encountered was significantly reduced: the  $\mathbb{T}$ -monad comprehension  $\llbracket f \ x \ \parallel \ x \leftarrow xs \rrbracket^{\mathbb{T}}$  can describe parallel application of  $f$ , duplicate elimination, aggregation, or a quantifier ranging over  $xs$ , dependent on the actual choice of monad  $\mathbb{T}$ . This uniformity enables us to spot useful and sometimes unexpected dualities between query constructs, *e. g.*, the close connection of the class of flat join queries and the queries evaluated in the **exists** monad (*e. g.*, existential quantification).

The terseness of the calculus additionally had a positive impact on the size of the rule sets necessary to express complex query rewrites. Rewriting rules could be established by appealing to the abstract monad notion in general and then used in many instantiations.

**9** At places, the thesis rediscovers well-known query optimization knowledge using unusual paths that are more practicable to follow for an optimizer, however. At places, we can generalize and at the same time simplify solutions that have been proposed by others. The query model is abstract enough to stress the common ground of a diversity of query constructs from bulk operations to quantifiers. This makes the model an ideal target for the translation of declarative OQL-like user query languages, including recent feature additions to these languages. The monad comprehension calculus effectively supports optimization in presence of, *e. g.*, grouping, nesting, quantifiers, and aggregates in queries. The query model is specific enough to provide the necessary hints and handles to serve as an effectively manipulable representation of query execution plans—this provides a static account (*i. e.*, at query optimization time) of query runtime issues that have been traditionally tackled on the implementation level only. Finally, the model has already shown its suitability as a platform on which the rapid prototypical development of a query engine for

complex value databases is viable.

### 3 Comprehending Queries

**10** Perhaps the most principle and influential decision in solving a problem is the choice of language in which we represent both the problem and its possible solutions. Choosing the “right” language can turn the concealed or difficult into the obvious or simple. The thesis revisits a series of problems in the advanced query processing domain. In all cases, it is the aim to show how a catamorphic and monadic query language can (a) simplify, if not automate, the derivation of proposed solutions to the problem, (b) help to assess the correctness of these solutions, (c) possibly generalize the class of queries described by the problem and thus clarify the applicability of its solution.

**11** Space restrictions prohibit the display of the gory details here, but nevertheless let us revisit a small number of query processing issues to provide a flavor of what kind of problems the thesis tackles.

**12** In [SAB94], Steenhagen, Apers and Blanken analyzed a class of SQL-like queries which exhibit correlated nesting in the **where**-clause, more specifically

$$\begin{array}{l} \text{select distinct } f x \\ \quad \text{from } xs \text{ as } x \\ \quad \text{where } p x z \\ \quad \text{with } z = \left( \begin{array}{l} \text{select } g x y \\ \quad \text{from } ys \text{ as } y \\ \quad \text{where } q x y \end{array} \right) . \end{array}$$

It is the question whether queries of this class may be rewritten into *flat join queries* of the form

$$\begin{array}{l} \text{select distinct } f x \\ \quad \text{from } xs \text{ as } x, ys \text{ as } y \\ \quad \text{where } q x y \\ \quad \text{and } p' x v \\ \quad \text{with } v = g x y . \end{array}$$

Queries for which such a replacement predicate  $p'$  cannot be found have to be processed either (a) using a nested loop strategy, or (b) by grouping, ideally via a *nestjoin*. Whether we can derive a flat join query is, naturally, dependent on the nature of the yet unspecified predicate  $p$ .

The thesis approaches the problem by a reformulation of the involved queries in terms of monad comprehensions. Once this has been done, the text can characterize a structural condition on  $p$  which enables query optimizers to efficiently detect all “flat join” scenarios.

**13** Monad comprehensions can grasp queries outside the classical relational domain. The thesis gives a purely calculational proof for the correctness of a parallelizing optimization of *group queries* [CR97]

```

select f x, agg(g x)
  from xs as x
group by f x .

```

As the idea behind the optimization of these queries relies on a mix of representations (SQL syntax, query graphs, relational algebra, explicit iteration), assessing its correctness bears subtleties. Reasoning in the monad comprehension calculus, however, removes this diversity and enables an almost mechanical proof of correctness.

**14** In a compositional query language, there exist constellations of query clauses that lend themselves to more efficient—in terms of space and time complexity—evaluation algorithms than the complexity of the evaluation of its parts, *i. e.*, subqueries, leads on to assume. The SQL queries in the following class provide instances of this phenomenon:

```

select f x (agg z)
  from xs as x
with z = (
  select y
    from ys as y
  where y θ x
) .

```

In [CM95], Cluet and Moerkotte realized that an optimizer can reduce both the space and time needed to evaluate these queries by the use of so-called  $\theta$ -tables. The problem remained of how to effectively detect these profitable situations (this depends on the nature of  $agg$  as well as  $\theta$ ) and, once detected, how to deduce the necessary input parameters for the underlying  $\theta$ -table optimization.

This thesis can provide effective answers to both questions: the necessary properties of  $agg$  are naturally expressed in our query model and—as a bonus—we can clarify the applicability of the  $\theta$ -table approach (which turns out to be more general than the original work envisioned).

**15** We have found this comprehension of queries based on catamorphisms and monads to cover, simplify, and generalize many of the proposed views of database queries [GS99]. Even better, however, it enabled us to offer a terse and thus elegant account of issues in query optimization that were cumbersome or impossible to express in a way that is effectively accessible for a query optimizer.

## 4 Toolbox

**16** This work draws ideas and methods on a variety of sources, some of which are somewhat alien to the query optimization domain. The text continuously walks the fine line—if there is any—between query optimization, category theory, program transformation, type theory, and functional programming.

(a) We let *category theory* play the role that set theory has in the world of relational databases. The categorical view provides a measure of abstraction that enables important generalizations and elegant reasoning at the same time. While set-theoretic accounts of query optimization dominate the field of research by far, others have paved the way for a categorical model of queries [BNTW95, Won94].

(b) There exists an extensive theory, the *Constructive Algorithmics* [Fok92, Gra90, Bir89], on the transformation of programs that were built from a small set of combining forms. This theory understands programs as objects that are subject to calculation just like numbers in arithmetic. Core query transformations are established through calculation with programs.

(c) Type theory, especially *parametric polymorphism* and the laws it justifies for free [Wad89, WB89], constitutes another field we benefit from.

(d) Last but not least, we perceive query transformation and optimization as a *functional programming activity*. Superficially, this concerns a number of notational conventions we adopted. More deeply, note that we generate query results solely through the side-effect free construction of values from simpler constituents. In fact, we find an approach to query optimization that does otherwise hard to imagine: referential transparency is the key to painless transformational programming and equational reasoning. Functional composition is the predominant way of forming complex queries from simpler ones. Finally, when it comes to the generation of query execution plans, we establish connections to implementation techniques for lazily evaluated functional programming languages [PJ87].

## References

- [Bir89] Richard S. Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122–126, 1989.
- [BNTW95] 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.
- [CM95] Sophie Cluet and Guido Moerkotte. Efficient Evaluation of Aggregates on Bulk Types. In *Proc. of the 5th Int'l Workshop on Database Programming Languages (DBPL)*, Gubbio, Italy, September 1995.
- [CR97] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise Processing of Relational Queries. In *Proc. of the 23rd Int'l Conference on Very Large Data Bases (VLDB)*, pages 476–485, Athens, Greece, August 1997.
- [Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Enschede, 1992.
- [Gra90] Malcolm Grant. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.
- [Gru99] Torsten Grust. *Comprehending Queries*. PhD thesis, University of Konstanz, September 1999. Available for download at URL <http://www.fmi.uni-konstanz.de/~grust/thesis.pdf>.
- [GS99] Torsten Grust and Marc H. Scholl. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems*, 12(2/3):191–218, March 1999. Special Issue on Functional Approach to Intelligent Information Systems.
- [Mog91] Eugenio Moggi. Notions of Computations and Monads. *Information and Computation*, 93(1):55–92, 1991.
- [PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1987.
- [SAB94] Hennie J. Steenhagen, Peter M.G. Apers, and Henk M. Blanken. Optimization of Nested Queries in a Complex Object Model. In

- Proc. of the 4th Int'l Conference on Extending Database Technology (EDBT)*, pages 337–350, Cambridge, UK, March 1994.
- [Wad89] Philip Wadler. Theorems for Free! In *Proc. of the 4th Int'l Conference on Functional Programming and Computer Architecture (FPCA)*, London, England, September 1989.
- [Wad90] Philip Wadler. Comprehending Monads. In *Conference on Lisp and Functional Programming*, pages 61–78, June 1990.
- [WB89] Philip Wadler and Stephen Blott. How to make Ad-hoc Polymorphism less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages (POPL)*, Austin, Texas, January 1989.
- [Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, Philadelphia, August 1994.

**Acknowledgements.** My sincere thanks go to Marc Scholl, my advisor, who has somehow managed to step up onto the stage in just the right moment and I lost little time to accept his invitation to join his newly founded research group at the University of Konstanz. There have been periods where my work has led me away from the core database research field into the world of functional programming, but Marc never hesitated to encourage me to follow this alien path. Maybe that's what I value the most about him.

Throughout the five years I had many, often amazing, encounters with the members of our research community. During a visit at the University of Twente I got to know Peter Apers and his group and I'm particularly grateful that Peter accepted to review this thesis. Without the help of Maarten Fokkinga there would be more errors than you will find now. Sometimes even short exchanges of thoughts can change the way you see things. I had such experiences when I met Peter Buneman (and the participants of the FDM workshop in London, July 1997), Erik Meijer, Doaitse Swierstra, and Phil Wadler in person, and during e-mail conversation with Mitch Cherniack and Phil Trinder.

**Torsten “Teggy” Grust**, born in August 1968, enrolled to study Computing Science at the Technical University of Clausthal in 1989. In September 1994 he joined Marc H. Scholl's database research group at the University of Konstanz and completed his Ph.D. studies in September 1999. Since 1991, his primary research interest has been database query representation and optimization with a growing emphasis on related functional programming issues starting ca. 1995. At the time of writing (Summer 2000), he has been a visiting researcher at IBM's Silicon Valley Labs—the former Santa Teresa Labs—to incorporate the ideas developed in this thesis into IBM's pervasive database system DB2<sub>e</sub> (*DB2 Everywhere*).