Comprehending Queries

Torsten Grust

Dissertation der Universität Konstanz
Fakultät für Mathematik und Informatik
1999

1. Referent: Prof. Dr. Marc H. Scholl, Universität Konstanz
2. Referent: Prof. dr. Peter M.G. Apers, Universität Twente (Niederlande)

Tag der mündlichen Prüfung: 30. September 1999

Abstract

There are no compelling reasons why database-internal query representations have to be designated by *operators*. This text 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 text then takes a step from catamorphisms towards a higher-level query representation based on the categorical notion of *monads*. 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*. 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.

The text 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 this thesis to show that catamorphisms and monad comprehensions enable a comprehension of queries that is *effective* and easily *exploitable* inside a query optimizer.

Zusammenfassung (Summary in German)

Obwohl es dafür keine zwingenden Gründe gibt, setzen Anfrageoptimierer fast ausschließlich interne Anfragerepräsentationen ein, die durch *Operatoren* charakterisiert sind. In diesem Text beschreiben wir dagegen eine Welt, in der *Datentypen* das Verständnis von Anfragen bestimmen. Innerhalb dieser Welt wird ein Datentyp durch die Algebra seiner Konstruktoren beschrieben. Diese Konstruktoralgebren sind zentral. Operatoren hingegen sind zweitrangig: ein Operator ist lediglich eine Box in der ein (rekursives) Programm eingekapselt ist, welches die Berechnung von Anfrageergebnissen durch die Anwendung von Konstruktoren beschreibt. Immer wieder werden wir diese Box aufbrechen, um das Verhalten des Operators genauer zu studieren oder sein internes Programm zu transformieren. Anfrageoptimierung bedeutet dann vor allem, sich mit Programmtransformationen auseinanderzusetzen.

In einer Welt, die durch Datentypen und ihre Konstruktoralgebren charakterisiert ist, sind Anfragen Abbildungen zwischen diesen Algebren. Zwei grundlegende Beobachtungen machen diese Sicht auf Anfragen praktikabel: (a) schon die *Homomorphismen* zwischen Konstruktoralgebren sind bereits ausdrucksstark genug, um deklarative Anfragesprachen wie OQL oder neuere SQL-Dialekte zu begreifen und (b) gleichzeitig ist *eine* Programmform ausreichend, um diese Homomorphismen zu implementieren. Der disziplinierte Einsatz dieser Programmform, die *Catamorphismen*, macht die Programme zugänglich für die *Constructive Algorithmics*, eine umfassende und zugleich effektive algebraische Theorie der Programmtransformation.

Der nächste Schritt führt uns dann von Catamorphismen zu einer Anfragedarstellung, die auf dem kategoriellen Begriff der *Monade* basiert. Monaden besitzen exakt die algebraische Struktur, die für die Interpretation eines generischen Anfragekalküls, dem *Monad Comprehension Kalkül*, notwendig ist. Der Monad Comprehension Kalkül erlaubt lediglich eine geringe Anzahl syntaktischer Formen, die jedoch ein breites Spektrum von Anfragekonstrukten (mengenorientierte Operationen, Aggregation, Quantifikation) uniform abbilden können. Die Analyse und Transformation, besonders die Normalisierung, der Ausdrücke dieses Kalküls profitiert signifikant von dieser Uniformität. Wenige generische Transformationsregeln im Kalkül können Anfragetransformationen beschreiben, die sonst durch umfassende Regelmengen implementiert werden.

Der Text entdeckt bekannte Optimierungstechniken teilweise erneut und bewegt sich dabei auf ungewöhnlichen Pfaden, die von einem Optimierer jedoch effektiv nachvollzogen werden können. Mehrmals können wir etablierte

Strategien vereinfachen und generalisieren, was vor allem durch die effektiv zu analysierende Struktur ermöglicht wird, die der Monad Comprehension Kalkül Anfragen aufprägt. Der Kalkül unterstützt die Optimierung von Anfragen, die Gruppierung, verschiedene Formen von Nestung, Aggregate und Quantoren einsetzen. Obwohl abstrakte Konzepte wie Homomorphismen und Monaden die Grundlage liefern, ist diese Anfragerepräsentation spezifisch genug, um auch die eigentliche Anfrageimplementation zu unterstützen (wie etwa die Generierung von Anfrageplänen, die die Erzeugung von Zwischenresultaten vermeiden). Damit können Probleme statisch angegangen werden, die traditionell dynamisch zur Anfrageausführungszeit betrachtet wurden.

Catamorphismen und Monad Comprehension Kalkül ermöglichen eine Anfragerepräsentation, die innerhalb eines Optimierers *effektiv analysierbar und manipulierbar* ist. Dies zu zeigen, ist das eigentliche Ziel dieser Arbeit.

Acknowledgments

I have no doubt that you can accomplish nothing without a challenging and lively setting. During the last five years I was lucky enough to live and work in such an environment and—in some sense—this thesis is my humble try to acknowledge the scientific and personal support I have received throughout.

Marc Scholl, my advisor, 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. To me, Marc has been the ideal advisor. I have never faced, literally, closed doors when I was seeking for his guidance or help. At the same time he gave me all the freedom you need to get your bearing as a young researcher. 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. I have found the discussions with him, Annita Wilschut, and David Spelt especially exciting. Without the help of Maarten Fokkinga there would be more errors than you see here 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.

I often felt that there was exactly the right mixture of people around in the Konstanz Computer Science department. The students (especially Andres Löh) and my colleagues proved to be constant sources of useful feedback and good humor. Ulrik Brandes and Dieter Gluche, my fellow contemporaries in the department, never failed to be sympathetic towards my matters. (Probably that's only possible if you share the same goals as well as uncounted rounds of beer.)

Finally, I owe thanks to Brita. Somehow she reliably managed to distract me when there was not much more on my mind than banana $/\!/$ brackets.

Contents

Chapter 1

Prelude

1 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.

2 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.

3 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.

Let F denote a function that, given a type A as argument, encodes the structure of terms (expressions) that construct values of type A. A constructor algebra α then is nothing but a mapping from $\mathsf{F}A$ to A, in symbols $\alpha : \mathsf{F}A \to A$. Consider a second algebra of identical structure $\beta : \mathsf{F}B \to B$. If h denotes a query, in our model, it is a mapping between values, say $h : A \to B$. The above restriction, however, says that the program for h may not be arbitrary but has to respect the term structure F. Such a structure-preserving computation will be denoted $\mathsf{F}h$ in this text. To get a bearing on this process, the overall picture of the involved mappings is

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\alpha} & A \\ {\scriptstyle \mathsf{F}h}\downarrow & & \downarrow{\scriptstyle h} \\ \mathsf{F}B & \xrightarrow[\beta]{} & B \end{array}$$

Intuitively, h meets the restrictions, if both paths from $\mathsf{F}A$ to B denote the

same function. We will trade 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 text.

4 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.

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 as algebras that offer just the right measure of structure to interpret the constructs of a query calculus. Monad comprehensions will be our query representation of choice throughout major portions of this 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.

5 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 [20, 114, 124].

(b) There exists an extensive theory, the *Constructive Algorithmics* [7, 8, 9, 42, 57, 67, 84], 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. We will establish core query transformations through calculation with programs.

(c) Type theory, especially *parametric polymorphism* and the laws it justifies for free [116, 122], constitutes another field we will 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 will be the predominant way of forming complex queries from simpler ones. Finally, when it comes to the generation of query execution plans, we will establish connections to implementation techniques for lazily evaluated functional programming languages [18, 51, 92].

6 The following exploits this toolbox to develop a comprehension of queries that is geared to be *effective* and easily *exploitable* inside a database query optimizer.

At places, we will rediscover well-known query optimization knowledge using unusual paths that are more practicable to follow for an optimizer, however (Chapter 4). At places, we can generalize and at the same time simplify solutions that have been proposed by others (Chapter 5). 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 (Chapter 3). 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 (Chapter 6).

To get us going, Chapter 2 lays the categorical foundation upon everything else will rest.[1] You feel better if you know where that clatter actually comes from.

[1] To quote Philip Wadler, "*No knowledge of category theory is assumed.*" [117]

Chapter 2

Categorical Datatypes and Monads

2.1 A Datatype-based Query Representation

7 *Datatypes* lie at the very heart of the query representation used throughout this text. The language we are going to develop in this and the upcoming chapter is designed around types, *not* operations. All datatypes which we will encounter in the sequel themselves induce a minimal set of functions which allow for the construction of arbitrary values of that type. Exactly these *constructors* (and their functional composition) will form the initial query language we will propose here. Once we have thus marked the start of our path, we will then pave our way towards a query representation that will be more suitable for particular stages of the query compilation process as well as the human mind and eye. We will often see, however, that query optimization can benefit from working with this rather fine granulated view of datatypes.

This does *not* mean that an actual implementation of our representation is forced to always operate at this fine level of granularity. It rather provides us with handles to manipulate small granules during query translation and optimization. Before a query is shipped to the underlying query engine for execution it is then our responsibility to identify those parts of the query expression which may be executed in a *bulk-oriented* fashion.

Note that there are *no mutators* defined for the different types. Query evaluation in this framework will be solely concerned with the construction of instances of a type from simpler constituents. The language will thus be *referentially transparent* or, put another way, *purely functional*. This feature, in turn, will smooth the way for query optimization based on equational reasoning ("*replace equals by equals.*")

As query representation (not implementation) is the focus of this chapter

we view datatypes as being *abstract*. We will devise an interface—which will consist of the list of constructors and, in some cases, certain *equations* that have to be obeyed—for the types and leave their implementation for later. Any datatype providing a compatible interface may be incorporated into language without significant effort.

8 The instances of a type and its constructors will be modeled as an *algebra* for that type. There may be more than one algebra for a particular type. Our presentation of the material will rely on the tools of *category theory*. Category theory provides a succinct notation for the concepts involved and seems to supply just the right measure of abstraction that we strive for in this chapter. Moreover, the query language's core operators will turn out to be familiar categorical concepts. At a first sight the formal machinery might seem awkward, but important questions like the guaranteed existence or uniqueness of the operators so defined are answered for free by the underlying theory. Categorical insights will prove useful in all query translation phases, notably during the generation of query execution plans (see Chapter 6). Following ideas of Fokkinga [42, 43], we will not only express the structural aspects of the constructor algebras using category theory but also the equations which certain types are expected to fulfill. This will save us from introducing the otherwise necessary machinery of signatures for algebras, syntax for terms, as well as variables and their (scoped) bindings.

The discussion in this chapter proceeds as follows. Right after we have provided the categorical view of algebras we will examine *catamorphisms*, functions whose recursion pattern mimics that of its source type. Despite their simplicity, catamorphisms provide the core concept upon which almost all of the query language will be built. One can prove, given few prerequisites, that compositions of catamorphisms collapse into a single catamorphism, which will later justify efficient optimizations. We will then generalize our observations to cover polymorphic type constructors (like $\mathsf{bag}\,\alpha$, α denoting an arbitrary type) as these are basic building blocks of type systems for query languages for complex collection types. Being so far, the notion of *equation* is formalized which will make level for the categorical representation of datatypes with commutative and/or idempotent constructor functions like bag and set. We already include a step towards a more high-level calculus-like query representation, *monad comprehensions*, which is shown to be quite easily definable on top of the algebraic datatypes presented so far.

The material presented here is not original but the arranging is unique in the sense that it is geared towards the problem of the formalization of datatypes with equations for database query languages. This is especially

true for the derivation of multi-monad comprehensions. We refer to [10, 42, 43, 84, 117] for an extensive categorical treatment of the ideas developed in the following paragraphs.

9 To let the reader gain familiarity with the involved notation and style of reasoning, we will provide more detailed lines of arguments in this chapter than in the rest of the thesis. Equational proofs (also called *calculations* or *rewrites* in the sequel) use the style adopted by the *Squiggol* [7, 11] and *Constructive Algorithmics* [7, 42, 57, 67, 84] communities because it allows for the convenient annotation of a sequence of replacements of equals by equals or similar connected chains of reasoning. In [3], Backhouse convincingly discusses the merits and benefits of this notation.

An equation display like (let f, g, h denote functions)

$$\begin{array}{ll} & (f \cdot g) = h \\ \equiv & \quad \{\, f \cdot g = h \,\} \\ & h = h \\ \equiv & \quad \{\, \text{reflexivity of } = \,\} \\ & \mathit{true}\ , \end{array}$$

represents the derivation of the last term from the first by step-wise replacement of equals by equals. The intermediate results are given. Justifications for the replacements are shown explicitly in curly braces $\{\cdot\}$. You may have noted that the above calculation is carried out at the function level and that no function is ever applied to a particular argument. Such calculations are called *point-free* (in contrast to *point-wise*). We shall mainly use point-free reasoning throughout this text as the introduction of a dummy function argument (x say) is not needed due to the principle of extensionality and merely clutters the display of the calculation.

2.2 Basic Category Theory

10 Category theory. A few basic concepts of category theory are sufficient to master the theory of datatypes we will use as the basis to build our query language on. These are: *category* (and the inherently related *object*, *morphism*, *target*, *source*), *initiality*, *isomorphism*, *functor*, *product*, and *sum*. Later on we will meet *natural transformations* and *monads*. We give a short introduction to all of these concepts before we delve into categorical datatypes. This is not an introduction to category theory in its own right. Our perception of the material is heavily biased towards the goal we try to

reach: a concise and formal yet sufficiently abstract framework in which we can talk about polymorphic (collection) type constructors. The theory does even better: it will also provide us with fundamental query operators as these are induced by the types and their algebras themselves.

Introductory texts on category theory may be found in [2, 10, 42, 63, 64, 79, 98, 103, 104]. Among these texts, Bird and de Moor [10] put particular emphasis on the exploration of categorical datatypes.

11 Category. A *category* is a system of *objects* and *morphisms* (also called *arrows*). Fix a category $\mathbf{C}$ (categories will be written using a **bold** font). Every morphism f in $\mathbf{C}$ uniquely determines two $\mathbf{C}$-objects, its *source* ($src\ f$) and *target* ($tgt\ f$). If $src\ f = A$ and $tgt\ f = B$ we write $f : A \to_{\mathbf{C}} B$, $A \to B$ is called the *type* of f in $\mathbf{C}$. If the category $\mathbf{C}$ is clear from the context we declare it to be the default category and write $f : A \to B$ instead.

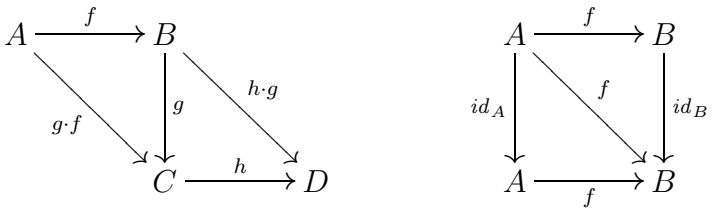
Two $\mathbf{C}$-morphisms $f : A \to B$ and $g : B \to C$ may be *composed* to obtain the morphism $g \cdot f$ of type $A \to C$ in $\mathbf{C}$ (*closure*). Composition is associative, *i. e.*, for morphisms $f : A \to B$, $g : B \to C$, and $h : C \to D$ we have

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f \ .$$

Each object A comes equipped with an *identity morphism* $id_A : A \to A$ which acts as a neutral element with respect to composition (let $f : A \to B$):

$$f \cdot id_A = f = id_B \cdot f \ .$$

It is sometimes insightful to display objects and morphisms as *diagrams*, *i. e.*, directed graphs having their nodes and edges labeled by objects and morphisms, respectively. The diagram *commutes*, if any two paths (compositions of morphisms) connecting the same pair of objects denote the same morphism. Associativity of composition and neutrality of identity morphisms may thus be depicted as two commutative diagrams:



While we will use diagrams to illustrate the typing of morphisms, we shall rarely employ them to conduct proofs. Proofs are carried out in the calculational style introduced in Paragraph 9.

In what follows we fix the default category to be the category **Set**. In **Set**, objects denote sets while a morphism f is to be interpreted as a total function with domain $src\, f$ and range (codomain) $tgt\, f$. If we interpret composition of morphisms as function composition and let id_A denote the identity function $\lambda x.x$ on A, we obey the category axioms. The type of a morphism f may then be understood as the type of a function f in the sense of typed functional programming languages. This is exactly the interpretation we will need during our discussion of datatypes.

12 Initiality. An object *1* in category $\mathbf{C}$ is *the initial object*, if $\mathbf{C}$ contains exactly one morphism of type $1 \to A$ for any object A of $\mathbf{C}$. *1* is uniquely determined *up to isomorphism*: let $1'$ be another initial object. Then there are morphisms $f : 1 \to 1'$ (by initiality of *1*) and $g : 1' \to 1$ (by initiality of $1'$). Since the identity morphisms $id_1 : 1 \to 1$ and $id_{1'} : 1' \to 1'$ are the only arrows of these types, we can conclude that $g \cdot f = id_1$ and $f \cdot g = id_{1'}$. These two equations exactly establish the categorical notion of *1* and $1'$ being isomorphic. The initial object in **Set** is $\varnothing$.

In due course, the initiality of an algebra for a datatype will assert the existence as well as uniqueness of the fundamental operators for that datatype.

13 Functor. A *functor* $\mathsf{F} : \mathbf{C} \to \mathbf{D}$ is a mapping from category $\mathbf{C}$ to category $\mathbf{D}$. This implies that F specifies an object as well as a morphism mapping. Each object A of $\mathbf{C}$ is mapped to its image $\mathsf{F}A$ in $\mathbf{D}$. The morphism mapping "preserves" arrows: if $f : A \to_{\mathbf{C}} B$ then $\mathsf{F}f : \mathsf{F}A \to_{\mathbf{D}} \mathsf{F}B$. Additionally, F is required to preserve identities and to distribute through composition:

$$\mathsf{F}id_A = id_{\mathsf{F}A} \qquad \text{and} \qquad \mathsf{F}(g \cdot f) = \mathsf{F}g \cdot \mathsf{F}f \ . \qquad \text{FUNCTOR}$$

Note that if $\mathsf{G} : \mathbf{D} \to \mathbf{E}$ is another functor, then the composition $\mathsf{GF} : \mathbf{C} \to \mathbf{E}$, with $\mathsf{GF}A$ defined to mean $\mathsf{G}(\mathsf{F}A)$ (likewise for the morphism mapping), is a functor as is the identity functor $\mathsf{Id} : \mathbf{C} \to \mathbf{C}$ with $\mathsf{Id}A = A$ and $\mathsf{Id}f = f$. (Categories and functors are the objects and morphisms in category **Cat** respectively.)

Which role do functors play in the context of categorical datatypes? A functor $\mathsf{F} : \mathbf{Set} \to \mathbf{Set}$ (also called an *endofunctor* in **Set**) can perfectly model a *type constructor* or, alternatively, *type former*. Functional programming languages like Haskell or ML provide a type constructor list that lifts any type A to the type of lists of A, namely $\mathsf{list}A$. This is the object mapping part. If we take $\mathsf{list}f = map\ f$ (*map* being the standard higher-order function

that applies a function to all elements of a list) we obtain a valid morphism mapping, because

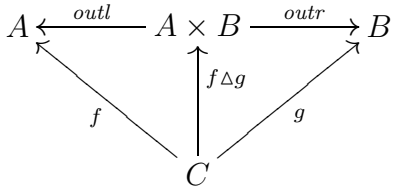
$$f : A \to B \quad \Rightarrow \quad map\, f : \mathsf{list} A \to \mathsf{list} B$$

and

$$map\, id_A = id_{\mathsf{list} A} \qquad \text{and} \qquad map\,(f \cdot g) = map\, f \cdot map\, g\ .$$

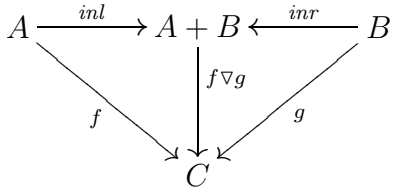
This indeed turns the list type constructor with *map* into an endofunctor $\mathsf{list} : \mathbf{Set} \to \mathbf{Set}$. The important thing to observe here is the close connection between the type constructor and its associated morphism mapping. We will formally justify this intuition soon.

14 Product and sum. Given two objects A and B of $\mathbf{C}$ we can construct their *product object* $A \times B$ with accompanying left and right *projections* $outl : A \times B \to A$ and $outr : A \times B \to B$ if we can find for all morphisms $f : C \to A$ and $g : C \to B$ a morphism $f \vartriangle g$ ("f *split* g") in $\mathbf{C}$ so that the following diagram commutes:



The diagram asserts the *cancellation* properties for products $outl \cdot (f \vartriangle g) = f$ and $outr \cdot (f \vartriangle g) = g$.

The notion of categorical *sum* or *coproduct* is completely dual. The sum of A and B, denoted $A + B$, and its *injections* $inl : A \to A + B$ and $inr : B \to A + B$ can be constructed if we are able to find for all morphisms $f : A \to C$ and $g : B \to C$ a morphism $f \triangledown g$ ("f *junc* g") in $\mathbf{C}$ so that the diagram



commutes. It displays the cancellation for sums: $(f \triangledown g) \cdot inl = f$ and $(f \triangledown g) \cdot inr = g$.

It may be helpful to perceive $\vartriangle$ and $\triangledown$ as mnemonics for the arrow heads of *split* and *junc* in the corresponding diagrams above.

If we can build the product (sum) for any two objects, **C** is said to *have products (sums)*. Category **Set** has products and sums: define $A \times B$ to be the cartesian product of pairs (x, y) with $x \in A, y \in B$ and $A + B$ to be the disjoint sum of sets A and B. The apparent projections and injections then are $outl\,(x, y) = x$, $outr\,(x, y) = y$ and $inl\,x = Lx$, $inr\,y = Ry$ respectively (L and R being tags to distinguish the origin set of the injected elements). To complete our definition of product and sum in **Set** finally define

$$f \vartriangle g = \lambda x.(f\,x, g\,x) \qquad \text{and} \qquad f \triangledown g = \lambda x.case\,x\,of\;\begin{array}{l} Ly \to f\,y \\ Ry \to g\,y\;, \end{array}$$

so that $\triangledown$ essentially implements case selection.

15 Bifunctors and polynomial functors. Fix **C** to be category with products. In **C**, we can lift the product former $\times$ to be applicable to morphisms, too: for morphisms $f : A \to B$ and $g : C \to D$, define

$$f \times g = (f \cdot outl) \vartriangle (g \cdot outr) : A \times C \to B \times D\;.$$

With the definition of $\vartriangle$ from the previous paragraph, we derive the meaning of $f \times g$ in **Set** as follows:

$$\begin{array}{ll} & f \times g \\ = & \quad \{\,\text{definition of } \times \text{ on morphisms}\,\} \\ & (f \cdot outl) \vartriangle (g \cdot outr) \\ = & \quad \{\,\text{definition of } \vartriangle \text{ in } \mathbf{Set}\,\} \\ & \lambda z.((f \cdot outl)\,z, (g \cdot outr)\,z) \\ = & \quad \{\,\text{set } z = (x, y)\,\} \\ & \lambda(x, y).((f \cdot outl)\,(x, y), (g \cdot outr)\,(x, y)) \\ = & \quad \{\,outl \text{ and } outr \text{ in } \mathbf{Set}\,\} \\ & \lambda(x, y).(f\,x, g\,y)\;. \end{array}$$

For sums we can argue dually and define $+$ on morphisms $f : A \to B$, $g : C \to D$ to mean

$$f + g = (inl \cdot f) \triangledown (inr \cdot g) : A + C \to B + D$$

A similar calculation like the one exercised above reveals the meaning of the thus lifted $+$ in **Set**:

$$f + g = \lambda x.case\,x\,of\;\begin{array}{l} Ly \to L(f\,y) \\ Ry \to R(g\,y)\;, \end{array}$$

i. e., $f + g$ uses the tags to scrutinize its argument for application of f or g and finally injects its argument into the result by restoring the original tag.

Now that $\times$ and $+$ act on objects *and* morphisms one might suspect (or wish) that they behave like functors. Indeed, defined like above, $\times$ and $+$ are *bifunctors*, *i. e.*, functors of two arguments: given the morphisms $f : A \to_{\mathbf{C}} B$, $g : B \to_{\mathbf{C}} C$ and $h : D \to_{\mathbf{D}} E$, $j : E \to_{\mathbf{D}} F$, $\odot$ is a bifunctor if $f \odot h : A \odot D \to_{\mathbf{E}} B \odot E$ and

$$id_A \odot id_D = id_{A \odot D} \qquad \text{and} \qquad (g \cdot f) \odot (j \cdot h) = (g \odot j) \cdot (f \odot h) \; .$$

We can alternatively view $\odot$ as a regular functor $\odot : \mathbf{C} \times \mathbf{D} \to \mathbf{E}$, where $\mathbf{C} \times \mathbf{D}$ denotes the *product category* of categories $\mathbf{C}$ and $\mathbf{D}$. The objects of $\mathbf{C} \times \mathbf{D}$ are all pairs (C, D) with C being an object of $\mathbf{C}$ and D being an object of $\mathbf{D}$. Morphisms (and likewise their composition) are defined coordinate-wise in $\mathbf{C} \times \mathbf{D}$: if $f : C_1 \to_{\mathbf{C}} C_2$ and $g : D_1 \to_{\mathbf{D}} D_2$ then $(f, g) : (C_1, D_1) \to_{\mathbf{C} \times \mathbf{D}} (C_2, D_2)$.

A bifunctor can be used to combine two given functors F and G to give a functor $\mathsf{F} \odot \mathsf{G}$ defined to mean $(\mathsf{F} \odot \mathsf{G})A = \mathsf{F}A \odot \mathsf{G}A$ (likewise for morphisms).

We will solely use $\times$ and $+$ in place of $\odot$. As usual, $\times$ binds stronger than $+$ so that $\mathsf{F} + \mathsf{G} \times \mathsf{H}$ parses as $\mathsf{F} + (\mathsf{G} \times \mathsf{H})$. Functor composition binds weakest.

Functors constructed from the identity functor Id, *constant functors* K_A (with $\mathsf{K}_A B = A$ and $\mathsf{K}_A f = id_A$), bifunctors $\times$ and $+$, and functor composition only are the so-called *polynomial functors*. Polynomial endofunctors in **Set** form the basis of our system of type constructors: they provide *tuples* (built by $\times$), *unions* (built by $+$) and simple or *atomic* types (built by Id and K respectively, *i. e.*, objects in **Set**).

2.3 Categorical Collection Type Constructors

Now that we have got the necessary categorical machinery at our disposal, we are ready to complete our type system by the introduction of collection type constructors.

16 We have already mentioned in Paragraph 7 that our view of datatypes will be non-monolithic, *i. e.*, the actual construction of a complex value of a particular type will be explicitly visible. To represent any such complex value, we need to assemble it from its constituents (which might be atomic or complex themselves) by means of its type's constructor functions.

For collection types, we will adopt the *insert representation* [111] throughout this text. In such a representation, it is sufficient for a collection type

constructor T[1] to come equipped with just two constructor functions[2]

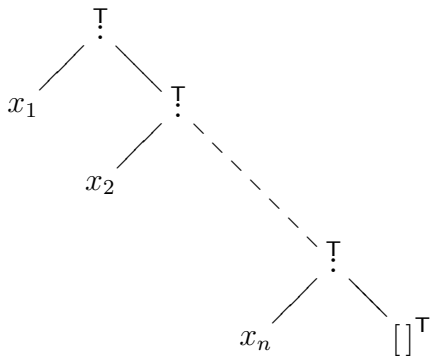
- nil^{T} is a constant function representing an empty T-collection, while
- $cons^{\mathsf{T}}(x, xs)$ inserts x into the T-collection xs.

Issues of polymorphism will be addressed soon. For now, you may assume x to be of some fixed *element type* E. The object in **Set** representing (or: "*the type of*") T-collections of E-typed elements will be $\mathsf{T}E$.

The economical use of constructors is one of the major advantages of the insert representation. See Section 2.10 for a discussion of alternative representations. Any finite T-collection value may be built by a finite iterative application of the insertion constructor $cons^{\mathsf{T}}$ to the seed nil^{T}. Put differently: the type of T-collections is *generated* by nil^{T} and $cons^{\mathsf{T}}$ (here and in what follows, we will abbreviate nil^{T} by $[\,]^{\mathsf{T}}$ and $cons^{\mathsf{T}}$ by the right-associative infix operator $\overset{\mathsf{T}}{:}$ whenever it renders expressions more readable):

$$[x_1, x_2, \ldots, x_n]^{\mathsf{T}} \quad = \quad x_1 \overset{\mathsf{T}}{:} (x_2 \overset{\mathsf{T}}{:} (\cdots (x_n \overset{\mathsf{T}}{:} [\,]^{\mathsf{T}}) \cdots)) \quad = \quad x_1 \overset{\mathsf{T}}{:} x_2 \overset{\mathsf{T}}{:} \cdots \overset{\mathsf{T}}{:} [\,]^{\mathsf{T}},$$

The application of this composition of nil^{T} and $cons^{\mathsf{T}}$ to elements x_i is sometimes called the *spine* of $[x_1, \ldots, x_n]^{\mathsf{T}}$, mnemonically depicted as



The above uses $[x_1, x_2, \ldots, x_n]^{\mathsf{T}}$ as a handy shorthand notation for a T-collection, but we should keep in mind that we are actually constructing the value by forming its spine.

In the sequel we will mainly use set, bag, and list as collection type constructors and the following paragraphs will introduce the respective functors

[1] Here we use the sans-serif font that has been reserved to denote functors which will be justified in Paragraph 34 where we establish T's functoriality.

[2] We realize that the names *nil* ("*nothing in list*") and *cons* ("*construct*") have already been taken by the functional programming community to denote the constructor functions of the list type constructor. To emphasize various useful analogies, however, we refrained from inventing new constructor names for collection formers other than list.

for these. For set as well as bag, the same value may be denoted by different spines. As the obvious interpretations for $cons^{\mathsf{set}}(x, xs)$ and nil^{set} are $\{x\} \cup xs$ and $\varnothing$, we have that $cons^{\mathsf{set}}$ is *left-commutative* and *left-idempotent*, *i. e.*,

$$y \mathbin{:^{\mathsf{set}}} (x \mathbin{:^{\mathsf{set}}} xs) \quad = \quad x \mathbin{:^{\mathsf{set}}} (y \mathbin{:^{\mathsf{set}}} xs) \qquad \text{and} \qquad x \mathbin{:^{\mathsf{set}}} (x \mathbin{:^{\mathsf{set}}} xs) \quad = \quad x \mathbin{:^{\mathsf{set}}} xs \ .$$

A bag can contain an element x multiple times but does not preserve the order of insertions so that $cons^{\mathsf{bag}}$ is left-commutative only. Constructor $cons^{\mathsf{list}}$ is neither. Section 2.7 will formalize the equalities of spines by the notion of *datatype equations*.

17 Fix an element type E and recall that we started out with the aim to model a collection type constructor T as an algebra. The *carrier* set of this algebra will clearly be $\mathsf{T}E$ as already noted in the previous paragraph. The only operations in this algebra will be the *constructor* functions nil^{T} and $cons^{\mathsf{T}}$. In **Set**, these are nothing else but the ordinary arrows[3] $nil^{\mathsf{T}} : 1 \to \mathsf{T}E$ and $cons^{\mathsf{T}} : E \times \mathsf{T}E \to \mathsf{T}E$. As both constructor functions share carrier $\mathsf{T}E$ as their target object, we can alternatively specify their types by stating:

$$nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}} : 1 + E \times \mathsf{T}E \to \mathsf{T}E \ .$$

The categorical representation of the thus induced algebra is remarkably concise.

18 Definition. Let F denote an endofunctor in category **C**. An **F-algebra** is an arrow $\alpha = \mathsf{F}A \to A$. The object A of **C** is called the **carrier** of α. //

19 $\mathsf{T}E$ collections in insert representation are modeled by the algebra $nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}} : \mathsf{F}(\mathsf{T}E) \to \mathsf{T}E$, with F being the polynomial endofunctor $\mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id}$ in **Set**. Note that

$$\begin{array}{ll} & \mathsf{F}(\mathsf{T}E) \\ = & \quad \{ \text{ definition of } \mathsf{F} \ \} \\ & (\mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id})(\mathsf{T}E) \\ = & \quad \{ \text{ bifunctors } + \text{ and } \times \ \} \\ & \mathsf{K}_1(\mathsf{T}E) + \mathsf{K}_E(\mathsf{T}E) \times \mathsf{Id}(\mathsf{T}E) \end{array}$$

[3] An arrow $f : 1 \to A$ is also called an *element* of object A. In **Set**, such arrows represent constant functions and for convenience we shall sometimes identify the arrow and its target object. In type theory, *1* is often called the *unit* type consisting of the only element () (also referred to as *unit*).

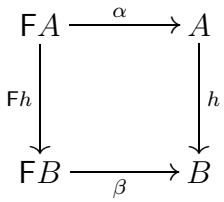
$$
\begin{array}{ll}
= & \{\,\text{functors } \mathsf{K} \text{ and } \mathsf{Id}\,\} \\
& 1 + E \times \mathsf{T}E\ ,
\end{array}
$$

and analogously $\mathsf{F}f = id_1 + id_E \times f$.

20 Definition. Pin down two F-algebras $\alpha = \mathsf{F}A \to A$ and $\beta = \mathsf{F}B \to B$. An **$\mathsf{F}$-homomorphism** from α to β is an arrow $h : A \to B$ such that

$$h \cdot \alpha = \beta \cdot \mathsf{F}h\ , \qquad \textsc{Hom}$$

which we can equivalently render as the commuting diagram



As an important note, observe that for any object A its identity arrow $id : A \to A$ is an F-homomorphism. For any two F-homomorphisms f and g, we have that $f \cdot g$ is an F-homomorphism as well. This already provides everything that is needed to form a category (*cf.* Paragraph 11): we define **Alg**(F) to denote the category whose objects are F-algebras. The arrows of **Alg**(F) are the F-homomorphisms. //

21 A simple calculation reveals that property Hom of h indeed coincides with the well-known concept of h being a homomorphism between algebras: let $\mathsf{F} = \mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id}$ once more denote the functor encoding the constructors of a type in insert representation introduced in Paragraph 19. Furthermore fix the F-algebras $\alpha = e \triangledown \otimes : \mathsf{F}A \to A$ and $\beta = z \triangledown \oplus : \mathsf{F}B \to B$ (see [10, p. 47] for a proof that any algebra for functor F with carrier A has the form $e \triangledown \otimes$ with arrows $e : 1 \to A$ and $\otimes : E \times A \to A$). The calculation is as follows:

$$
\begin{array}{ll}
& h \cdot \alpha = \beta \cdot \mathsf{F}h \\
\equiv & \quad \{\,\alpha = e \triangledown \otimes,\ \beta = z \triangledown \oplus\,\} \\
& h \cdot (e \triangledown \otimes) = (z \triangledown \oplus) \cdot \mathsf{F}h \\
\equiv & \quad \{\,\text{sum: } h \cdot (f \triangledown g) = (h \cdot f) \triangledown (h \cdot g)\,\} \\
& (h \cdot e) \triangledown (h \cdot \otimes) = (z \triangledown \oplus) \cdot \mathsf{F}h \\
\equiv & \quad \{\,\text{unfold } \mathsf{F}, \text{polynomial functor}\,\} \\
& (h \cdot e) \triangledown (h \cdot \otimes) = (z \triangledown \oplus) \cdot (id_1 + id_E \times h)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\,\text{sum: } (h \triangledown j) \cdot (f + g) = (h \cdot f) \triangledown (j \cdot g)\,\} \\
& (h \cdot e) \triangledown (h \cdot \otimes) = (z \cdot id_1) \triangledown (\oplus \cdot (id_E \times h)) \\
\equiv & \{\,\text{identity arrow}\,\} \\
& (h \cdot e) \triangledown (h \cdot \otimes) = z \triangledown (\oplus \cdot (id_E \times h)) \\
\equiv & \{\,\text{cancellation}\,\} \\
& \quad h \cdot e = z \\
& \wedge \quad h \cdot \otimes = \oplus \cdot (id_E \times h) \\
\equiv & \{\,\text{point-wise reasoning}\,\} \\
& \quad h\, e = z \\
& \wedge \quad h(x \otimes xs) = x \oplus (h\, xs)
\end{array}
$$

The two last equations constitute the familiar statement of h being a homomorphism from α to β.

2.4 Catamorphisms

22 Up to now we have informally called nil^{T} and $cons^{\mathsf{T}}$ the *constructors* for a datatype former T. The intuition behind the constructor notion is that

- every element of the carrier $\mathsf{T}E$ of $\alpha = nil \triangledown cons : \mathsf{F}(\mathsf{T}E) \to \mathsf{T}E$ can be constructed using applications of nil^{T} and $cons^{\mathsf{T}}$ to values in E only. No other elements are in $\mathsf{T}E$, *i. e.*, there is no *"junk."*
- Secondly, given that there are no equations yet, two elements of $\mathsf{T}E$ are identical only if their spines are exactly the same. Jumping ahead to Section 2.7 in which we introduce equations, this means that two elements are identical only if this can be derived from the equations of the datatype. This prevents us from accidentally *"confusing"* two elements of $\mathsf{T}E$ with each other.

Let us shed more categorical light on this here.

Assume for now that **Alg**(F) has an initial object, say $\tau : \mathsf{F}T \to T$. Actually this is no strong assumption at all in our setting as for polynomial functors F, the existence of an initial object in **Alg**(F) is guaranteed (see [81] for a proof of this statement).

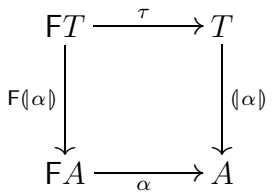
23 Definition. Let $\tau : \mathsf{F}T \to T$ be the initial and $\alpha : \mathsf{F}A \to A$ an arbitrary object of **Alg**(F). An F-homomorphism of type $T \to A$ (which is, due to the initiality of τ, guaranteed to exist and to be unique, *cf.* Paragraph 12) from

τ to α is a so-called **F-catamorphism**. These special homomorphisms will accompany us through the entire text, so we will adopt the succinct notation $(\!|\alpha|\!)_{\mathsf{F}}$ for the catamorphism from τ to α.[4] If the functor is evident from the context we take the freedom to omit the subscript.

The uniqueness of $(\!|\alpha|\!)$ indeed is its prominent property. Any other F-homomorphism h from τ to α clearly has to be identical with $(\!|\alpha|\!)$:

$$h \cdot \tau = \alpha \cdot \mathsf{F}h \qquad \equiv \qquad h = (\!|\alpha|\!)\ . \tag{CATA}$$

The morphisms involved in law CATA are shown in the commuting diagram below. A (unique) homomorphic translation between algebras like the one depicted here will form the basis of our query language.



//

24 Initial algebras. What does the initiality of τ buy us? First, Paragraph 12 introduced initiality as a concept that determined the initial object in a category up to isomorphism. Of course the same is true in category **Alg**(F): suppose we can identify another initial algebra τ', we know that τ and τ' are isomorphic and therefore *abstractly the same*. But this notion of abstraction perfectly coincides with the independence of representation of an abstract datatype. Abstraction is inherent to initiality. In what follows, we will use the term *abstract datatype* to refer to the isomorphism class of the initial algebra τ.

Second, as the next paragraph will show, τ is an isomorphism from $\mathsf{F}T$ (the type of expressions built from operators in τ and elements of the carrier T) to T. As τ is a morphism in **Set**, we thus know that τ is a bijective mapping from $\mathsf{F}T$ to T:

- τ is surjective, so any element in T is represented by at least one expression in $\mathsf{F}T$. There is no *"junk."*
- τ is injective, so we may conclude that the elements of T have a unique spine representation in $\mathsf{F}T$. Consequently there is no *"confusion."*

[4] The "*banana brackets*" $(\!|\cdot|\!)$—presumably introduced by Lambert Meertens—are the established catamorphism symbol in the Squiggol community.

Most importantly however, initiality justifies the principle of defining functions on T by *structural recursion*. As each element e of T is constructed by its unique spine in $\mathsf{F}T$, *i. e.*, a composition of the constructors of τ, we can define a function h by case selection on the constructor whose application is outermost in the spine of e. Each case branch may then descend further down the spine and recursively apply h.

We can also see this as follows: as $\tau : \mathsf{F}T \to T$ is an isomorphism, we are assured of the existence of an "inverse" isomorphism $\bar{\tau} : T \to \mathsf{F}T$ with $\tau \cdot \bar{\tau} = id_T$ and $\bar{\tau} \cdot \tau = id_{\mathsf{F}T}$. Observe that the type of $\bar{\tau}$ suggests to interpret this arrow as a *destructor* or *parser* mapping values in T to the expression (spine) in $\mathsf{F}T$ that constructs this value. This gives us more insight into the nature of the action of a catamorphism $h = (\!|\alpha|\!)$ for some F-algebra α:

$$
\begin{aligned}
& h = (\!|\alpha|\!) \\
\equiv \quad & \{\, \textsc{Cata} \,\} \\
& (\!|\alpha|\!) \cdot \tau = \alpha \cdot \mathsf{F}(\!|\alpha|\!) \\
\equiv \quad & \{\, \tau \cdot \bar{\tau} = id_T \,\} \\
& (\!|\alpha|\!) = \alpha \cdot \mathsf{F}(\!|\alpha|\!) \cdot \bar{\tau} \ ,
\end{aligned}
$$

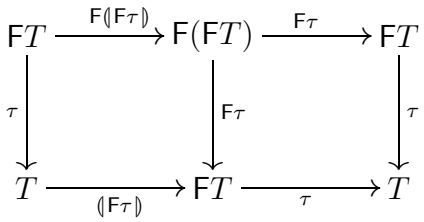
i. e., h forms the spine of its argument, recursively applies h by walking down the spine, and finally builds the result by interpreting the spine in algebra α.

The following lemma guarantees the existence of $\bar{\tau}$ and justifies the above.

25 Lambek's Lemma [71]. Let $\tau : \mathsf{F}T \to T$ be the initial algebra of endofunctor F. Then τ is an isomorphism, *i. e.*, there exists an arrow $\bar{\tau} : T \to \mathsf{F}T$ such that $\tau \cdot \bar{\tau} = \bar{\tau} \cdot \tau = id$.

Proof. Applying the endofunctor F to the algebra τ yields the F-algebra $\mathsf{F}\tau : \mathsf{F}(\mathsf{F}T) \to \mathsf{F}T$. The unique mapping from τ to $\mathsf{F}\tau$—which is guaranteed to exist due to the initiality of τ—is $(\!|\mathsf{F}\tau|\!) : T \to \mathsf{F}T$. We now argue that $\bar{\tau} = (\!|\mathsf{F}\tau|\!)$.

In the diagram below, the left square merely states that the catamorphism $(\!|\mathsf{F}\tau|\!)$ is an F-homomorphism which is evident. Both paths in the right square denote the very same arrow composition so that it trivially commutes. "Pasting" commutative diagrams yields a commutative diagram.



The diagram reveals $\tau \cdot (\!|\mathsf{F}\tau|\!)$ to be a mapping from the algebra τ to τ itself, so we have $\tau \cdot (\!|\mathsf{F}\tau|\!) = id$ by initiality. All that remains to be done now is to carefully look at the left square from which we get

$$
\begin{array}{ll}
 & (\!|\mathsf{F}\tau|\!) \cdot \tau = \mathsf{F}\tau \cdot \mathsf{F}(\!|\mathsf{F}\tau|\!) \\
\equiv & \quad \{\, \text{functoriality of } \mathsf{F} \,\} \\
 & (\!|\mathsf{F}\tau|\!) \cdot \tau = \mathsf{F}(\tau \cdot (\!|\mathsf{F}\tau|\!)) \\
\equiv & \quad \{\, \tau \cdot (\!|\mathsf{F}\tau|\!) = id \text{ established above} \,\} \\
 & (\!|\mathsf{F}\tau|\!) \cdot \tau = \mathsf{F}id \\
\equiv & \quad \{\, \text{functoriality of } \mathsf{F} \,\} \\
 & (\!|\mathsf{F}\tau|\!) \cdot \tau = id \ ,
\end{array}
$$

which completes the proof. □

26 Lambek's Lemma suggests an alternative view of datatype semantics [72]. Observe that the initial object $\tau : \mathsf{F}T \to T$ in $\mathbf{Alg}(\mathsf{F})$ may be understood as a *fixpoint* of endofunctor F in the underlying category $\mathbf{C}$ since $\mathsf{F}T$ and T are isomorphic and therefore abstractly the same ($\mathbf{C} = \mathbf{Set}$ in our case). Since τ is initial it is indeed the *least fixpoint* of F if we interpret the arrows in $\mathbf{C}$ as a pre-order. This observation has led to the so-called *least fixpoint semantics* which defines the algebra for a datatype as $\tau = \mu\mathsf{F}$ (symbol μ denotes the least fixpoint operator).

27 Let us take up functor $\mathsf{F} = 1 + \mathsf{K}_E \times \mathsf{Id}$ of datatypes in insert representation again (E fixed). Algebra $\tau = nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}} : \mathsf{F}(\mathsf{T}E) \to \mathsf{T}E$ as defined in Paragraph 19 is initial in $\mathbf{Alg}(\mathsf{F})$.

The proof of this claim necessarily involves to show that for any homomorphism h from τ to an arbitrary F-algebra $\alpha = e \triangledown \otimes : \mathsf{F}A \to A$ we have that $h = (\!|\alpha|\!) = (\!|e \triangledown \otimes|\!)$ for a suitable definition of $(\!|\cdot|\!)$, *i. e.*, we have to establish the validity of law Cata. We can see this as follows:

$$
\begin{array}{ll}
 & h \cdot \tau = \alpha \cdot \mathsf{F}h \\
\equiv & \quad \{\, \tau = nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}}, \alpha = e \triangledown \otimes \,\} \\
 & h \cdot (nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}}) = (e \triangledown \otimes) \cdot \mathsf{F}h \\
\equiv & \quad \{\, \text{sum} \,\} \\
 & (h \cdot nil^{\mathsf{T}}) \triangledown (h \cdot cons^{\mathsf{T}}) = (e \triangledown \otimes) \cdot \mathsf{F}h \\
\equiv & \quad \{\, \mathsf{F} = 1 + \mathsf{K}_E \times \mathsf{Id}, \text{functor} \,\} \\
 & (h \cdot nil^{\mathsf{T}}) \triangledown (h \cdot cons^{\mathsf{T}}) = (e \triangledown \otimes) \cdot (id_1 + id_E \times h)
\end{array}
$$

$$
\begin{array}{lll}
\equiv & \{\,\text{sum}\,\} & \\
 & (h \cdot nil^{\mathsf{T}}) \triangledown (h \cdot cons^{\mathsf{T}}) = (e \cdot id_1) \triangledown (\otimes \cdot (id_E \times h)) & \\
\equiv & \{\,\text{cancellation}\,\} & \\
 & \quad h \cdot nil^{\mathsf{T}} = e & \\
 & \wedge \;\; h \cdot cons^{\mathsf{T}} = \otimes \cdot (id_E \times h) & \\
\equiv & \{\,\text{point-wise reasoning}\,\} & \\
 & \quad h\, nil^{\mathsf{T}} = e & \\
 & \wedge \;\; h\,(cons^{\mathsf{T}}(x, xs)) = x \otimes (h\, xs)\;. &
\end{array}
$$

Finally, define $(\!|e \triangledown \otimes|\!)$ to mean

$$
\begin{array}{lll}
 & (\!|e \triangledown \otimes|\!)\, nil^{\mathsf{T}} = e & \\
\wedge & (\!|e \triangledown \otimes|\!)\,(cons^{\mathsf{T}}(x, xs)) = x \otimes ((\!|e \triangledown \otimes|\!)\, xs)\;, & \textsc{Cata-Ins-Rep}
\end{array}
$$

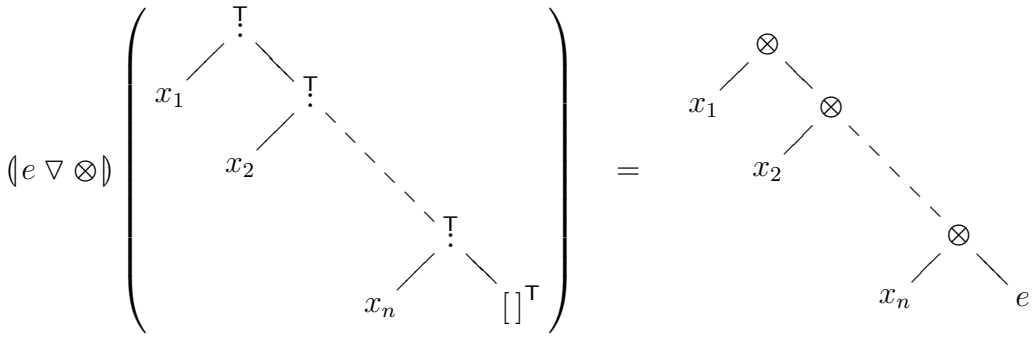
which validates law Cata because $(\!|e \triangledown \otimes|\!) = h$.

Note how $(\!|e \triangledown \otimes|\!)$ performs the case selection we have just referred to at the end of Paragraph 24. We can think of $(\!|e \triangledown \otimes|\!)$ as using a *pattern match* on the root of the argument's spine to decide for the selection of one of the two branches shown in Cata-Ins-Rep:

- if nil^{T} is encountered as the spine's root, it is replaced by e. Likewise,
- if $cons^{\mathsf{T}}$ is matched, it is replaced by function $\otimes$ and the catamorphism recursively descends into the spine xs whose height is one less than that of the original argument. This guarantees the recursion's termination.

The latter case explains the name *catamorphism* quite figuratively: the Greek prefix $\kappa\alpha\tau\acute{\alpha}$ (*downwards*) refers to the catamorphism's descent of the argument's spine.

The tree notation for spines which we have introduced in Paragraph 16, reveals that an F-catamorphism essentially performs the following simple tree transformation:



This observation immediately yields the so-called *reflection law* for catamorphisms:

$$(\!|\tau|\!) = id\ . \qquad \textsc{Cata-Reflect}$$

Cata-Ins-Rep obviously adheres to the general recursion scheme of *primitive recursive functions* [89]. The simplicity of the action of $(\!|\cdot|\!)_{\mathsf{F}}$ is all the more remarkable as it is already powerful enough to express the core operators of relational algebra, nested relational algebra [13, 20], and—as we will discuss to some depth in this text—modern orthogonal query languages for complex collection types like ODMG's OQL [23]. We can be more specific here: as the catamorphism combinator $(\!|\cdot|\!)_{\mathsf{F}}$ is a generic variant of the *sri* structural recursion operator on collections in insert representation it is capable of expressing exactly the PTIME queries, *i. e.*, those computable in polynomial time [13, 110, 111].

28 A functional inclined programmer has surely noticed that the action Cata-Ins-Rep of catamorphism $(\!|e \triangledown \otimes|\!)$ on a spine is exactly that of the well-known higher-order function *foldr* on a list:

$$\begin{array}{lcl} \mathit{foldr} & : & (E \times A \to A) \to A \to \mathsf{list}E \to A \\ \mathit{foldr} \otimes e\ \mathit{nil}^{\mathsf{list}} & = & e \\ \mathit{foldr} \otimes e\,(\mathit{cons}^{\mathsf{list}}(x, xs)) & = & x \otimes (\mathit{foldr} \otimes e\, xs)\ . \end{array}$$

As it happens, *foldr* is a catamorphism for lists in insert representation ($\mathsf{F} = 1 + \mathsf{K}_E \times \mathsf{Id}, \mathsf{T} = \mathsf{list}$), *i. e.*, we have $\mathit{foldr} \otimes e = (\!|e \triangledown \otimes|\!)$:

$$\begin{array}{ccc} 1 + E \times \mathsf{list}E & \xrightarrow{\mathit{nil}^{\mathsf{list}} \triangledown \mathit{cons}^{\mathsf{list}}} & \mathsf{list}E \\ {\scriptstyle \mathit{id}_1 + \mathit{id}_E \times (\mathit{foldr} \otimes e)} \downarrow & & \downarrow {\scriptstyle \mathit{foldr} \otimes e} \\ 1 + E \times A & \xrightarrow[e \triangledown \otimes]{} & A \end{array}$$

F-catamorphisms—with F being the endofunctor describing the insert representation of a datatype—indeed define a generalized *foldr* for any datatype former T.

More generally even, executing a calculation like the one in Paragraph 27 automatically provides us with a *fold* for any endofunctor F and datatype constructor T.

29 Example. A wide range of list-processing functions turn out to be catamorphisms from the list algebra $\tau = nil^{\mathsf{list}} \triangledown cons^{\mathsf{list}} : \mathsf{F}(\mathsf{list}E) \to \mathsf{list}E$ to other F-algebras, *i. e.*, these functions compute their result by spine transformations alone.

Let *Num* denote the object in **Set** representing the set of natural numbers $\mathbb{N}$. The function $length : \mathsf{list}E \to Num$ which computes the length of a list is a catamorphism from τ to the F-algebra $0 \triangledown (+ \cdot (K_1 \times id)) : \mathsf{F}Num \to Num$ with $K_x\, y = x$. Note that $+$ denotes the addition on *Num* in this example. We have $length = (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!)$ as the following calculation in which we apply the *length* catamorphism to the argument $[0,1,2]^{\mathsf{list}}$ exemplifies:

$$
\begin{array}{ll}
 & (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!)\, [0,1,2]^{\mathsf{list}} \\
= & \quad \{ \text{list insert representation} \} \\
 & (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!)\, (0^{\mathsf{list}} \mathbin{:} 1^{\mathsf{list}} \mathbin{:} 2^{\mathsf{list}} \mathbin{:} [\,]^{\mathsf{list}}) \\
= & \quad \{ \textsc{Cata-Ins-Rep} \} \\
 & (+ \cdot (K_1 \times id))\, (0, (\!|0 \triangledown (K_1 \times id)|\!)\, (1^{\mathsf{list}} \mathbin{:} 2^{\mathsf{list}} \mathbin{:} [\,]^{\mathsf{list}})) \\
= & \quad \{ \text{calculation} \} \\
 & 1 + ((\!|0 \triangledown (K_1 \times id)|\!)\, (1^{\mathsf{list}} \mathbin{:} 2^{\mathsf{list}} \mathbin{:} [\,]^{\mathsf{list}})) \\
= & \quad \{ \textsc{Cata-Ins-Rep} \} \\
 & 1 + ((+ \cdot (K_1 \times id))\, (1, (\!|0 \triangledown (K_1 \times id)|\!)\, (2^{\mathsf{list}} \mathbin{:} [\,]^{\mathsf{list}}))) \\
= & \quad \{ \text{calculation} \} \\
 & 1 + 1 + ((\!|0 \triangledown (K_1 \times id)|\!)\, (2^{\mathsf{list}} \mathbin{:} [\,]^{\mathsf{list}})) \\
= & \quad \{ \textsc{Cata-Ins-Rep} \} \\
 & 1 + 1 + ((+ \cdot (K_1 \times id))\, (2, (\!|0 \triangledown (K_1 \times id)|\!)\, [\,]^{\mathsf{list}})) \\
= & \quad \{ \text{calculation} \} \\
 & 1 + 1 + 1 + ((\!|0 \triangledown (K_1 \times id)|\!)\, [\,]^{\mathsf{list}}) \\
= & \quad \{ \textsc{Cata-Ins-Rep} \} \\
 & 1 + 1 + 1 + 0 \\
= & \quad \{ \text{arithmetic} \} \\
 & 3\,.
\end{array}
$$

◊

30 Example. A spine resulting from a catamorphism application may perfectly have a greater or smaller height than the argument spine. The selection function $filter\, p : \mathsf{list}E \to \mathsf{list}E$, with $p : E \to Bool$ being some predicate on the list elements of type E, is a catamorphism from τ to the F-algebra

$nil^{\mathsf{list}} \triangledown ((cons^{\mathsf{list}} \triangledown outr) \cdot (p \cdot outl)?) : \mathsf{F}(\mathsf{list}E) \to \mathsf{list}E$. Here, ? denotes the *infix conditional* defined by

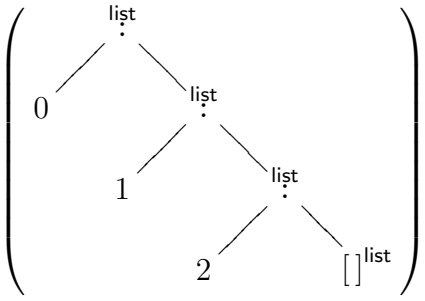
$$\begin{array}{lcl} ? & : & (A \to Bool) \to A \to A + A \\ p\,?\,x & = & \mathit{if}\ p\,x\ \mathit{then}\ \mathit{inl}\,x\ \mathit{else}\ \mathit{inr}\,x \end{array}$$

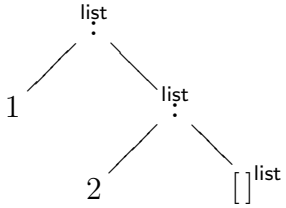
The morphism $(cons^{\mathsf{list}} \triangledown outr) \cdot (p \cdot outl)? : E \times \mathsf{list}E \to \mathsf{list}E$ implements a *conditional* $cons^{\mathsf{list}}$ in **Set** as a straightforward calculation shows:

$$\begin{array}{ll} & (cons^{\mathsf{list}} \triangledown outr) \cdot (p \cdot outl)? \\ \equiv & \quad \{\, \text{unfold } ? \,\} \\ & (cons^{\mathsf{list}} \triangledown outr) \cdot (\lambda z.\mathit{if}\ (p \cdot outl)\ z\ \mathit{then}\ \mathit{inl}z\ \mathit{else}\ \mathit{inr}z) \\ \equiv & \quad \{\, \text{conditional}, (cons^{\mathsf{list}} \triangledown outr)\ \text{strict} \,\} \\ & \lambda z.\mathit{if}\ (p \cdot outl)\ z\ \mathit{then}\ ((cons^{\mathsf{list}} \triangledown outr) \cdot inl)\ z \\ & \qquad\qquad\qquad \mathit{else}\ \ ((cons^{\mathsf{list}} \triangledown outr) \cdot inr)\ z \\ \equiv & \quad \{\, \text{sum cancellation} \,\} \\ & \lambda z.\mathit{if}\ (p \cdot outl)\ z\ \mathit{then}\ cons^{\mathsf{list}} z\ \mathit{else}\ outr z \\ \equiv & \quad \{\, \text{set } z = (x, xs), outl, outr \text{ in } \mathbf{Set} \,\} \\ & \lambda(x, xs).\mathit{if}\ p\,x\ \mathit{then}\ cons^{\mathsf{list}}(x, xs)\ \mathit{else}\ xs\ . \end{array}$$

(As the default category we are working in is **Set**, we will sometimes alternatively denote morphisms in this category by λ-expressions directly. The context of the current calculation will determine which notation suits best to improve readability.)

We thus have $\mathit{filter}\,p = (\!|\, nil^{\mathsf{list}} \triangledown ((cons^{\mathsf{list}} \triangledown outr) \cdot (p \cdot outl)?) \,|\!)$. To illustrate the action of $\mathit{filter}\,p$ on a spine let us instantiate E as Num and the filter predicate as $p = \lambda x.x \neq 0$:

$\mathit{filter}(\lambda x.x \neq 0)$ 

$=$ 

$\Diamond$

2.5 Type Functors

31 The collection type constructors T that we have met up to now have been monomorphic in the sense that were assuming a *fixed* element type E. To remedy this shortcoming, this section introduces real polymorphic type formers.

In the case of algebras for polymorphic type constructors T, there is the need to additionally parameterize the algebra's endofunctor by the object that represents the element type. This observation naturally leads to the use of bifunctors to model such algebras, *i. e.*, we enhance our definition of F-algebras as follows:

32 Definition. Let F denote a bi-endofunctor in a category $\mathbf{C}$ and let E, A be objects in $\mathbf{C}$. A **polymorphic F-algebra** is an arrow $\alpha : \mathsf{F}(E, A) \to A$. The object A is called the **carrier** of α.

Using the lingo of type theory, polymorphic algebras defined like above have a second-order type [22]. Type parameter E is *universally quantified* so that the proper type for α is $\forall E.\mathsf{F}(E, A) \to A$. Once we have fixed a specific element type E', we can revert to the situation in which the monofunctor $\mathsf{F}_{E'}$ (the left *section* of the bifunctor F) is sufficient to model the resulting, now monomorphic, algebra:

$$\alpha_{E'} : \mathsf{F}_{E'} A \to A \qquad \text{with} \qquad \mathsf{F}_{E'} A = \mathsf{F}(E', A) \quad \text{and} \quad \mathsf{F}_{E'} f = \mathsf{F}(id_{E'}, f) \ .$$

//

33 Polymorphic algebras already provide everything we need to define the algebra of a polymorphic collection type constructor T in insert representation. As before (see Paragraph 19), $\mathsf{T}E$ will be its carrier but the polymorphic algebra differs in that we pass the element type E as an additional argument.

$$\alpha = cons^{\mathsf{T}} \triangledown nil^{\mathsf{T}} : \mathsf{F}(E, \mathsf{T}E) \to \mathsf{T}E$$

with

$$\begin{array}{lrcl} & \mathsf{F}(E, A) & = & 1 + E \times A \\ \wedge & \mathsf{F}(f, g) & = & id_1 + f \times g \quad . \end{array}$$

Fixing a specific element type E' leads to the algebra functor we were used to until now so that our earlier discussion of monomorphic algebras applies without revision:

$$\begin{array}{lrcccl} & \mathsf{F}_{E'}(\mathsf{T}E') & = & \mathsf{F}(E', \mathsf{T}E') & = & 1 + E' \times \mathsf{T}E' \\ \wedge & \mathsf{F}_{E'} f & = & \mathsf{F}(id_{E'}, f) & = & id_1 + id_{E'} \times f \quad , \end{array}$$

i. e., $\mathsf{F}_{E'} = \mathsf{K}_1 + \mathsf{K}_{E'} \times \mathsf{Id}$.

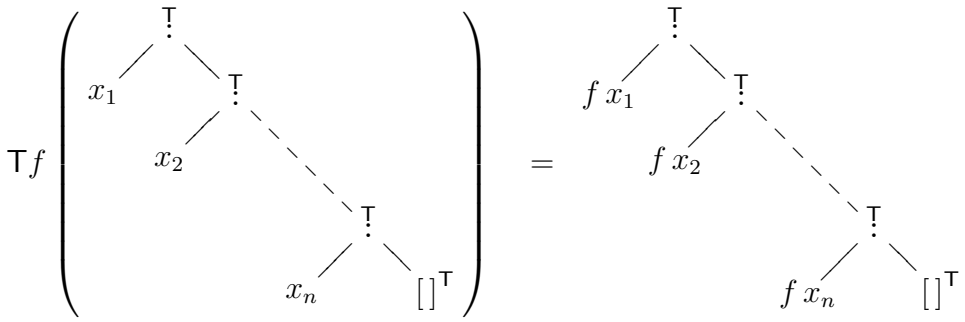
34 Type functors. We have claimed earlier, that there is more to the type former T than lifting a type E to a constructed (collection) type $\mathsf{T}E$: T may be turned into a functor as well.

Let $\tau : \mathsf{F}(E, \mathsf{T}E) \to \mathsf{T}E$ be the initial algebra for a polymorphic type former T. To establish T's functoriality there does not remain much to be done. We are already quite familiar with the object mapping part of T: T forms a new type $\mathsf{T}E$ from a given type E (types being objects in **Set**). Given this, the type of T's object mapping part clearly is

$$f : A \to B \quad \Rightarrow \quad \mathsf{T}f : \mathsf{T}A \to \mathsf{T}B \ ,$$

which exactly states the typing rules for *map* which we have encountered earlier in Paragraph 13 (replace T by list to obtain a perfect match).

To be more concrete, let $\tau = nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}} : \mathsf{F}(E, \mathsf{T}E) \to \mathsf{T}E$ with $\mathsf{F}(f, g) = id_1 + f \times g$ as in the previous paragraph. If we assume for a minute that *map* indeed implements the desired arrow mapping we have that



where $f : E \to E'$. Computing $\mathsf{T}f$ is a perfect job for a catamorphism! The spine transformation depicted above is obviously implemented by

$$\begin{array}{ll} & (\!| nil^{\mathsf{T}} \triangledown (cons^{\mathsf{T}} \cdot (f \times id)) |\!) \\ = & \quad \{ \text{ sum } \} \\ & (\!| (nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}}) \cdot (id_1 + f \times id) |\!) \\ = & \quad \{ \text{ definitions of } \mathsf{F} \text{ and } \tau \} \\ & (\!| \tau \cdot \mathsf{F}(f, id) |\!) \ . \end{array}$$

It turns out that this *map functor* $\mathsf{T}f = (\!| \tau \cdot \mathsf{F}(f, id) |\!)$ exactly provides what we were after.

35 Definition. Let $\tau : \mathsf{F}(E, \mathsf{T}E) \to \mathsf{T}E$ be the initial algebra for the polymorphic type former T. Its associated **type functor** (sometimes referred to as **map functor**) is defined by

$$\mathsf{T}f \quad = \quad (\!|\tau \cdot \mathsf{F}(f, id)|\!)\ . \qquad \textsc{Type-Functor}$$

The pair (τ, T) is called the **initial type** [10] induced by the bifunctor F.

We omit the details of formally establishing that this indeed turns T into a functor. The proof of $\mathsf{T}id = id$ is a trivial consequence of F's bifunctoriality and law Cata-Reflect. The distribution property follows immediately with the help of law Cata-Map-Fusion (see Paragraph 39 below and [10, 42, 57]). //

36 Example. For the type constructor $\mathsf{T} = \mathsf{list}$ we have $\mathsf{list}f = mapf$ where *map* denotes the well-known higher-order function which applies function f to every element of its list argument. For the sake of uniformity we write $\mathsf{list}f$ instead of $mapf$ throughout this text as we will also encounter the type functors set and bag among others.

The type functor concept is rather generic and gives us a tool to derive *map*-like functions for any initial datatype.

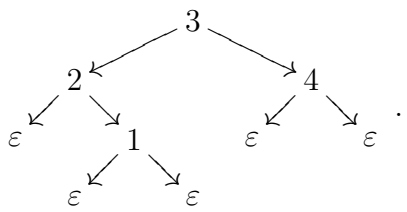
Consider the abstract datatype bintree of binary trees with inner nodes labeled with a tag of type E. Such trees are built by means of the two constructor functions

$$\begin{array}{lcl} empty & : & 1 \to \mathsf{bintree}E \\ node & : & E \times \mathsf{bintree}E \times \mathsf{bintree}E \to \mathsf{bintree}E \end{array}$$

where *empty* denotes the empty tree ε and $node(n, t_1, t_2)$ constructs a tree with root n and left and right subtrees t_1 and t_2 respectively. The constructor expression

$$node(3, node(2, empty, node(1, empty, empty)), node(4, empty, empty))$$

thus builds the binary tree



The initial algebra for this datatype is $\tau = empty \triangledown node : \mathsf{F}(E, \mathsf{T}E) \to \mathsf{T}E$ where $\mathsf{F}(E, \mathsf{T}E) = 1 + E \times \mathsf{T}E \times \mathsf{T}E$, $\mathsf{F}(f, g) = id_1 + f \times g \times g$, and of course $\mathsf{T} = \mathsf{bintree}$.

How does the associated type functor look like? By instantiating the ingredients of TYPE-FUNCTOR and conducting the simple calculation below, we see that bintree's map functor indeed is a *tree map*: the catamorphism for the functor F defined above (whose derivation we leave to the kind reader) recursively descends into the two subtrees of an inner node until an empty tree ε is encountered which is left untouched by the tree map:

$$
\begin{array}{ll}
 & \mathsf{bintree}\, f \\
= & \quad \{\, \text{definition of type functor} \,\} \\
 & (\!|\, \tau \cdot \mathsf{F}(f, id) \,|\!) \\
= & \quad \{\, \text{unfold } \tau, \text{unfold } \mathsf{F} \,\} \\
 & (\!|\, (\mathit{empty} \triangledown \mathit{node}) \cdot (\mathit{id}_1 + f \times \mathit{id} \times \mathit{id}) \,|\!) \\
= & \quad \{\, \text{sum} \,\} \\
 & (\!|\, \mathit{empty} \triangledown (\mathit{node} \cdot (f \times \mathit{id} \times \mathit{id})) \,|\!) \; .
\end{array}
$$

As another and final example let us investigate a non-collection type former. The type $\mathsf{maybe}E$ is the same as E but it additionally comes with a distinguished element *nothing* which may serve as, *e. g.*, an error value. Type former maybe has constructors

$$
\begin{array}{lcl}
\mathit{nothing} & : & 1 \to \mathsf{maybe}E \\
\mathit{just} & : & E \to \mathsf{maybe}E \; .
\end{array}
$$

The accompanying initial algebra is $\tau = \mathit{nothing} \triangledown \mathit{just} : \mathsf{F}(E, \mathsf{maybe}E) \to \mathsf{maybe}E$ with $\mathsf{F}(E, \mathsf{maybe}E) = 1 + E$, $\mathsf{F}(f, g) = \mathit{id}_1 + f$. Its type functor turns out to be a specialized function application operator that propagates the error value *nothing* and otherwise simply applies the given function:

$$
\begin{array}{ll}
 & \mathsf{maybe}\, f \\
= & \quad \{\, \text{definition of type functor} \,\} \\
 & (\!|\, \tau \cdot \mathsf{F}(f, id) \,|\!) \\
= & \quad \{\, \text{unfold } \tau, \text{unfold } \mathsf{F} \,\} \\
 & (\!|\, (\mathit{nothing} \triangledown \mathit{just}) \cdot (\mathit{id}_1 + f) \,|\!) \\
= & \quad \{\, \text{sum} \,\} \\
 & (\!|\, \mathit{nothing} \triangledown (\mathit{just} \cdot f) \,|\!) \\
\equiv & \quad \{\, \text{point-wise reasoning} \,\} \\
 & \quad \mathsf{maybe}\, f\, \mathit{nothing} \;=\; \mathit{nothing} \\
 & \wedge \quad \mathsf{maybe}\, f\, (\mathit{just}\, x) \;=\; \mathit{just}(f\, x) \; .
\end{array}
$$

Among other uses, $\mathsf{maybe}\, f$ is helpful during the implementation of and reasoning about functional exception handling mechanisms. ◊

2.6 Fusion

37 Now that it has now become more clear that spine transformations will be the basic building blocks from which we will construct queries, we can imagine that a query will typically be built by the *functional composition* of such transformations.

In other words, if Q denotes a query expression then it will exhibit a structure that may be roughly sketched as

$$Q \quad \equiv \quad \cdots h \cdot g \cdot f \cdots$$

where the functions f, g, and h represent spine transformers, in particular catamorphisms and type functor applications (*map*s). Building query expressions like this leads to a clear and modular way of thinking about query construction itself: complex queries may be formulated by incrementally adding a basic transformer to an already existing sequence of function compositions. This *compositional* flavor of internal query representation facilitates to offer the same flexibility at the user level (where this taste of query formulation is often referred to as *orthogonality* if there is nothing but the language's type system that stops the user from freely combining query expressions).

We will not go into the details here as Chapter 6 will investigate the matter more closely, but let us note that the compositional style does not come for free:

- Each function generates an *intermediate result* that has to be taken up later by the next function in the composition sequence. The management of intermediate results may be quite costly.
- Catamorphisms essentially traverse the spine of their arguments while performing their transformation task and each catamorphism in the sequence does so from the spine's root downwards. It may be desirable to combine two or more of these transformations in such a way that a *single* walk of the spine is sufficient.

What we are calling for here is the *fusion* of two adjacent spine transformers into one [37, 44].

Two well-known theorems provide the handles we need to fuse two catamorphisms as well as a type functor application followed by a catamorphism in the way just described. In Chapter 6, we will accompany these two laws with a third theorem—also known as the *acid rain theorem* [112]—which justifies an effective fusion technique that will prove especially useful during the derivation of a stream-based (or pipelined) program from a given query.

38 Promotion Theorem [3, 56, 57]. Let F be an endofunctor in category $\mathbf{C}$ so that $\mathbf{Alg}(\mathsf{F})$ has an initial object τ. For any two F-algebras $f : \mathsf{F}A \to A$ and $g : \mathsf{F}B \to B$ and an F-homomorphism $h : A \to B$ from f to g we have that

$$h \cdot (\!|f|\!) \quad = \quad (\!|g|\!) \; . \qquad \text{CATA-FUSION}$$

Proof. A single calculation suffices to establish the claim:

$$\begin{array}{ll} & h \cdot (\!|f|\!) \cdot \tau \\ = & \quad \{ \text{CATA: } (\!|f|\!) \cdot \tau = f \cdot \mathsf{F}(\!|f|\!) \} \\ & h \cdot f \cdot \mathsf{F}(\!|f|\!) \\ = & \quad \{ \text{HOM: } h \cdot f = g \cdot \mathsf{F}h \} \\ & g \cdot \mathsf{F}h \cdot \mathsf{F}(\!|f|\!) \\ = & \quad \{ \mathsf{F} \text{ functor} \} \\ & g \cdot \mathsf{F}(h \cdot (\!|f|\!)) \\ \Rightarrow & \\ & h \cdot (\!|f|\!) \cdot \tau = g \cdot \mathsf{F}(h \cdot (\!|f|\!)) \\ \equiv & \quad \{ \text{CATA, uniqueness of } (\!|g|\!) \} \\ & h \cdot (\!|f|\!) = (\!|g|\!) \; . \end{array}$$

□

As any F-catamorphism is a special F-homomorphism, law CATA-FUSION indeed is the catamorphism fusion tool we have appealed for above.

39 Theorem. If (τ, T) denotes the initial type (Definition 35) for the endofunctor F, then we can fuse a composition of an F-catamorphism with its type functor as follows (under typings $f : A \to B$, $g : \mathsf{F}B \to B$):

$$(\!|g|\!) \cdot \mathsf{T}f \quad = \quad (\!|g \cdot \mathsf{F}(f, id)|\!) \; . \qquad \text{CATA-MAP-FUSION}$$

Proof. The proof invokes law CATA-FUSION as a subroutine which is hardly surprising because the arrow mapping of the type functor T has been defined as a specific F-catamorphism (see Definition 35). We calculate as follows (*cf.* [10]):

$$\begin{array}{ll} & (\!|g|\!) \cdot \mathsf{T}f = (\!|g \cdot \mathsf{F}(f, id)|\!) \\ \equiv & \quad \{ \text{TYPE-FUNCTOR for } \mathsf{T} \} \\ & (\!|g|\!) \cdot (\!|\tau \cdot \mathsf{F}(f, id)|\!) = (\!|g \cdot \mathsf{F}(f, id)|\!) \end{array}$$

$$
\begin{array}{ll}
\Leftarrow & \quad \{\, \textsc{Cata-Fusion} \,\} \\
& (\!|g|\!) \cdot \tau \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, (\!|g|\!)) \\
\equiv & \quad \{\, \textsc{Cata}\text{: } (\!|g|\!) \cdot \tau = g \cdot \mathsf{F}(id, (\!|g|\!)) \,\} \\
& g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(f, id) \cdot \mathsf{F}(id, (\!|g|\!)) \\
\equiv & \quad \{\, \mathsf{F} \text{ bifunctor} \,\} \\
& g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(f, id) = g \cdot \mathsf{F}(id, (\!|g|\!)) \cdot \mathsf{F}(f, id) \\
\equiv & \quad \{\, \text{reflexivity of } = \,\} \\
& \mathit{true}\ ,
\end{array}
$$

which establishes the law as a logical consequence of the trivial premise *true*. □

40 Example. To become acquainted with the flavor of program transformations which now have been justified by the fusion laws, let us take a closer look at a simple spine transformer composition and how to derive its fused equivalent.

The expression $\mathit{filter}\, p \cdot \mathit{map}\, f : \mathsf{list} A \to \mathsf{list} B$ traverses a given argument spine twice. During the first traversal, $\mathit{map}\, f$ generates a new spine by replacing each list element x by $f\, x$. The second traversal then takes up this intermediate spine to built the final result from all elements that satisfy predicate p.

We can clearly accomplish the same by walking down the spine only once. In order no to alter the meaning of the expression we only have to take care to apply f to the list elements *before* predicate p checks for their inclusion in the final result. Fusion allows us to automatically derive an expression that does just this. No intermediate result is produced.

As before, we are working with collections in insert representation, *i. e.*, the underlying algebra functor is $\mathsf{F}(f, g) = id_1 + f \times g$. Although the example expression operates on lists only this is not principle to the method, so let us generalize the argument to some arbitrary type constructor T. We have $\mathit{map}\, f = \mathsf{T} f$, making the expression subject to law Cata-Map-Fusion. Once we have unfolded *filter*'s definition as an F-catamorphism (see Example 30), we immediately spot the possibility for fusion:

$$
\begin{array}{ll}
& \mathit{filter}\, p \cdot \mathsf{T} f \\
= & \quad \{\, \text{unfold } \mathit{filter} \,\} \\
& (\!|\mathit{nil}^{\mathsf{T}} \triangledown ((\mathit{cons}^{\mathsf{T}} \triangledown \mathit{outr}) \cdot (p \cdot \mathit{outl})?)|\!) \cdot \mathsf{T} f \\
= & \quad \{\, \textsc{Cata-Map-Fusion} \,\} \\
& (\!|(\mathit{nil}^{\mathsf{T}} \triangledown ((\mathit{cons}^{\mathsf{T}} \triangledown \mathit{outr}) \cdot (p \cdot \mathit{outl})?)) \cdot \mathsf{F}(f, id)|\!)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{\, \text{definition of } \mathsf{F} \,\} \\
&\quad (\!|(nil^{\mathsf{T}} \triangledown ((cons^{\mathsf{T}} \triangledown outr) \cdot (p \cdot outl)?)) \cdot (id_1 + f \times id)|\!) \\
&= \quad \{\, \text{sum} \,\} \\
&\quad (\!|nil^{\mathsf{T}} \triangledown ((cons^{\mathsf{T}} \triangledown outr) \cdot (p \cdot outl)? \cdot (f \times id))|\!) \;,
\end{aligned}
$$

which already completes the fusion step. We can rewrite this expression a bit further so that it reveals its nature of actually being a specialized *filter*:

$$
\begin{aligned}
&\quad (\!|nil^{\mathsf{T}} \triangledown ((cons^{\mathsf{T}} \triangledown outr) \cdot (p \cdot outl)? \cdot (f \times id))|\!) \\
&= \quad \{\, p? \cdot f = (f + f) \cdot (p \cdot f)? \,\} \\
&\quad (\!|nil^{\mathsf{T}} \triangledown ((cons^{\mathsf{T}} \triangledown outr) \cdot (f \times id + f \times id) \cdot (p \cdot outl \cdot (f \times id))?)|\!) \\
&= \quad \{\, \text{product, sum} \,\} \\
&\quad (\!|nil^{\mathsf{T}} \triangledown (((cons^{\mathsf{T}} \cdot (f \times id)) \triangledown (outr \cdot (f \times id))) \cdot (p \cdot f \cdot outl)?)|\!) \\
&= \quad \{\, \text{product} \,\} \\
&\quad (\!|nil^{\mathsf{T}} \triangledown (((cons^{\mathsf{T}} \cdot (f \times id)) \triangledown outr) \cdot (p \cdot f \cdot outl)?)|\!) \;.
\end{aligned}
$$

We finally ended up with a single spine transformer instead of two. Subsequent transformation steps could now proceed with fusion given that the newly derived catamorphism is adjacent to other transformers in a larger composition chain. While we do not formulate this as our primary optimization goal, "excessive" fusion—possibly hand in hand with helper transformations that rely on commutativity, distributivity, or similar algebraic properties—could thus be exploited to minimize the spine transformer count in a given expression. ◊

41 The development of strategies for the removal of intermediate data structures, for which Wadler [118] has coined the term *deforestation*, has always been a major driving force of the program transformation community.[5] The impact of this research on our work will be the focus of Chapter 6.

Taken as a whole, the transformation exercised in the last example may look rather involved. This is especially true for the step in which the infix conditional is rewritten: $p? \cdot f = (f + f) \cdot (p \cdot f)?$. While this is no deep insight at all, the step must be regarded as a so-called *Eureka step* in this context, *i. e.*, a rewriting that is not immediately motivated by the goal which the current calculation strives for. Even worse, asking if a particular morphism is a homomorphism which, *e. g.*, is necessary to check the applicability of law Cata-Fusion, is undecidable in general [14, 20].

[5] Deforestation removes intermediate data structures which Wadler collectively refers to as trees, hence the name *deforestation*.

Such obstacles hinder the construction of a completely unguided program transformation system, an instance of which—a database query optimizer—we are discussing in this text.

As it will turn out, however, we do *not* need the full power of fusion at our disposal to generate efficient programs from query expressions. While still retaining the expressiveness needed to cover modern orthogonal query languages, the query compiler (Chapter 3) will emit expressions in a form that is amenable to fusion using a simple yet powerful one-step transformation, *cheap deforestation*. This process will be fully automatic (see Chapter 6 and [50, 51, 59, 74, 112]).

2.7 Datatype Equations

42 The algebras we have met up to now have all been *anarchic* or *free* in the sense that they do not obey any *equations*. An equation for a datatype—actually for its algebra to be precise—is imposed by the designer of that type at will, or, put more positive, equations provide the instrument to introduce a controlled form of *confusion*: two distinct algebra expressions may denote the very same object if the equality of these expressions is derivable from a set of specified equations. It is important not to confuse equations with the notion of *law* which represents provable statements like the CATA-FUSION theorem.

We have chosen list as the predominant example of a type former until now because its algebra is indeed free. The equational theory of the list algebra in insert representation is empty: there is no confusion between a list spine (in $\mathsf{F}(\mathsf{list}E)$) and the actual list value (in $\mathsf{list}E$) it represents. Note that this is just the isomorphism statement of Lambek's Lemma:

$$\mathsf{F}(\mathsf{list}E) \underset{\bar{\tau}}{\overset{\tau}{\rightleftarrows}} \mathsf{list}E \quad ,$$

which, in this context, can be read as: syntactical equality means semantical equality.

The need for datatypes that obey equations arises with the problem of modeling non-free algebras like those for the type formers bag and set. No isomorphism like the above exists between $\mathsf{F}(\mathsf{bag}E)$ or $\mathsf{F}(\mathsf{set}E)$ and their respective carriers: the left-commutativity of $\overset{\mathsf{bag}}{:}$ and $\overset{\mathsf{set}}{:}$ (the latter additionally being left-idempotent or left-absorptive) renders these constructors *non-injective* (see Paragraph 16). The set spines $[1,2,4]^{\mathsf{set}}$ and $[2,4,4,1,2]^{\mathsf{set}}$ both represent the set value $\{1,2,4\}$, for example.

43 How does the presence of confusion affect query processing in our catamorphism-based framework? To retain the determinism of the catamorphism action on a spine we need the action to be indifferent towards the non-injectivity of the underlying algebra's constructors. Otherwise we would in general have $(\!|e \triangledown \otimes|\!)\,[1,2,4]^{\mathsf{set}} \neq (\!|e \triangledown \otimes|\!)\,[2,4,4,1,2]^{\mathsf{set}}$ which clearly is not what we aim for.

To be more precise, assume a type T in insert representation with a left-commutative constructor $\oplus$. A catamorphism $(\!|e \triangledown \otimes|\!)$ over values of T does not care about the actual order of constructor applications, *i. e.*, we have

$$(\!|e \triangledown \otimes|\!) \cdot \oplus \cdot (outl \vartriangle (\oplus \cdot outr)) \\ = \otimes \cdot ((outl \cdot outr) \vartriangle (\otimes \cdot (id \times outr))) \cdot (id \times (id \times (\!|e \triangledown \otimes|\!))) \; . \qquad \textsc{Cata-Comm}$$

Read point-wise and slightly simplified, this is equivalent to the equality

$$(\!|e \triangledown \otimes|\!)\,(y \oplus x \oplus xs) = x \otimes ((\!|e \triangledown \otimes|\!)\,(y \oplus xs)) \; .$$

Let $\ominus$ analogously denote a left-idempotent insertion constructor. A duplicate element at the front of a spine does not influence a catamorphism action. We express this by

$$(\!|e \triangledown \otimes|\!) \cdot \ominus \cdot (outl \vartriangle \ominus) = \otimes \cdot (id \times (\!|e \triangledown \otimes|\!)) \; , \qquad \textsc{Cata-Idem}$$

or, expressed point-wise,

$$(\!|e \triangledown \otimes|\!)\,(x \ominus x \ominus xs) = x \otimes (\!|e \triangledown \otimes|\!)\,xs \; .$$

44 The complexity of Cata-Comm and Cata-Idem may make it hard to detect, but both equalities follow a common theme. If we let τ denote the initial algebra for the type T in insert representation (so that we have $\oplus = \tau \cdot inr$ and $\ominus = \tau \cdot inr$ in the case of Cata-Comm and Cata-Idem, respectively), the equalities adhere to the pattern

$$(\!|f|\!) \cdot L\tau = Rf \cdot \mathsf{H}(\!|f|\!) \; ,$$

if we instantiate functions L, R, and H for Cata-Comm as $L\tau = (\tau \cdot inr) \cdot (outl \vartriangle ((\tau \cdot inr) \cdot outr))$, $Rf = (f \cdot inr) \cdot ((outl \cdot outr) \vartriangle ((f \cdot inr) \cdot (id \times outr)))$, and $\mathsf{H}(\!|f|\!) = id \times (id \times (\!|f|\!))$. For equality Cata-Idem we get $L\tau = (\tau \cdot inr) \cdot (outl \vartriangle (\tau \cdot\ inr))$, $Rf = f \cdot inr$, and $\mathsf{H}(\!|f|\!) = id \times (\!|f|\!)$.

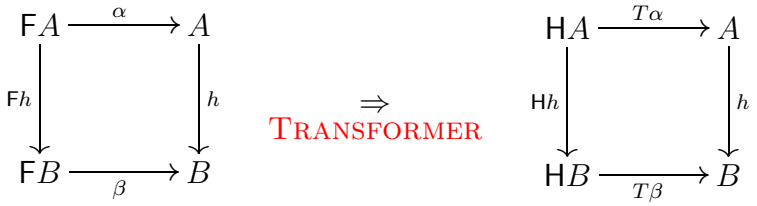
Functions L and R are examples of *transformers* [42, 43], as we will soon see. Transformers are categorical encodings of *terms with variables* of type

H. In the form we are using them here, "terms" L and R play the role of representing *the left- and right-hand side* (*lhs* and *rhs*) of a datatype equation, respectively. For our purposes, transformers make the burdensome introduction of non-categorical notions like *signature*, *syntax*, and *variables*—which are otherwise needed to reason about terms—superfluous. Using transformers, we can stay in the categorical realm.

45 Definition. Let $h : A \to B$ be an F-homomorphism from F-algebra $\alpha : \mathsf{F}A \to A$ to $\beta : \mathsf{F}B \to B$, and let H denote another endofunctor. An arrow $T : \mathbf{Alg}(\mathsf{F}) \to \mathbf{Alg}(\mathsf{H})$ is called a **transformer** of type H if

$$h \cdot \alpha = \beta \cdot \mathsf{F}h \quad \Rightarrow \quad h \cdot T\alpha = T\beta \cdot \mathsf{H}h \; . \qquad \textsc{Transformer}$$

Rephrasing Transformer diagrammatically clearly reveals T's property of preserving the homomorphism h:



∎

46 Verifying the Transformer property for equalities Cata-Comm and Cata-Idem is a straightforward but somewhat tedious task. We will establish the validity of Transformer for the left-hand side term $L\tau = (\tau \cdot inr) \cdot (outl \vartriangle ((\tau \cdot inr) \cdot outr))$ of type $\mathsf{H}h = id \times (id \times h)$ in equality Cata-Comm only here. The other terms are shown to be transformers using similar calculational arguments.

So let $\alpha = e \triangledown \otimes : \mathsf{F}A \to A$ and $\beta = z \triangledown \oplus : \mathsf{F}B \to B$ be two F-algebras and $h : A \to B$ any F-homomorphism (which does not necessarily need to be a catamorphism as they show up in the equalities) between these as in the above definition:

$$\begin{array}{ll} & h \cdot L\alpha = L\beta \cdot \mathsf{H}h \\ \equiv & \quad \{ \text{ unfold } L \text{ and } \mathsf{H}, \text{product} \} \\ & \quad h \cdot \otimes \cdot (outl \vartriangle (\otimes \cdot outr)) \\ = & \quad \oplus \cdot (outl \vartriangle (\oplus \cdot outr)) \cdot (id \times (id \times h)) \\ \equiv & \quad \{ h \text{ homomorphism} \} \end{array}$$

$$
\begin{array}{ll}
 & \oplus \cdot (id \times h) \cdot (outl \vartriangle (\otimes \cdot outr)) \\
= & \oplus \cdot (outl \vartriangle (\oplus \cdot outr)) \cdot (id \times (id \times h) \\
\equiv & \quad \{ \text{absorption} \} \\
 & \oplus \cdot (outl \vartriangle (h \cdot \otimes \cdot outr)) \\
= & \oplus \cdot (outl \vartriangle (\oplus \cdot outr)) \cdot (id \times (id \times h) \\
\equiv & \quad \{ \text{product} \} \\
 & \oplus \cdot (outl \vartriangle (h \cdot \otimes \cdot outr)) \\
= & \oplus \cdot ((outl \cdot (id \times (id \times h))) \vartriangle (\oplus \cdot outr \cdot (id \times (id \times h)))) \\
\equiv & \quad \{ \text{product: } outl\,(f \times g) = f \cdot outl, outr\,(f \times g) = g \cdot outl \} \\
 & \oplus \cdot (outl \vartriangle (h \cdot \otimes \cdot outr)) = \oplus \cdot ((outl \cdot id) \vartriangle (\oplus \cdot (id \times h) \cdot outr)) \\
\equiv & \quad \{ h \text{ homomorphism, identity} \} \\
 & \oplus \cdot (outl \vartriangle (h \cdot \otimes \cdot outr)) = \oplus \cdot (outl \vartriangle (h \cdot \otimes \cdot outr)) \\
\equiv & \quad \{ \text{reflexivity of } = \} \\
 & true\ .
\end{array}
$$

The transformer notion is the only tool we have to fall back upon to define equations for a datatype. Two transformers sharing the same type encode the lhs and rhs terms of an equality, so it is obvious to simply define a datatype equation as a pair of correctly typed transformers. This is exactly what the following definition states.

47 Definition. A pair of transformers (L, R) defines a **datatype equation** (or **equation** for short) if L and R are of the same type. Equation (L, R) **holds** for algebra α if $L\alpha = R\alpha$. //

48 The equalities Cata-Comm and Cata-Idem of Paragraph 43 do not only describe the action of a catamorphism on the non-free bag and set algebras, they also induce the transformers we need to characterize these algebras themselves. For any algebra α in insert representation, the datatype equation describing the left-commutativity of insertion is given by the transformers of Cata-Comm, the left-idempotence equation is derived from the transformers occurring in Cata-Idem:

$$
\begin{array}{c}
\big((\alpha \cdot inr) \cdot (outl \vartriangle ((\alpha \cdot inr) \cdot outr))\ , \\
(\alpha \cdot inr) \cdot ((outl \cdot outr) \vartriangle ((\alpha \cdot inr) \cdot (id \times outr)))\ \big)
\end{array}
\qquad \text{Eq-Comm}
$$

$$
\big((\alpha \cdot inr) \cdot (outl \vartriangle (\alpha \cdot\ \ inr))\ ,\ \alpha \cdot inr\ \big)\ . \qquad \text{Eq-Idem}
$$

Imposing EQ-COMM on the list algebra will give us a bag algebra as we were calling for in Paragraph 16: if EQ-COMM holds for the algebra $[\,]^{\mathsf{bag}} \triangledown {}^{\mathsf{bag}}_{\cdot\cdot}$, then, according to Definition 47, we have

$$
\begin{array}{ll}
& L([\,]^{\mathsf{bag}} \triangledown {}^{\mathsf{bag}}_{\cdot\cdot}) = R([\,]^{\mathsf{bag}} \triangledown {}^{\mathsf{bag}}_{\cdot\cdot}) \\
\equiv & \quad \{\, \text{instantiate } L, R \text{ for EQ-COMM, product} \,\} \\
& {}^{\mathsf{bag}}_{\cdot\cdot} \cdot (outl \vartriangle ({}^{\mathsf{bag}}_{\cdot\cdot} \cdot outr)) = {}^{\mathsf{bag}}_{\cdot\cdot} \cdot ((outl \cdot outr) \vartriangle ({}^{\mathsf{bag}}_{\cdot\cdot} \cdot (id \times outr))) \\
\equiv & \quad \{\, \text{point-wise, apply both sides to } (y,(x,xs)) \,\} \\
& y \,{}^{\mathsf{bag}}_{\cdot\cdot}\, (x \,{}^{\mathsf{bag}}_{\cdot\cdot}\, xs) = x \,{}^{\mathsf{bag}}_{\cdot\cdot}\, (y \,{}^{\mathsf{bag}}_{\cdot\cdot}\, xs)\ .
\end{array}
$$

The construction of a set algebra includes the above step and additionally uses equation EQ-IDEM to establish the left-idempotence of constructor ${}^{\mathsf{set}}_{\cdot\cdot}$:

$$
\begin{array}{ll}
& L([\,]^{\mathsf{set}} \triangledown {}^{\mathsf{set}}_{\cdot\cdot}) = R([\,]^{\mathsf{set}} \triangledown {}^{\mathsf{set}}_{\cdot\cdot}) \\
\equiv & \quad \{\, \text{instantiate } L, R \text{ for EQ-IDEM, product} \,\} \\
& {}^{\mathsf{set}}_{\cdot\cdot} \cdot (outl \vartriangle {}^{\mathsf{set}}_{\cdot\cdot}) = {}^{\mathsf{set}}_{\cdot\cdot} \\
\equiv & \quad \{\, \text{point-wise, apply both sides to } (x,xs) \,\} \\
& x \,{}^{\mathsf{set}}_{\cdot\cdot}\, (x \,{}^{\mathsf{set}}_{\cdot\cdot}\, xs) = x \,{}^{\mathsf{set}}_{\cdot\cdot}\, xs\ .
\end{array}
$$

49 Imposing an equation (L,R) on an F-algebra $\alpha : \mathsf{F}A \to A$ induces an equivalence relation Eq on $\mathsf{F}A$ that is a congruence for α [42]. Terms in $\mathsf{F}A$ that belong to the same equivalence class denote the same element of the carrier A—which renders the constructors described by F non-injective and thus introduces the desired confusion.

More precisely, Eq is the least equivalence relation containing all pairs $((L\alpha)t, (R\alpha)t)$ for terms t in $\mathsf{F}A$. Additionally, Eq is a congruence for α so that Eq features a closure property with respect to the constructors encoded in F. For the insert representation case, which means that α is of the form $e \triangledown \otimes$, this boils down to (let x, y, xs, ys be in $\mathsf{F}A$)

$$(x,y) \in Eq \wedge (xs,ys) \in Eq \quad \Rightarrow \quad (x \otimes xs, y \otimes ys) \in Eq\ .$$

Forcing the equation upon the algebra thus means to work with the *quotient algebra*, denoted $\alpha/(L,R)$. This construction extends to and preserves initial algebras: if $\mathbf{Alg}(\mathsf{F})$ includes an initial object τ, then $\tau/(L,R)$ is initial in the subcategory of $\mathbf{Alg}(\mathsf{F})$ containing all and only those F-algebras for which (L,R) holds. For a proof of this statement see Fokkinga's original work on transformers [42, page 122 ff].

50 Let us not proceed without explicitly noting that query processing with catamorphisms over non-free algebras like those for bag and set introduces a well-definedness condition for any catamorphism application.

To illustrate this caveat, suppose that τ is an initial F-algebra upon which we have forced the equation (L, R) of type H as described in the previous paragraph. Furthermore, let α denote another algebra, implementing the target datatype of a query. With these ingredients, we have

$$
\begin{array}{ll}
 & \mathit{true} \\
\equiv & \quad \{\, \text{reflexivity of } = \,\} \\
 & (\!|\alpha|\!) = (\!|\alpha|\!) \\
\equiv & \quad \{\, \text{equation } (L, R) \text{ holds for } \tau \,\} \\
 & (\!|\alpha|\!) \cdot L\tau = (\!|\alpha|\!) \cdot R\tau \\
\Rightarrow & \quad \{\, \textsc{Transformer} \text{ on both sides} \,\} \\
 & L\alpha \cdot \mathsf{H}(\!|\alpha|\!) = R\alpha \cdot \mathsf{H}(\!|\alpha|\!) \;,
\end{array}
$$

which tells us that (L, R) holds for α as well, at least on the range of $(\!|\alpha|\!)$.

In the sequel, which will put the theory into query processing practice, this observation will require us to make sure that a catamorphism applied to a bag or set has as its target an algebra with a left-commutative or left-commutative as well as left-idempotent insertion constructor, respectively. This is in compliance with our initial conjecture that a catamorphism must be indifferent to the confusion in the algebra it is applied to.

A check for the left-commutativity and left-idempotence of a constructor is undecidable in general [14, 20]. However, as the query processing context only requires a rather narrow and fixed spectrum of algebras, we can indicate the relevant properties of the involved constructors beforehand, thus freeing the system from the burden to deduce these. Actually, we find ourselves in an even better position: user-level queries will be mapped into the world of initial algebras and catamorphisms by means of an intermediate language, comprehensions (see Section 2.9), whose programs yield well-defined spine transformer compositions only.

We have now assembled everything that is needed to give the final specifications of the collection formers and their algebras that we will encounter in this text. All that is left is to put the pieces together.

51 Definition. Let $\mathsf{F}(E, \mathsf{T}E) = 1 + E \times \mathsf{T}E$. The **list**, **bag**, and **set collection type formers in insert representation** are defined by the following

initial types induced by F, respectively:

$$\begin{array}{ll} (nil^{\mathsf{list}} \triangledown cons^{\mathsf{list}} & , \mathsf{list}) \\ ((nil^{\mathsf{list}} \triangledown cons^{\mathsf{list}})/\textsc{Eq-Comm} & , \mathsf{bag}) \\ ((nil^{\mathsf{list}} \triangledown cons^{\mathsf{list}})/\textsc{Eq-Comm-Idem} & , \mathsf{set}) \ , \end{array}$$

where the equation Eq-Comm-Idem is given by

$$\left(\ L_{\mathrm{C}} \triangledown L_{\mathrm{I}}, R_{\mathrm{C}} \triangledown R_{\mathrm{I}} \ \right) \ , \qquad \textsc{Eq-Comm-Idem}$$

with $(L_{\mathrm{C}}, R_{\mathrm{C}}) =$ Eq-Comm and $(L_{\mathrm{I}}, R_{\mathrm{I}}) =$ Eq-Idem. Building the sum of the lhs and rhs of two equations like this retains Transformer (this is easily checked). The resulting equation forms the conjunction of the two original equations which is a direct consequence of the properties of categorical sum:

$$(L_{\mathrm{C}} \triangledown L_{\mathrm{I}}) = (R_{\mathrm{C}} \triangledown R_{\mathrm{I}}) \quad \Longleftrightarrow \quad L_{\mathrm{C}} = R_{\mathrm{C}} \wedge L_{\mathrm{I}} = R_{\mathrm{I}} \ .$$

We will use the shorthand $(nil^{\mathsf{bag}} \triangledown cons^{\mathsf{bag}}) = (nil^{\mathsf{list}} \triangledown cons^{\mathsf{list}})/$Eq-Comm to render the notation more compact. Analogously for set. //

2.8 Monads

52 Now that the gory details of the underlying categorical machinery have all been set, let us again take a bird's eye view to see how the theory developed so far fits into our overall idea of query processing.

Initial types and catamorphisms are the core concepts for *internal* query representation and manipulation in this text. In Chapter 3 we will thus face the challenge of providing a translation from an *external* query syntax into this formal representation. To be honest, the gap between a user-level SQL or OQL query and its equivalent expressed by means of catamorphisms or, more generally, spine transformers, seems to be quite wide.

Of the complications that will crop up, let us merely highlight two here:

- the presence of *variables* in user-level queries, and
- the arbitrarily deep *nesting* of query clauses.

A query representation based on algebras as we have chosen it in this text is an inherently variable-free formalism. It is not at all obvious how the binding and scoping of variables translates into a catamorphism-based algebraic equivalent, given that catamorphisms are—in the lingo of λ-calculus—*combinators*, *i. e.*, closed expressions that are not dependent on an environment of variables. The issue is indeed relevant: we will meet examples of

OQL queries where binding a formerly free variable completely changes the query semantics. We have to find ways to reflect this phenomenon.

Query nesting poses further problems. A composition of spine transformers is basically a *flat* sequence of operations. Nested queries have to be flattened or *unnested* prior to their translation into catamorphisms and type functors. We are thus better off to choose an internal query representation that facilitates such unnesting and related simplification procedures.

53 These observations suggested the employment of a query representation form that will act as a mediator between the user and the catamorphism-determined world.

It is one of the major claims of this text, that a very general query calculus notation, *monad comprehensions* [117][6], can successfully play the role of this intermediate representation [58, 60, 101]. Comprehensions feature variable binding and scoping much like the user query languages. Nesting is naturally expressed by means of comprehensions, and—by far more important for our purposes—comprehension expressions enjoy a *normal form* [38, 58] whose derivation actually implements the unnesting transformations we have just identified as a major problem source.

This notion of a query calculus is general in the sense that it is based upon *monads* [75, 79], algebras that offer just enough structure to interpret the constructs of a calculus in the style of the relational calculus [13, 117]. The initial types in insert representation induce monad instances, as we will shortly see. Putting the monad operations down to the now well-known expressions over initial algebras will ensure that the transition from monad comprehensions to catamorphisms is not too hard to define. We may indeed perceive monad comprehensions as mere syntactic sugar for their defining initial algebra expressions. Monad comprehensions, however, enable us to take the before mentioned hurdles with remarkable ease, which is the ultimate rationale behind their employment in this text.

Last but not least: the rather terse comprehension expressions—whose syntactic appearance closely resembles that of Zermelo-Fraenkel (ZF) expressions [61]—are more easily grasped by the human mind and eye than ever longer sequences of catamorphisms.

In what follows now, we will provide a monad notion based on our initial algebras and define the calculus language on top of it. The calculus will be put to full use in the upcoming chapters.

[6]The title of this thesis is actually a pun on the title of Wadler's seminal article *Comprehending Monads* [117].

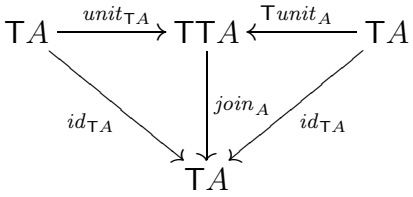
54 Definition. For an endofunctor T in a category $\mathbf{C}$ (as before, $\mathbf{C} = \mathbf{Set}$ will be the only instantiation of interest for us) we define a **T-monad** to mean a triple $(\mathsf{T}, \mathit{unit}_A : A \to \mathsf{T}A, \mathit{join}_A : \mathsf{T}\mathsf{T}A \to \mathsf{T}A)$ so that

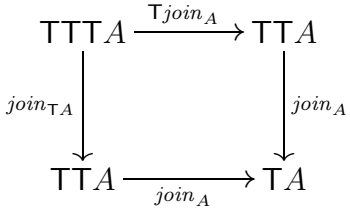
$$\mathit{join}_A \cdot \mathit{unit}_{\mathsf{T}A} = \mathit{id}_{\mathsf{T}A} \,, \qquad \text{Monad-1}$$

$$\mathit{join}_A \cdot \mathsf{T}\mathit{unit}_A = \mathit{id}_{\mathsf{T}A} \,, \qquad \text{Monad-2}$$

$$\mathit{join}_{\mathsf{T}A} \cdot \mathit{join}_A = \mathit{join}_A \cdot \mathsf{T}\mathit{join}_A \,. \qquad \text{Monad-3}$$

The involved types are more easily read off the corresponding diagrams:





If there is the danger of confusing the operations of different monads, we will indicate the monad as a superscript as in $\mathit{join}^{\mathsf{T}}$. Note that the arrow mapping part of functor T plays the role of the monadic $\mathit{map}^{\mathsf{T}}$ appearing in Wadler's monad definition [117]. //

55 In the above definition, subscript A ranges over all objects of $\mathbf{C}$, so that unit_A and join_A indeed define two families of arrows. For $\mathbf{C} = \mathbf{Set}$, this essentially amounts to saying that *unit* and *join* are polymorphic functions with types $\mathit{unit} : \forall A.A \to \mathsf{T}A$ and $\mathit{join} : \forall A.\mathsf{T}\mathsf{T}A \to \mathsf{T}A$.

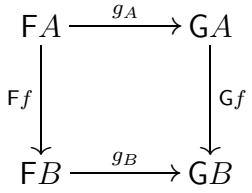
Due to *parametricity* [116], *i. e.*, only by looking at the polymorphic types of *unit* and *join*, we can tell that the two monadic operators satisfy the *free theorems* (let $f : A \to B$ in $\mathbf{C}$ be arbitrary):

$$\mathsf{T}f \cdot \mathit{unit}_A = \mathit{unit}_B \cdot f$$
$$\mathsf{T}f \cdot \mathit{join}_A = \mathit{join}_B \cdot \mathsf{T}\mathsf{T}f \,,$$

which, to a category theorist anyway, reveals *unit* and *join* to be *natural transformations* of types $\mathit{unit} : \mathsf{Id} \dot{\to} \mathsf{T}$ and $\mathit{join} : \mathsf{T}\mathsf{T} \dot{\to} \mathsf{T}$, respectively.

56 Definition. Given two functors of the same type $\mathsf{F}, \mathsf{G} : \mathbf{C} \to \mathbf{D}$, a family of arrows $g_A : \mathsf{F}A \to \mathsf{G}A$ (for each A in $\mathbf{C}$) forms a **natural transformation** from F to G (in symbols: $g : \mathsf{F} \dot{\to} \mathsf{G}$), if the *naturality* condition

$$\mathsf{G}f \cdot g_A = g_B \cdot \mathsf{F}f \qquad \textit{i. e.,}$$



Naturality

is met for each arrow $f : A \to B$ in $\mathbf{C}$. Naturality, at least to a certain extent, thus serves as a categorical encoding of the notion of parametric polymorphism. //

57 We will employ monad comprehensions as a *query calculus*, so that they clearly have to support predicates that can act as filters constraining an expression's result to those elements that satisfy the predicate.

The goal to support filters in monad comprehensions (see Section 2.9) will require us to work with slightly extended monads, namely *monads with zero* [117]. To complete the monad introduction, let us define this enriched monad concept here before we go straight on to derive the actual monads that will be useful in defining our query calculus.

58 Definition. If $(\mathsf{T}, \mathit{unit}, \mathit{join})$ denotes a T-monad in $\mathbf{C}$, we can derive a **T-monad with zero** $(\mathsf{T}, \mathit{unit}, \mathit{join}, \mathit{zero}_A : 1 \to \mathsf{T}A)$ from it, provided that

$$\mathit{join}_A \cdot \mathit{zero}_{\mathsf{T}A} = \mathit{zero}_A \qquad \text{MONAD-4}$$
$$\mathit{join}_A \cdot \mathsf{T}\mathit{zero}_A = \mathit{zero}_A \; . \qquad \text{MONAD-5}$$

As before, A ranges over the objects of $\mathbf{C}$. Operator *zero*, just like *unit* and *join*, defines a polymorphic function in **Set** and fulfills law NATURALITY (with $f : A \to B$) which is once more a consequence of parametricity (of *zero*):

$$\mathsf{T}f \cdot \mathit{zero}_A = \mathit{zero}_B \cdot f \; ,$$

i. e., $\mathit{zero} : \mathsf{K}_1 \dot{\to} \mathsf{T}$ (observe that the result of *zero* is independent of its argument). //

59 The step from an initial datatype (τ, T) in insert representation to a T-monad with zero is actually a small one, as we will now see. Following the construction outlined below we will in particular derive *collection monads*, *i. e.*, list, bag, and set monads according to the collection formers we defined in Definition 51.

60 Lemma. Any initial type $([\,]^{\mathsf{T}} \triangledown \vdots^{\mathsf{T}}, \mathsf{T})$ induced by the bifunctor $\mathsf{F}(f, g) = \mathit{id}_1 + f \times g$ gives rise to a T-monad $(\mathsf{T}, \mathit{unit}, \mathit{join}, \mathit{zero})$ with zero by defining

$$\begin{aligned} \mathit{unit} &= \vdots^{\mathsf{T}} \cdot (\mathit{id} \vartriangle [\,]^{\mathsf{T}}) \\ \mathit{join} &= (\!|\, [\,]^{\mathsf{T}} \triangledown +\!|) \\ \mathit{zero} &= [\,]^{\mathsf{T}} \end{aligned}$$

where function $+\!\!+ : \mathsf{T}A \times \mathsf{T}A \to \mathsf{T}A$ is defined to mean

$$xs +\!\!+ ys = (\!| ys \triangledown \overset{\mathsf{T}}{!} |\!)\, xs \ .$$

Proof. First, it is easily checked that the so defined monadic operators are correctly typed. Second, observe that *join* is well-defined regardless of T (respectively $\overset{\mathsf{T}}{!}$) being left-commutative or left-idempotent: as $+\!\!+$ merely concatenates the spines of its two arguments, $+\!\!+$ is commutative or idempotent whenever $\overset{\mathsf{T}}{!}$ is (in this case we will call the T-monad itself *commutative* or *idempotent*, respectively). This, in turn, makes Eq-Comm or Eq-Idem hold for the algebra $[\,]^{\mathsf{T}} \triangledown +\!\!+$, *i. e.*,

$$ys +\!\!+ (xs +\!\!+ zs) = xs +\!\!+ (ys +\!\!+ zs) \qquad \text{and} \qquad xs +\!\!+ (xs +\!\!+ zs) = xs +\!\!+ zs \ ,$$

which ensures the well-definedness of *join* (see Paragraph 50). Observe that $+\!\!+$ and *join* commute in the sense that

$$join \cdot +\!\!+ = +\!\!+ \cdot (join \times join) \ ,$$

which essentially is a consequence of the free theorem corresponding to the polymorphic type of $+\!\!+$ [116]. We additionally have that $[\,]^{\mathsf{T}}$ is a left and right unit for $+\!\!+$:

$$\begin{array}{ll} & [\,]^{\mathsf{T}} +\!\!+ xs \\ = & \quad \{\, \text{unfold } +\!\!+ \,\} \\ & (\!| xs \triangledown \overset{\mathsf{T}}{!} |\!)\, [\,]^{\mathsf{T}} \\ = & \quad \{\, \textsc{Cata-Ins-Rep} \,\} \\ & xs \end{array} \qquad \begin{array}{ll} & xs +\!\!+ [\,]^{\mathsf{T}} \\ = & \quad \{\, \text{unfold } +\!\!+ \,\} \\ & (\!| [\,]^{\mathsf{T}} \triangledown \overset{\mathsf{T}}{!} |\!)\, xs \\ = & \quad \{\, \textsc{Cata-Reflect} \,\} \\ & xs \ . \end{array}$$

The main work we have to do here, namely the verification of the monad laws Monad-1 through Monad-5, is carried out calculationally. At several points we will make good use of the catamorphism fusion laws Cata-Fusion and Cata-Map-Fusion to shorten the calculations.

Monad-1:

$$\begin{array}{ll} & join \cdot unit \\ = & \quad \{\, \text{unfold monadic operators} \,\} \\ & (\!| [\,]^{\mathsf{T}} \triangledown +\!\!+ |\!) \cdot (\overset{\mathsf{T}}{!} \cdot (id \triangledown [\,]^{\mathsf{T}})) \\ = & \quad \{\, \textsc{Cata-Ins-Rep} \,\} \end{array}$$

$$
\begin{array}{ll}
& +\!\!+ \cdot (id \triangledown [\,]^{\mathsf{T}}) \\
= & \quad \{\, [\,]^{\mathsf{T}} \text{ is unit for } +\!\!+ \,\} \\
& id \ .
\end{array}
$$

Monad-2:

$$
\begin{array}{ll}
& join \cdot \mathsf{T} unit \\
= & \quad \{\, \text{unfold, Cata-Map-Fusion} \,\} \\
& (\!|([\,]^{\mathsf{T}} \triangledown +\!\!+) \cdot \mathsf{F}(unit, id)|\!) \\
= & \quad \{\, \text{unfold } \mathsf{F} \,\} \\
& (\!|([\,]^{\mathsf{T}} \triangledown +\!\!+) \cdot (id + (unit \times id))|\!) \\
= & \quad \{\, \text{sum} \,\} \\
& (\!|[\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (unit \times id))|\!) \\
= & \quad \{\, \text{unfold } unit \,\} \\
& (\!|[\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot ((\overset{\mathsf{T}}{!} \cdot (id \triangledown [\,]^{\mathsf{T}})) \times id))|\!) \\
= & \quad \{\, +\!\!+ \text{ is catamorphism on its first argument} \,\} \\
& (\!|[\,]^{\mathsf{T}} \triangledown (\overset{\mathsf{T}}{!} \cdot (id \times (+\!\!+ \cdot ([\,]^{\mathsf{T}} \times id))))|\!) \\
= & \quad \{\, [\,]^{\mathsf{T}} \text{ is unit for } +\!\!+ \,\} \\
& (\!|[\,]^{\mathsf{T}} \triangledown (\overset{\mathsf{T}}{!} \cdot (id \times id))|\!) \\
= & \quad \{\, \text{product, Cata-Reflect} \,\} \\
& id \ .
\end{array}
$$

Monad-3:

$$
\begin{array}{ll}
& join \cdot join = join \cdot \mathsf{T} join \\
\equiv & \quad \{\, \text{Cata-Map-Fusion} \,\} \\
& join \cdot join = (\!|([\,]^{\mathsf{T}} \triangledown +\!\!+) \cdot (id + (join \times id))|\!) \\
\equiv & \quad \{\, \text{sum} \,\} \\
& join \cdot join = (\!|[\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (join \times id))|\!) \\
\underset{(*)}{\equiv} & \quad \{\, \text{Cata-Fusion} \,\} \\
& true \ .
\end{array}
$$

To see the applicability of Cata-Fusion at $(*)$, observe that $join$ is an F-homomorphism from $[\,]^{\mathsf{T}} \triangledown +\!\!+$ to $[\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (join \times id))$:

$$
join \cdot ([\,]^{\mathsf{T}} \triangledown +\!\!+) = ([\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (join \times id))) \cdot \mathsf{F}(id, join)
$$

$$
\begin{array}{ll}
\equiv & \{\,\text{unfold } \mathsf{F}\,\} \\
& \mathit{join} \cdot ([\,]^{\mathsf{T}} \triangledown +\!\!+) = ([\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (\mathit{join} \times \mathit{id}))) \cdot (\mathit{id} + (\mathit{id} \times \mathit{join})) \\
\equiv & \{\,\text{sum}\,\} \\
& \mathit{join} \cdot ([\,]^{\mathsf{T}} \triangledown +\!\!+) = [\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (\mathit{join} \times \mathit{join})) \\
\equiv & \{\,\text{sum}\,\} \\
& (\mathit{join} \cdot [\,]^{\mathsf{T}}) \triangledown (\mathit{join} \cdot +\!\!+) = [\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot (\mathit{join} \times \mathit{join})) \\
\equiv & \{\,\mathit{join} \text{ is catamorphism, free theorem for } +\!\!+\,\} \\
& \mathit{true}\ .
\end{array}
$$

MONAD-4:

$$
\begin{array}{ll}
& \mathit{join} \cdot \mathit{zero} \\
= & \{\,\text{unfold}\,\} \\
& (\!|\, [\,]^{\mathsf{T}} \triangledown +\!\!+ \,|\!) \cdot [\,]^{\mathsf{T}} \\
= & \{\,\text{CATA-INS-REP}\,\} \\
& [\,]^{\mathsf{T}} \\
= & \{\,\text{definition of } \mathit{zero}\,\} \\
& \mathit{zero}\ .
\end{array}
$$

MONAD-5:

$$
\begin{array}{ll}
& \mathit{join} \cdot \mathsf{T}\mathit{zero} \\
= & \{\,\text{unfold, CATA-MAP-FUSION}\,\} \\
& (\!|\, ([\,]^{\mathsf{T}} \triangledown +\!\!+) \cdot \mathsf{F}(\mathit{zero}, \mathit{id}) \,|\!) \\
= & \{\,\text{unfold } \mathsf{F}, \mathit{zero}\,\} \\
& (\!|\, ([\,]^{\mathsf{T}} \triangledown +\!\!+) \cdot (\mathit{id} + ([\,]^{\mathsf{T}} \times \mathit{id})) \,|\!) \\
= & \{\,\text{sum}\,\} \\
& (\!|\, [\,]^{\mathsf{T}} \triangledown (+\!\!+ \cdot ([\,]^{\mathsf{T}} \times \mathit{id})) \,|\!) \\
= & \{\,[\,]^{\mathsf{T}} \text{ is unit for } +\!\!+\,\} \\
& (\!|\, [\,]^{\mathsf{T}}, \mathit{outr} \,|\!) \\
= & \{\,\text{CATA-INS-REP}\,\} \\
& [\,]^{\mathsf{T}} \\
= & \{\,\text{definition of } \mathit{zero}\,\} \\
& \mathit{zero}\ .
\end{array}
$$

□

61 Collection monads. Applying the above procedure to the three collection formers of Definition 51 constructs three *collection monads*. In **Set**, the category in which we model the query evaluation process, we specifically get

$$\begin{aligned}(\mathsf{list}, \mathit{unit}^{\mathsf{list}}, \mathit{join}^{\mathsf{list}}, \mathit{zero}^{\mathsf{list}}) &= (\mathsf{list}, \lambda x.[x], \mathit{flatten}, [\,]) \\ (\mathsf{bag}, \mathit{unit}^{\mathsf{bag}}, \mathit{join}^{\mathsf{bag}}, \mathit{zero}^{\mathsf{bag}}) &= (\mathsf{bag}, \lambda x.\{\!\{x\}\!\}, \biguplus, \{\!\{\}\!\}) \\ (\mathsf{set}, \mathit{unit}^{\mathsf{set}}, \mathit{join}^{\mathsf{set}}, \mathit{zero}^{\mathsf{set}}) &= (\mathsf{set}, \lambda x.\{x\}, \bigcup, \varnothing)\ ,\end{aligned}$$

where square brackets $[\cdot]$ and double curly brackets $\{\!\{\cdot\}\!\}$ are used to denote list and bag values, respectively. Function $\mathit{flatten} : \mathsf{list}\,\mathsf{list}A \to \mathsf{list}A$ concatenates a list of lists into a flat list as in

$$\mathit{flatten}\,\big[[x_1, x_2], [\,], [x_3, x_4, x_5], [x_6]\big] = \big[x_1, x_2, x_3, x_4, x_5, x_6\big]\ .$$

The natural transformations $\bigcup : \mathsf{set}\,\mathsf{set}A \to \mathsf{set}A$ and $\biguplus : \mathsf{bag}\,\mathsf{bag}A \to \mathsf{bag}A$ flatten a set of sets and a bag of bags in much the same manner. These three monads provide the primary collection abstraction in the discussion ahead.

62 Our use of monads, however, goes further than that. Although the name *insert representation* might suggest so, we can perfectly imagine algebras $e \bigtriangledown \otimes$ in insert representation that do *not* model collection type constructors. Instead, we can instantiate e and $\otimes$ in a way that provides us with *aggregation operators* or *quantifiers*. Everything we have said about such algebras until now remains valid, with the difference that a spine over these algebras describes an aggregation or quantification and does not construct a collection value. Consequently, these algebras operate on carrier sets which correspond to *atomic types*. The monads associated with these non-collection algebras then embed aggregation and quantification seamlessly into the soon to be defined monad comprehension calculus (see Section 2.9). Much of the elegance of the approach devised in this text is derived from this uniformity.

63 So let E denote the domain of an atomic type like *Num* or *Bool*; additionally let the identity functor Id instantiate the type functor T. The endofunctor describing the insert representation constructors $\mathsf{F}_E = \mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id}$ remains unchanged. We obtain a free non-collection algebra over carrier E with constructors $e : 1 \to E$ and $\otimes : E \times E \to E$ (*cf.* Paragraph 19):

$$\begin{aligned} & e \bigtriangledown \otimes : \mathsf{F}_E(\mathsf{T}E) \to \mathsf{T}E \\ = \quad & \{\,\text{unfold } \mathsf{F}, \mathsf{T}\,\} \\ & e \bigtriangledown \otimes : 1 + E \times E \to E\ . \end{aligned}$$

64 Definition. Let $\mathsf{F}_E = \mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id}$ and $e \triangledown \otimes$ as in the previous paragraph. We define **aggregation operators** and **quantifiers in insert representation** by means of the following F_{Num} and F_{Bool} algebras, respectively (ignore the functor annotations until the next paragraph):

$$\begin{array}{lcll}
(0 \triangledown +) & = & (e \triangledown \otimes)/\text{EQ-COMM} & (\mathsf{sum}) \\
(1 \triangledown *) & = & (e \triangledown \otimes)/\text{EQ-COMM} & (\mathsf{prod}) \\
(\infty_{Num} \triangledown min_2) & = & (e \triangledown \otimes)/\text{EQ-COMM-IDEM} & (\mathsf{min}) \\
(-\infty_{Num} \triangledown max_2) & = & (e \triangledown \otimes)/\text{EQ-COMM-IDEM} & (\mathsf{max}) \\
(true \triangledown \wedge) & = & (e \triangledown \otimes)/\text{EQ-COMM-IDEM} & (\mathsf{all}) \\
(false \triangledown \vee) & = & (e \triangledown \otimes)/\text{EQ-COMM-IDEM} & (\mathsf{exists})\ .
\end{array}$$

The binary operator min_2 (max_2) returns the smaller (greater) of its two arguments of type Num. Symbol ∞_{Num} shall be perceived as the largest value representable in a concrete implementation of type Num, so that ∞_{Num} acts a unit with respect to min_2. A dual remark applies to $-\infty_{Num}$ and max_2. //

65 As the above algebras are not initial, we cannot mimic the construction of Lemma 60 to derive their monads. There is, however, a monad naturally associated with $e \triangledown \otimes$ and the functor Id, namely

$$\begin{aligned}
unit &= id \\
join &= id \\
zero &= e\ .
\end{aligned}$$

It is easily checked that $(\mathsf{Id}, unit, join, zero)$ satisfies laws MONAD-1 through MONAD-5 and thus indeed constitutes a monad, namely, adopting a term used by Wadler in [117], an *identity monad* with zero. In the following, we will use the functor aliases listed in Definition 64 to refer to the identity monads associated with the non-collection algebras, *e. g.*, $\mathsf{exists} = (\mathsf{Id}, id, id, false)$.

2.9 Monad Comprehensions

66 We are finally ready to keep our promise given in the introduction of this chapter and take a step towards a higher-level query representation, *monad comprehensions* [19, 60, 91, 117, 124]. Due to their syntactic resemblance with the set comprehension notation and thus with the relational calculus, monad comprehension should be particularly accessible to readers used to

these classical query notations. Of the things to come, much may indeed be grasped by applying knowledge acquired in the relational (tuple) calculus domain, although one is trading generality and uniformity for simplicity then.

To start off with a simple example consider the following comprehension to be interpreted in the set monad

$$[\![\, x \mid x \leftarrow xs, x \neq 0 \,]\!]^{\mathsf{set}} ,$$

which is equivalent to the set comprehension $\{x : x \in xs \wedge x \neq 0\}$ ("*the set of all x, so that $x \in xs$ and $x \neq 0$*"). While $x \in xs$ acts as pool from which successive bindings for variable x are generated (exactly one binding for each element in the set xs), predicate $x \neq 0$ restricts the final result set to include only those x that fulfill the specified condition.

More generally, in the T-monad comprehension

$$[\![\, e \mid q_1, \ldots, q_n \,]\!]^{\mathsf{T}} ,$$

the *qualifiers* q_i are either *generators* $x_i \leftarrow e_i$ or *filters* (expressions of result type *Bool*).

The informal semantics of this expression are as follows: starting with qualifier q_1, a generator $q_i = x_i \leftarrow e_i$ enriches an initially empty environment of variable bindings by (sequentially) binding x_i to (elements of) its *range* e_i. (If the range of a generator indeed is of an atomic type, the generator is essentially equivalent to a *let*-expression, see Example 71). The binding of x is propagated through the list of qualifiers $q_{i+1}, \ldots, q_n$. Filters, as they are encountered, inhibit further propagation if they evaluate to *false* in the current environment. *Head* e is evaluated in those environments that pass all qualifiers. The results are then injected into the T-monad using $unit^{\mathsf{T}}$ and finally accumulated via $join^{\mathsf{T}}$.

Let us state the monad comprehension syntax more precisely before we supersede the above informal semantics by a mapping that translates a T-monad comprehension into a computation over monad T.

67 Definition. Let the non-terminal symbol *expr* represent the terms of some basic expression language (think of a variant of the λ-calculus). This language is enriched by **monad comprehension syntax** through the following

BNF grammar with start symbol e:

e	$\rightarrow$	mc		expressions
	$\mid$	*expr*		
mc	$\rightarrow$	$[\![\, e \mid qs \,]\!]^{\mathsf{T}}$		monad comprehensions
qs	$\rightarrow$	ε		empty
	$\mid$	q		qualifiers
	$\mid$	qs, qs		qualifier lists
q	$\rightarrow$	$v \leftarrow e$		generators (v identifier)
	$\mid$	e		filters
T	$\rightarrow$	set $\mid$ bag $\mid$ list $\mid$ exists $\mid$ all		monads
	$\mid$	sum $\mid$ prod $\mid$ max $\mid$ min		

Note that the grammar allows for the arbitrary nesting of monad comprehensions. The occurrence of a comprehension as a generator range, predicate, or head in an enclosing comprehension will allows us to express the diverse types of query nesting found in user-level query languages. //

68 Interestingly, we can give a translation function for the desugaring of a T-monad comprehension to an equivalent computation in the T-monad that is guided by the comprehension *syntax* we have just defined. The translation scheme employs a recursion over the syntactic structure of the qualifier list, arbitrarily subdividing it at each recursion level until we are facing qualifier lists of length one or zero only. The method is originally due to Wadler [117] (and the translation also being known as the *Wadler identities*).

Our variant includes a considerable generalization of the idea in that it allows for the translation of nested comprehensions over *different monads*—a characteristic that, to the best of our knowledge, has not appeared before in the monad comprehension literature [113, 114, 117, 124]. The comprehensions of Wadler's original work were *endo-monadic* in the sense that were evaluated in a single monad only [117]. Working with multi-monad comprehensions is crucial if they are to be useful as a query calculus built for query languages featuring an orthogonal type system of collection constructors as well as an orthogonal syntactic structure ("*subqueries may occur everywhere as long as typing rules are obeyed*").

The key to this generalization is the catamorphism-based coercion of monadic types in branch $\mathscr{M}$-3 of the following definition.

69 Definition. Let τ denote an algebra in insert representation and be $(\mathsf{T}, unit^{\mathsf{T}}, join^{\mathsf{T}}, zero^{\mathsf{T}})$ its associated monad. Let $\mathsf{T}' \neq \mathsf{Id}$ denote another

monad. We define the **T-monad comprehension desugaring scheme** $\mathscr{M}$ (the stylized M serves as a mnemonic for *monad*) to be given by the recursive function below. The occurring identifiers are chosen to fit with those in the abstract comprehension syntax of Definition 67 and represent their respective syntactic category, *e. g.* *qs* may be replaced by an arbitrary qualifier list.

$$\begin{array}{lcll}
\mathscr{M}\,[\![\, e \,|\,]\!]^{\mathsf{T}} & = & \mathit{unit}^{\mathsf{T}}\,(\mathscr{M}\,e) & \mathscr{M}\text{-}1 \\
\mathscr{M}\,[\![\, e \,|\, v \leftarrow e' \!:\! \mathsf{T}E \,]\!]^{\mathsf{T}} & = & \mathsf{T}(\lambda v.\mathscr{M}\,e)\,(\mathscr{M}\,e') & \mathscr{M}\text{-}2 \\
\mathscr{M}\,[\![\, e \,|\, v \leftarrow e' \!:\! \mathsf{T}'E \,]\!]^{\mathsf{T}} & = & (\!|\tau|\!)\,(\mathscr{M}\,[\![\, e \,|\, v \leftarrow e' \,]\!]^{\mathsf{T}'}) & \mathscr{M}\text{-}3 \\
\mathscr{M}\,[\![\, e \,|\, e' \,]\!]^{\mathsf{T}} & = & \mathit{if}\ (\mathscr{M}\,e')\ \mathit{then}\ \mathit{unit}^{\mathsf{T}}\,(\mathscr{M}\,e)\ \mathit{else}\ \mathit{zero}^{\mathsf{T}} & \mathscr{M}\text{-}4 \\
\mathscr{M}\,[\![\, e \,|\, qs, qs' \,]\!]^{\mathsf{T}} & = & \mathit{join}^{\mathsf{T}}\ (\mathscr{M}\,[\![\, [\![\, e \,|\, qs' \,]\!]^{\mathsf{T}} \,|\, qs \,]\!]^{\mathsf{T}}) & \mathscr{M}\text{-}5 \\
\mathscr{M}\,e & = & e & \mathscr{M}\text{-}6
\end{array}$$

∎

70 Notes. (a) Note that, according to branch $\mathscr{M}$-5, we can arbitrarily divide the comprehension qualifier list into two sublists *qs* and *qs*′ without affecting the outcome of $\mathscr{M}$. This is a consequence of qualifier enumeration (denoted by ',') being an associative operation with the empty qualifier list as its unit [117].

(b) The argument to type functor T in case $\mathscr{M}$-2 is an anonymous λ-term, *i. e.*, a morphism in category **Set**. Consequently, scheme $\mathscr{M}$ is bound to the assumption that we are operating with **Set** as the underlying category.

(c) Whenever we encounter a generator whose range is of a foreign monad $\mathsf{T}' \neq \mathsf{T}$, branch $\mathscr{M}$-3 temporarily switches the monad in which the monad comprehension is to be interpreted in to T'. The results are then coerced to type T using $(\!|\tau|\!)$. Observe that the cases $\mathscr{M}$-2 and $\mathscr{M}$-3 could be merged to read (with T' not necessarily different to T)

$$\mathscr{M}\,[\![\, e \,|\, v \leftarrow e' \!:\! \mathsf{T}'E \,]\!]^{\mathsf{T}} \quad = \quad (\!|\tau|\!)\,(\mathsf{T}'(\lambda v.\mathscr{M}\,e)\,(\mathscr{M}\,e'))\ , \qquad \mathscr{M}\text{-}2'$$

which amounts to the same due to law Cata-Reflect.

Coercion is not completely arbitrary though since the well-definedness condition for catamorphisms derived in Paragraph 50 applies: whenever τ' is a left-commutative respective left-idempotent algebra, so has to be τ. This restriction is quite natural, however, as it forbids ambiguous and non-well-defined coercions, like the non-deterministic conversion of set value into a list.

(d) Note how branch $\mathscr{M}$-4 implements the filter q by mapping values failing to fulfill the filter predicate to $\mathit{zero}^{\mathsf{T}}$ so that they do not contribute to the result in the embracing $\mathit{join}^{\mathsf{T}}$ (see Monad-4).

(e) The exchange of qualifiers in $\mathscr{M}$-5, in interplay with the introduction of λ-abstractions for each generator $v \leftarrow e'$ in $\mathscr{M}$-2, ensures the visibility of v in each qualifier occurring later in the qualifier list as well as in the head e.

71 Example. A rather broad diversity of computational tasks may be expressed by monad comprehensions and it is the crucial benefit of employing the comprehension notation that this variety is mapped to a completely uniform representation. Quite often we will be able to abstract from the specific monad the query is to be evaluated in and instead reason in the context of *some* monad T. Such general findings may then be reused during the query translation process in several instantiations and places.

The following examples are, in some sense, teasers that are given here to provide some insight of what the upcoming chapters have in store.

(a) We can now convince ourselves that the initial example of this section, the set comprehension $[\![\, x \mid x \leftarrow xs, x \neq 0 \,]\!]^{\mathsf{set}}$, is indeed equivalent to $\{x : x \in xs \wedge x \neq 0\}$. By means of desugaring we get

$$
\begin{aligned}
& \mathscr{M}\,[\![\, x \mid x \leftarrow xs, x \neq 0 \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-}5 \,\} \\
& join^{\mathsf{set}}\left(\mathscr{M}\,[\![\, [\![\, x \mid x \neq 0 \,]\!]^{\mathsf{set}} \mid x \leftarrow xs \,]\!]^{\mathsf{set}}\right) \\
= \quad & \{\, \mathscr{M}\text{-}2 \,\} \\
& join^{\mathsf{set}}\left(\mathsf{set}\,(\lambda x.\mathscr{M}\,[\![\, x \mid x \neq 0 \,]\!]^{\mathsf{set}})\; xs\right) \\
= \quad & \{\, \mathscr{M}\text{-}4 \,\} \\
& join^{\mathsf{set}}\,(\mathsf{set}\,(\lambda x.\mathit{if}\;(\mathscr{M}\;(x \neq 0))\;\mathit{then}\;(unit^{\mathsf{set}}\,(\mathscr{M}\;x))\;\mathit{else}\;zero^{\mathsf{set}})\;xs) \\
= \quad & \{\, \mathscr{M}\text{-}6 \,\} \\
& join^{\mathsf{set}}\,(\mathsf{set}\,(\lambda x.\mathit{if}\;(x \neq 0)\;\mathit{then}\;(unit^{\mathsf{set}}\;x)\;\mathit{else}\;zero^{\mathsf{set}})\;xs) \\
= \quad & \{\, \mathsf{set}\text{ monad} \,\} \\
& \bigcup\,(\mathsf{set}\,(\lambda x.\mathit{if}\;(x \neq 0)\;\mathit{then}\;\{x\}\;\mathit{else}\;\varnothing)\;xs)\;\;.
\end{aligned}
$$

The resulting set-monad computation implements the set comprehension by mapping each element x of xs to the singleton set $\{x\}$ if $x \neq 0$. Otherwise x is mapped to the empty set $\varnothing$ so that these elements do not contribute to the result during the outer $\bigcup$.

(b) Monad comprehensions over monad all effectively implement universal quantification. Consider the all-monad comprehension expression $[\![\, p\,x \mid x \leftarrow xs : \mathsf{set} E \,]\!]^{\mathsf{all}}$ in which p denotes a predicate of type $E \rightarrow Bool$. Observe that this already constitutes a multi-monad comprehension which was not expressible by earlier related proposals. We calculate as follows:

$$\mathscr{M}\,[\![\, p\,x \mid x \leftarrow xs : \mathsf{set} E \,]\!]^{\mathsf{all}}$$

$$
\begin{array}{ll}
= & \quad \{\, \mathscr{M}\text{-}3 \,\} \\
& (\!|\, \mathit{true} \triangledown \wedge \,|\!)\, (\mathscr{M}\, [\![\, p\, x \mid\!]\, x \leftarrow \mathit{xs} : \mathsf{set} E \,]\!]^{\mathsf{set}}) \\
= & \quad \{\, \mathscr{M}\text{-}2 \,\} \\
& (\!|\, \mathit{true} \triangledown \wedge \,|\!)\, ((\mathsf{set}\, (\lambda x.\mathscr{M}\, (p\, x))\, (\mathscr{M}\, \mathit{xs}))) \\
= & \quad \{\, \mathscr{M}\text{-}6, \eta\text{-conversion} \,\} \\
& (\!|\, \mathit{true} \triangledown \wedge \,|\!)\, ((\mathsf{set}\, p\, \mathit{xs})) \; .
\end{array}
$$

The resulting expression obviously computes $\forall x \in \mathit{xs} : p\, x$. Note that this also does the right thing if $\mathit{xs} = \mathit{zero}^{\mathsf{set}} = \varnothing$ in which case we get *true* as desired.

(c) Monad comprehensions are not bound to denote *collection comprehensions* (as they are understood in [113, 124]). Here is an example of a non-collection comprehension to be interpreted in the sum monad:

$$
\begin{array}{ll}
& \mathscr{M}\, [\![\, (x, y) \mid\!]\, x \leftarrow 2, y \leftarrow 3 \,]\!]^{\mathsf{sum}} \\
= & \quad \{\, \mathscr{M}\text{-}5 \,\} \\
& \mathit{join}^{\mathsf{sum}}\, (\mathscr{M}\, [\![\, [\![\, (x, y) \mid\!]\, y \leftarrow 3 \,]\!]^{\mathsf{sum}} \mid\!]\, x \leftarrow 2 \,]\!]^{\mathsf{sum}}) \\
= & \quad \{\, \mathscr{M}\text{-}2 \,\} \\
& \mathit{join}^{\mathsf{sum}}\, (\mathsf{sum}\, (\lambda x.(\mathscr{M}\, [\![\, (x, y) \mid\!]\, y \leftarrow 3 \,]\!]^{\mathsf{sum}})\, 2)) \\
= & \quad \{\, \mathsf{sum}\ \text{monad} \,\} \\
& \mathit{join}^{\mathsf{sum}}\, (\mathsf{sum}\, (\lambda x.(\mathsf{sum}\, (\lambda y.\mathscr{M}\, (x, y))\, 3))\, 2) \\
= & \quad \{\, \mathsf{sum}\ \text{monad} \,\} \\
& \lambda x.((\lambda y.(x, y))\, 3)\, 2 \; .
\end{array}
$$

The final outcome reveals that the sum comprehension boils down to a nested *let*-expression:

$$\mathit{let}\, x = 2\, \mathit{in}\, \mathit{let}\, y = 3\, \mathit{in}\, (x, y) \; .$$

Note that we could compute the above in any identity monad and that the actual choice of sum is arbitrary.

(d) Let us conclude this introduction to monad comprehensions with a more substantial example. The occurrence of a nested exists-comprehension inside an outer sum-comprehension does not violate the well-definedness condition for multi-monad comprehensions, as the inner comprehension serves as a filter (comprehensions interpreted in monads exists and all reduce to values of type *Bool*) and not as a generator range. We assume the typings $\mathit{xs} : \mathsf{list} \mathit{Num}$ and $\mathit{ys} : \mathsf{bag} \mathit{Num}$.

$$\mathscr{M}\, [\![\, x \mid\!]\, x \leftarrow \mathit{xs}, [\![\, x = y \mid\!]\, y \leftarrow \mathit{ys} \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{sum}}$$

$$
\begin{array}{ll}
= & \{\, \mathscr{M}\text{-}5 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,(\mathscr{M}\,[\![\,[\![\,x \mid [\![\,x = y \mid y \leftarrow ys\,]\!]^{\mathsf{exists}}\,]\!]^{\mathsf{sum}} \mid x \leftarrow xs\,]\!]^{\mathsf{sum}}) \\
= & \{\, \mathscr{M}\text{-}3 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,((\!|0 \triangledown +|\!)\,(\mathscr{M}\,[\![\,[\![\,x \mid [\![\,x = y \mid y \leftarrow ys\,]\!]^{\mathsf{exists}}\,]\!]^{\mathsf{sum}} \mid x \leftarrow xs\,]\!]^{\mathsf{list}})) \\
= & \{\, \mathscr{M}\text{-}2 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,((\!|0 \triangledown +|\!)\,(\mathsf{list}\,(\lambda x.\mathscr{M}\,[\![\,x \mid [\![\,x = y \mid y \leftarrow ys\,]\!]^{\mathsf{exists}}\,]\!]^{\mathsf{sum}})\,xs)) \\
= & \{\, \mathscr{M}\text{-}4 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,((\!|0 \triangledown +|\!)\,(\mathsf{list}\,(\lambda x.\mathit{if}\ (\mathscr{M}\,[\![\,x = y \mid y \leftarrow ys\,]\!]^{\mathsf{exists}}) \\
& \qquad\qquad\qquad\qquad \mathit{then}\,(\mathit{unit}^{\mathsf{sum}}\,x)\ \mathit{else}\ \mathit{zero}^{\mathsf{sum}})\,xs)) \\
= & \{\, \mathscr{M}\text{-}3 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,((\!|0 \triangledown +|\!)\,(\mathsf{list}\,(\lambda x.\mathit{if}\ ((\!|\mathit{false} \triangledown \vee|\!)\,(\mathscr{M}\,[\![\,x = y \mid y \leftarrow ys\,]\!]^{\mathsf{bag}})) \\
& \qquad\qquad\qquad\qquad \mathit{then}\,(\mathit{unit}^{\mathsf{sum}}\,x)\ \mathit{else}\ \mathit{zero}^{\mathsf{sum}})\,xs)) \\
= & \{\, \mathscr{M}\text{-}2 \,\} \\
& \mathit{join}^{\mathsf{sum}}\,((\!|0 \triangledown +|\!)\,(\mathsf{list}\,(\lambda x.\mathit{if}\ ((\!|\mathit{false} \triangledown \vee|\!)\,(\mathsf{bag}\,(\lambda y.x = y)\,ys)) \\
& \qquad\qquad\qquad\qquad \mathit{then}\,(\mathit{unit}^{\mathsf{sum}}\,x)\ \mathit{else}\ \mathit{zero}^{\mathsf{sum}})\,xs))\ .
\end{array}
$$

The point to note here is that this expressions shows the typical structure we expected a query to exhibit in our representation form. Unfolding the monadic operators discloses the initial monad comprehension to be completely implemented by means of catamorphisms and type functors:

$$(\!|0 \triangledown +|\!)\,(\mathsf{list}\,(\lambda x.\mathit{if}\ ((\!|\mathit{false} \triangledown \vee|\!)\,(\mathsf{bag}\,(\lambda y.x = y)\,ys))\ \mathit{then}\,x\ \mathit{else}\,0)\,xs)\ . \qquad \Diamond$$

We have reached a point where the foundations for the internal query representation used in this text have all been laid. The forthcoming chapters will put this categorical toolbox to, as we will argue, beneficial use in query translation, transformation, and optimization.

2.10 Union Representation

72 Although the insert representation of algebras has been pervasive in everything we have developed so far, it is not principle to the method. In fact there is at least one alternative class of algebras, the *algebras in union representation*, which has been extensively studied as an alternative foundation for collection (query and programming) languages [13, 14, 20, 110, 111].

A collection value in union representation is built from *singleton collections* which are then merged using an *associative binary union operator*

$\oplus$. The union representation comes with an additional constant constructor (function), the *empty collection* z. This last constructor is required to be a unit of the binary union so that $(z, \oplus)$ forms a *monoid*. In other words, general binary trees take the place of the right-deep insertion spines we are employing in this text. The elements in the front of these trees are the collection members.

Remarkably few details have to be adapted to turn the material of this chapter into a discussion of the union representation of types. As we have encoded the type constructors by means of the insert representation functor F, it is only this functor that is to be replaced. In principle, everything else may be left untouched! Let us take the union representation view of the theory of datatypes for a minute.

Given a polymorphic type constructor T, its G-algebra in union representation is

$$\mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \triangledown \mathit{union}^{\mathsf{T}} : 1 + E + \mathsf{T}E \times \mathsf{T}E \to \mathsf{T}E \ ,$$

i. e., $\mathsf{G}(E, \mathsf{T}E) = 1 + E + \mathsf{T}E \times \mathsf{T}E$ and $\mathsf{G}(f, g) = \mathit{id}_1 + f + g \times g$.

Obvious instances of this generic algebra, for $\mathsf{T} = \mathsf{set}$ and $\mathsf{T} = \mathsf{list}$ respectively, include $\varnothing \triangledown \lambda x.\{x\} \triangledown \cup$ and $[\,] \triangledown \lambda x.[x] \triangledown +\!\!+$. Setting $\mathsf{T} = \mathsf{Id}$ yields non-collection instances (with $\mathit{sng}^{\mathsf{Id}} = \mathit{id}$) as in $0 \triangledown \mathit{id} \triangledown +$ and $\mathit{true} \triangledown \mathit{id} \triangledown \wedge$.

As before, $\mathbf{Alg}(\mathsf{G})$ has an initial object τ and it this initial algebra that describes the datatype. The catamorphisms in $\mathbf{Alg}(\mathsf{G})$ turn out to be general *tree transformers*: let $z \triangledown s \triangledown \oplus$ denote any other G-algebra so that $(z, \oplus)$ has the monoid property, then we get (*cf.* Paragraph 27)

$$\begin{array}{lll}
(\!|\, z \triangledown s \triangledown \oplus \,|\!)\, \mathit{empty}^{\mathsf{T}} & = & z \\
\wedge\ (\!|\, z \triangledown s \triangledown \oplus \,|\!)\, (\mathit{sng}^{\mathsf{T}}\, x) & = & s\, x \\
\wedge\ (\!|\, z \triangledown s \triangledown \oplus \,|\!)\, (\mathit{union}^{\mathsf{T}}(\mathit{xs}, \mathit{ys})) & = & ((\!|\, z \triangledown s \triangledown \oplus \,|\!)\, \mathit{xs}) \oplus ((\!|\, z \triangledown s \triangledown \oplus \,|\!)\, \mathit{ys}) \ .
\end{array}$$

CATA-UNION-REP

Not surprisingly, the induced type functor T behaves like a *tree map*, since

$$\begin{array}{ll}
 & \mathsf{T}f \\
= & \quad \{\, \text{TYPE-FUNCTOR} \,\} \\
 & (\!|\, \tau \cdot \mathsf{G}(f, \mathit{id}) \,|\!) \\
= & \quad \{\, \text{unfold } \tau \text{ and union representation functor } \mathsf{G} \,\} \\
 & (\!|\, (\mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \triangledown \mathit{union}^{\mathsf{T}}) \cdot (\mathit{id} + f + \mathit{id} \times \mathit{id}) \,|\!) \\
= & \quad \{\, \text{sum} \,\} \\
 & (\!|\, \mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \cdot f \triangledown \mathit{union}^{\mathsf{T}} \,|\!) \ ,
\end{array}$$

which tells us that $\mathsf{T}f$ is the specific tree transformer that applies f to the elements of the front of its argument while retaining the structure of the tree.

From the above it is apparent that the G-catamorphisms $(\!|\cdot|\!)_{\mathsf{G}}$ generalize the structural recursion operator over collections in union representation *sru* of [13, 110, 111] and the set divide-and-conquer operator *pump* appearing in the functional database language FAD [4] as well as in the work of Beeri and Kornatzky on algebraic optimization of object query languages [6].

Initial types in union representation induce monads in much the same manner as their insert representation variants. Specifically, for a given type $(\mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \triangledown \mathit{union}^{\mathsf{T}}, \mathsf{T})$, obtain the corresponding T-monad with zero $(\mathsf{T}, \mathit{unit}, \mathit{join}, \mathit{zero})$ by defining

$$\begin{aligned} \mathit{unit} &= \mathit{sng}^{\mathsf{T}} \\ \mathit{join} &= (\!|\, \mathit{empty}^{\mathsf{T}} \triangledown \mathit{id} \triangledown \mathit{union}^{\mathsf{T}} \,|\!) \\ \mathit{zero} &= \mathit{empty}^{\mathsf{T}} \ , \end{aligned}$$

which exactly generates the monads we have already met in Paragraph 61. This implies that an identical notion of monad comprehensions may alternatively be built on top of types in union representation.

What, then, has been our principle rationale for the choice of the insert representation in this text?

The more obvious reason, which we have already mentioned, is related to a core virtue: laziness. The insert representation makes economical use of just two (instead of three) constructors which, in a world of inductive types and structural recursion, has the pertinent effect of rendering definitions more compact and to simplify calculations. A similar observation can be made for datatype equations. Encoding the monoidal structure of $(\mathit{empty}^{\mathsf{T}}, \mathit{union}^{\mathsf{T}})$ requires the specification of two equations even for the $\mathsf{T} = \mathsf{list}$ algebra in union representation. Remember that the list algebra is freely generated in insert representation.

Another, and deeper, reason relates the efficiency and expressiveness of programs written in insert and union representation style, respectively: for some fixed type constructor T, suppose that $(\sigma = \mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \triangledown \mathit{union}^{\mathsf{T}}, \mathsf{T})$ denotes the initial type induced by the functor G. Given this, we can derive an equivalent F-algebra $\tau = \mathit{nil}^{\mathsf{T}} \triangledown \mathit{cons}^{\mathsf{T}}$ in insert representation that operates over the same carrier:

$$\begin{aligned} \mathit{nil}^{\mathsf{T}} &= \mathit{empty}^{\mathsf{T}} \\ \mathit{cons}^{\mathsf{T}} &= \mathit{union}^{\mathsf{T}} \cdot (\mathit{sng}^{\mathsf{T}} \times \mathit{id}) \ . \end{aligned}$$

This construction preserves the initiality property: τ is initial in $\mathbf{Alg}(\mathsf{F})$ if and only if σ is initial in $\mathbf{Alg}(\mathsf{G})$ [14]. A query in union representation style,

i. e., effectively a G-catamorphism $(\!|\, z \triangledown s \triangledown \oplus \,|\!)_{\mathsf{G}}$, thus has a simple first-order and efficient translation into insert representation style:

$$(\!|\, z \triangledown (\oplus \cdot (s \times id)) \,|\!)_{\mathsf{F}} \ .$$

Can we do the converse? We can certainly construct a G-algebra σ from a given initial F-algebra $\tau = nil^{\mathsf{T}} \triangledown cons^{\mathsf{T}}$:

$$\begin{array}{rcl} empty^{\mathsf{T}} & = & nil^{\mathsf{T}} \\ sng^{\mathsf{T}} & = & cons^{\mathsf{T}} \cdot (id \vartriangle nil^{\mathsf{T}}) \\ union^{\mathsf{T}}\,(xs, ys) & = & (\!|\, ys \triangledown cons^{\mathsf{T}} \,|\!)\, xs \ , \end{array}$$

and the resulting algebra will be initial in $\mathbf{Alg}(\mathsf{F})$ as well. The bad news is that the above construction is *not* valid for an arbitrary (especially non-initial) source algebra. This implies that we cannot use this scheme to derive a query in union representation style from a given insertion style query as the query's target algebra will not be initial in general [14]. In fact there is no translation from the insertion to the union style that is simple (first-order) and efficient at the same time:

- In [14], Breazu-Tannen and Subrahmanyam give a simple translation that makes use of the monoidal structure inherent to $(id, \cdot)$ and thus is higher-order. This would require us to extend our system of type constructors by the function type former $(\rightarrow)$ which we are trying to avoid.
- Suciu and Wong present a first-order translation that is inefficient in that it maps certain polynomial time computable insertion style queries into union style equivalents that require exponential space [111].

The bottom line of this argument is that a query representation based on the insert representation is at least as expressive as its union representation variant. At the same time, an insertion style query is efficiently computable whenever the equivalent union style query is. The converse is not true.

73 Collection Comprehensions, Kleisli Monads, and Ringads. In a series of articles, Trinder et al. developed a theory of collection types based on an extended monad notion, the *ringad* [113, 123]. This work used the union representation of a collection type constructor T as its starting point, *i. e.*, the G-algebra (which is non-free: $empty^{\mathsf{T}}$ is required to be a unit of $union^{\mathsf{T}}$):

$$empty^{\mathsf{T}} \triangledown sng^{\mathsf{T}} \triangledown union^{\mathsf{T}} : \mathsf{G}(E, \mathsf{T}E) \rightarrow \mathsf{T}E \ ,$$

with G denoting the union representation functor of the previous paragraph. The theory then added an iteration abstraction in the form of the binary operator $\gg$ (pronounced *bind*) to this algebra to complete its notion of a *collection type*. Operator $\gg$ is expected to interact with the above algebra as follows (with $f : E \to \mathsf{T}E'$):

$$\begin{array}{lcl} (\gg f) \cdot \mathit{empty}^{\mathsf{T}} & = & \mathit{empty}^{\mathsf{T}} \\ (\gg f) \cdot \mathit{sng}^{\mathsf{T}} & = & f \\ (\gg f) \cdot \mathit{union}^{\mathsf{T}} & = & \mathit{union}^{\mathsf{T}} \cdot ((\gg f) \times (\gg f)) \ , \end{array}$$

in which $(\gg f)$ denotes a *section* (partial application) of $\gg$ defined to mean $\lambda x.x \gg f$. Note that $\gg$ provides a restricted form of structural recursion that, in contrast to a general G-catamorphism $(\!|\cdot|\!)_{\mathsf{G}}$, is always well-defined:

$$(\gg f) \quad = \quad (\!| \mathit{empty}^{\mathsf{T}} \triangledown f \triangledown \mathit{union}^{\mathsf{T}} |\!)_{\mathsf{G}} \ .$$

Operator $\gg$ has been shown sufficiently expressive to cover a broad range of query primitives.

Trinder observed that the triple $(\mathsf{T}, \mathit{sng}^{\mathsf{T}}, \gg)$ forms a *Kleisli triple* [70], an alternative encoding of the monad concept that is, however, equivalent to the one we introduced in Definition 54 which

$$\begin{array}{lcl} \mathit{sng}^{\mathsf{T}} & = & \mathit{unit}^{\mathsf{T}} \\ (\gg f) & = & \mathit{join}^{\mathsf{T}} \cdot \mathsf{T}f \end{array}$$

suggest (for a formal proof of the equivalence see [80]). Based on this *ringad* structure, *i. e.*, the triple $(\mathsf{T}, \mathit{sng}^{\mathsf{T}}, \gg)$ in companion with $\mathit{empty}^{\mathsf{T}}$ and $\mathit{union}^{\mathsf{T}}$ (these two constructors indeed add to the expressivity as they are not expressible in the monad, see [20]), Trinder postulated monad comprehensions as an effective notation for collection query languages.

Chan and Trinder developed this basic theory of collection types into a so-called *object comprehension* language that incorporated the features needed to communicate with the object type hierarchy of an object-oriented DBMS [24, 25]. However, they never intended to interpret comprehensions in non-collection monads. Query language concepts like quantification and aggregation have instead been added as additional features outside the collection theory domain.

Along similar lines, Buneman, Naqvi, Tannen, and Wong laid the foundation of a theory of collection types by successively enriching Kleisli monads $(\mathsf{T}, \mathit{sng}^{\mathsf{T}}, \gg)$ to ringads, which were subsequently enhanced with a notion of equality and conditional expressions [13, 20]. (Note that for the monad extension operator *ext* occuring in this work we have $\mathit{ext}\ f\ \mathit{xs} = \mathit{xs} \gg f$.) The

resulting language, if restricted to operate in the set monad, was shown to be exactly as expressive as the nested relational algebra of Schek and Scholl and related proposals for relational non-first normal form algebras [99].

Building on this foundation, Wong devised the *Kleisli* collection processing framework which incorporated collection monad comprehensions as a sublanguage [124]. This language relies on *monad morphisms*—mappings between two monads T' and T that preserve the monadic operators (see Definition 86 and [117])—to allow for generator ranges of type T' inside a comprehension to be interpreted in collection monad T. The relevant translation step is (the rest of the translation resembles desugaring scheme $\mathscr{M}$ of Definition 69)

$$[\![\, e \mid v \leftarrow e' : \mathsf{T}'E \,]\!]^{\mathsf{T}} \quad \rightsquigarrow \quad e' \gg\!=' (\lambda v.\mathit{sng}^{\mathsf{T}} \cdot e) \;,$$

based on a modified monadic *bind* $\gg\!='$ defined to mean

$$\begin{array}{lcl} (\gg\!=' f) \cdot \mathit{empty}^{\mathsf{T}'} & = & \mathit{empty}^{\mathsf{T}} \\ (\gg\!=' f) \cdot \mathit{sng}^{\mathsf{T}'} & = & f \\ (\gg\!=' f) \cdot \mathit{union}^{\mathsf{T}'} & = & \mathit{union}^{\mathsf{T}} \cdot ((\gg\!=' f) \times (\gg\!=' f)) \;, \end{array}$$

so that $(\gg\!=' \mathit{sng}^{\mathsf{T}})$ constitutes a monad morphism between the collection monads T' and T.

In the form proposed by Wong however, the monad morphism approach did not enable the interpretation of comprehensions in non-collection monads. The resulting language is thus a hybrid assembled from collection comprehensions and separate structural recursion operators to implement aggregation and quantification. The work of Buneman, Naqvi, Tannen, and Wong has, nonetheless, been a major source of inspiration for the work we are discussing in this text. It is the group of researchers around Buneman to which many of the foundational ideas of a categorical treatment of query languages have to be attributed to.

74 Monoid Comprehensions. Fegaras and Maier fabricated an alternative query calculus, the *monoid comprehension calculus*, based on *monoids* and *homomorphisms* between these [38, 40]. The core of the calculus is, once more, formed by G-algebras, *i. e.*, types in union representation. The principal query construct are monoid homomorphisms—or rather homomorphic extensions—between instances of these types. In the lingo of initial algebras, given the initial type $(\tau = \mathit{empty}^{\mathsf{T}} \triangledown \mathit{sng}^{\mathsf{T}} \triangledown \mathit{union}^{\mathsf{T}}, \mathsf{T})$ and the G-algebra $\alpha = z \triangledown s \triangledown \oplus$, the *homomorphic extension* of function f from τ to α is

$$\mathit{hom}^{\tau \to \alpha} f \quad = \quad (\!|\, z \triangledown f \triangledown \oplus \,|\!)_{\mathsf{G}} \;.$$

Like $\gg$-, combinator *hom* defines a well-behaved structural recursion.[7] Unlike $\gg$-, *hom* is additionally capable of coercing types (from τ to α in the case above) while $\gg$ is bound to operate in the T-monad induced by its initial type only:

$$(\gg f) \;=\; (\!|\, empty^{\mathsf{T}} \triangledown f \triangledown union^{\mathsf{T}} \,|\!)_{\mathsf{G}} \;=\; hom^{\tau\to\tau}\, f\ .$$

This property of *hom* enabled Fegaras and Maier to postulate monoid comprehensions over different union representation types, including non-collection instances like, *e. g.*, $true \triangledown id \triangledown \wedge$. Let τ' denote the monoid associated with the type constructor T', then the relevant monoid comprehension translation rule reads:

$$[\![\, e \mid v \leftarrow e' : \mathsf{T}'E \,]\!]^{\mathsf{T}} \quad \rightsquigarrow \quad hom^{\tau'\to\tau}\,(\lambda v.sng^{\mathsf{T}} \cdot e)\, e'\ .$$

If we rewrite the above only slightly, we can spot a (more or less rough) similarity to the catamorphism-based coercion of monadic types (*cf.* $\mathscr{M}$-2'):

$$\begin{aligned}
& hom^{\tau'\to\tau}\,(\lambda v.sng^{\mathsf{T}} \cdot e) \\
=\;& \quad \{\, \text{unfold homomorphic extension } hom \,\} \\
& (\!|\, empty^{\mathsf{T}} \triangledown (\lambda v.sng^{\mathsf{T}} \cdot e) \triangledown union^{\mathsf{T}} \,|\!) \\
=\;& \quad \{\ \text{sum, } v \text{ not free in } sng^{\mathsf{T}} \,\} \\
& (\!|\, (empty^{\mathsf{T}} \triangledown sng^{\mathsf{T}} \triangledown union^{\mathsf{T}}) \cdot (id + \lambda v.e + id \times id) \,|\!) \\
=\;& \quad \{\, \text{union representation functor } \mathsf{G} \,\} \\
& (\!|\, (empty^{\mathsf{T}} \triangledown sng^{\mathsf{T}} \triangledown union^{\mathsf{T}}) \cdot \mathsf{G}(\lambda v.e, id) \,|\!) \\
=\;& \quad \{\, \textsc{Cata-Map-Fusion} \,\} \\
& (\!|\, \tau \,|\!) \cdot \mathsf{T}(\lambda v.e)\ ,
\end{aligned}$$

which relates monoid comprehensions and our proposal for multi-monad comprehensions.

2.11 Monad Comprehensions in Functional Programming

75 Moggi has been the first to establish a connection between the categorical monad notion and the semantics of programming languages [86, 87]. Since these days, Wadler and others proposed monads as a paradigm after which functional programs and libraries should be structured. Monads have become

[7] Given that $\oplus$ is commutative and idempotent whenever $union^{\mathsf{T}}$ is, respectively.

ubiquitous in functional programming [68, 73, 93, 94, 117, 119, 120, 121]. In spite of the terseness of their interface, monads opened new and exciting ways to construct combinator libraries for a broad variety of tasks, including those that, up to then, were stumbling stones in a purely functional environment: side-effect free I/O, stateful computation, exception handling, and the often notationally cumbersome continuation passing style of programming.

Even better: the intimate connection between monads and comprehensions provided a convenient syntax to express the just mentioned tasks in functional programs *for free*. This had an impact on the design of the functional programming language Haskell 1.4 [91] in which the list monad lost its exceptional role: comprehensions could be defined over an arbitrary monad T (with the generator ranges bound to be of type T, too).[8]

Let us close this chapter with the review of two monad instances which could especially affect the optimization of database programming and update languages: the *exception* and *state transformer* monads, respectively.

Recall from Example 36 the definition of the type constructor maybe which lifts any type E onto a type with a distinguished constant *nothing*:

$$nothing \triangledown just : 1 + E \to \mathsf{maybe}E \ .$$

An expression evaluated in the monad induced by maybe may be interpreted as a computation that may fail and thus raise an exception (which is indicated by returning *nothing* as the result).

We get hold of the maybe monad through

$$\begin{aligned} unit^{\mathsf{maybe}} &= just \\ join^{\mathsf{maybe}} &= (\!|\, nothing \triangledown id \,|\!) \\ zero^{\mathsf{maybe}} &= nothing \ . \end{aligned}$$

(Remember that the morphism mapping maybe f yields $f\,x$ for an argument $just\,x$ and returns *nothing* otherwise.)

The meaning of a comprehension evaluated in the maybe monad is given by the desugaring scheme $\mathscr{M}$ of Definition 69: a generator $v \leftarrow e : \mathsf{maybe}E$ binds variable v to x if $e = just\,x$ and propagates the binding. Should e raise an exception, the comprehension evaluates to *nothing* as a whole (the exception is "thrown"); a filter acts like an assertion that, should it fail, also raises an exception. Given this, the comprehension

$$[\![\, x/y \,|\, x \leftarrow f, y \leftarrow g, y \neq 0 \,]\!]^{\mathsf{maybe}}$$

[8] At the time of writing, this design decision has been reversed on a syntactical level: the now current Haskell 98 definition adopts the equivalent do-notation to express monadic computations [65].

computes $just\,(f/g)$ provided that both arguments are well-defined and the denominator is not zero. Otherwise the comprehension indicates failure by evaluating to *nothing*.

Stateful computation and purely functional programming can live together if a representation of the program state (of type S, say) is explicitly passed as an extra argument between function calls. State update is implemented by construction of a fresh copy of the state representation. Passing the state explicitly quickly becomes cumbersome and error-prone, especially if the type of state representation S is subject to change during program development. However, an expression that is evaluated in the state monad to be defined in a minute, may act *as if* updatable state were available to the program: the state is passed implicitly and its representation becomes opaque (which is what we desire).

The type constructor we need here is $\mathsf{state}E = S \to E \times S$, which constructs the type of *state transformers*. A value of type $\mathsf{state}E$ denotes a computation that yields a result of type E (which possibly depends on the state it is evaluated in) and additionally "side-effects" the state by constructing a (modified) copy of it. Both E and the new state are returned. Note that state makes use of the function type former $(\to)$ which we are not covering as a datatype constructor in this text; since we are working in the cartesian-closed category **Set** this is no principle obstacle however (*e. g.*, see [10]).

The ingredients of the state monad thus are as follows (state gets by without *zero*):

$$\begin{aligned} \mathsf{state}E &= S \to E \times S \\ \mathsf{state}\,f &= \lambda s.(f \times id) \cdot s \end{aligned}$$

$$\begin{aligned} unit^{\mathsf{state}}\,x &= (K_x \vartriangle id) \\ join^{\mathsf{state}}\,ss &= apply \cdot ss \qquad \text{with } apply\,(f,x) = f\,x\ . \end{aligned}$$

The functoriality of state directly follows from the properties of arrow composition. Observe that $unit^{\mathsf{state}}\,x$ denotes a computation that simply returns x without affecting the state. $join^{\mathsf{state}}\,ss$ applies the state transformer ss yielding a state transformer which is then applied in a subsequent step (*i. e.*, $join^{\mathsf{state}}$ *threads* the state through a sequence of transformations).

Based on the above, we can readily implement *mutable references* in a pure functional language. Ohori used a closely related technique to enrich a referentially transparent database programming language with a notion of *object identity* and *mutable objects* without sacrificing the optimizability

of the language [90]. Assuming that a reference is identified by a unique sequence number, it is sufficient to keep the next sequence number to assign and a list that associates those numbers with the referenced content (of type *String* in this example) in the state, thus $S = Num \times \mathsf{list}(Num \times String)$. To unclutter the syntax let us additionally define three primitives that operate on a store of type S: *new* allocates a new sequence number and initializes the content of the newly created reference; *asgn* overwrites the content of a given reference; *deref* accesses the contents of a given reference but does not alter the store:

$$\begin{array}{ll} new & : String \to \mathsf{state} Num \\ new\, c & = \lambda(n,l).(n,(n+1, assoc\, l\, (n,c))) \\ \\ asgn & : (Num \times String) \to \mathsf{state} 1 \\ asgn\, (n',c) & = \lambda(n,l).((),(n, assoc\, l\, (n',c))) \\ \\ deref & : Num \to \mathsf{state} String \\ deref\, n' & = \lambda(n,l).(lookup\, l\, n', (n,l)) \end{array}$$

(the functions *assoc* and *lookup* do the obvious on assocation lists of type $\mathsf{list}(Num \times String)$).

Comprehensions evaluated in the state monad exhibit a remarkably "imperative" flavor although we are, of course, still operating in a purely functional environment. The comprehension (applied to the initial state $(0, nil)$ in which no references have been allocated yet)

$$[\![\, z [\![x \leftarrow new\, \texttt{"foo"}, y \leftarrow new\, \texttt{"bar"}, z \leftarrow deref\, y, p \leftarrow asgn\, (x,z)]\!]^{\mathsf{state}}\, (0, nil)$$

evaluates to $\big(\texttt{"bar"}, (2, [(0, \texttt{"bar"}), (1, \texttt{"bar"})])\big)$ and thus describes the same computation as the imperative program

$$x := \texttt{"foo"}; \quad y := \texttt{"bar"}; \quad z := y; \quad x := z\ .$$

It is our conjecture (which we regrettably cannot investigate more deeply this text) that not only queries but database languages in general may benefit from a monadic interpretation. An appropriately adapted multi-monad comprehension desugaring scheme $\mathscr{M}$ could provide the basis for the optimization theory for database programming languages that speak about collections and stateful computation (*i. e.*, updates) at the same time.

But let us not get carried away. We started out to Comprehend Queries.

Chapter 3

Query Compilation and Normalization

76 A comprehension of queries is not complete if it lacks the apparatus to assign meaning to *user-level* query syntax. The first part of this chapter will connect the internal categorical machinery to the "outside" world of declarative query languages. Thanks to the work we have spent with the introduction of monad comprehensions, we will be able to establish this connection by means of a *syntactic mapping*, $\mathscr{Q}$, to be defined shortly. Once we have established $\mathscr{Q}$, the composition $\mathscr{M} \cdot \mathscr{Q}$ will assign a categorical semantics to user queries and thus provides us with a formal, albeit preliminary, query comprehension in terms of catamorphic mappings.

Interpreting a query by way of $\mathscr{M} \cdot \mathscr{Q}$ alone is far from feasible if we hope for an efficient query execution. $\mathscr{M} \cdot \mathscr{Q}$ emits nothing but nested spine transformers from which an underlying query engine can hardly derive anything but naive nested-loop iteration strategies. In certain cases, however, nested-loop iteration may actually be an adequate execution strategy and in these cases an execution plan may be directly "*read off*" the catamorphisms emitted by $\mathscr{M} \cdot \mathscr{Q}$. This is the reason why, from time to time, we invest some initial thought in optimizations which are immediately applicable to catamorphic queries. We will have to say more about this.

Nevertheless, it is our hypothesis that monad comprehensions assign meaning to queries in a form from which more efficient execution strategies are derivable. The *normalization* of the monad comprehension expressions constructed by $\mathscr{Q}$ will provide an important rewriting step towards a query form that facilitates exploitation of the various advanced *join* or *grouping* operators which database query engines have to offer. A number of well-established query unnesting techniques turn out to be normalization steps if we express them in the lingo of monad comprehensions. Normalization will

take up the second part of this chapter.

77 Throughout this text, ODMG's second version of the object query language OQL will be the source language of choice. In [23, Chapter 4], it is stated that OQL shall be understood as a *functional language* and we will treat it as such: OQL queries are built from basic query formers which, given that the language's typing rules are obeyed, may be freely composed to form more complex query expressions. None of the query formers initiate side-effects.

It is our ambition to understand almost anything of OQL in terms of the categorical framework we have set up so far. We are, however, falling short of emphasizing the purely object-oriented features of the language: object types will be viewed as abstract data types that allow for equality tests (presumably based on object identity but we are not looking into this). Invocation of side-effect free methods will be modeled as the application of a function which—besides type information—remains opaque to us. Lacking the *object* notion, we will also not reason about inheritance and late binding on method calls. These omissions are no principle obstacles. Fegaras and Maier have sketched a translation from OQL to a monoid comprehension calculus (see Paragraph 74) that has been extended later by Fegaras with a notion of object identity and destructive updates expressed within the monoid calculus itself (*cf.* Section 2.11 and the discussion of the `state` monad) [39, 40].

Given these restrictions, OQL queries may be typed using the orthogonal system of categorical datatype constructors we have discussed in Chapter 2. Put differently, we are querying values of arbitrary but finite nesting depth which—as it was argued by Abiteboul and Kanellakis in [1]—form the core of object-oriented database systems. The adoption of OQL as a source language is, of course, not principle to the method. We are positive that other query languages for complex value databases, *e. g.*, the query fragments of SQL-92 and SQL-3, may be comprehended using the techniques we are developing: OQL constitutes a test case that embraces the SQL dialects in terms of expressiveness of its type system as well as orthogonal applicability of its query formers.

Familiarity with OQL syntax as it was proposed in [23] is assumed from now on. The forthcoming will supply the semantics and additionally—as remarked in the last paragraph—initial catamorphic means for the actual execution of OQL queries. In anticipation of the things to come and to provide reference, Riedel and Scholl as well as Cherniack discuss alternative semantic assignments that are reasonably close to ours [30, 97].

3.1 A Catamorphic Interpretation of OQL

This is the right time to explain a few notational conventions and preliminaries, some of which are consequences of the categorical model of data types we are dealing with. OQL keywords will be typeset in `typewriter` style.

78 Types. (a) The default category is **Set**. As before we will use the names $E, E', \ldots$ as well as subscripted variants thereof to indicate variables that range over types, *i. e.*, objects in **Set**. If not stated otherwise, T and variants denote one of the type functors list, bag, and set. The type functors implement the OQL collection type constructors *list*, *bag*, and *set*, respectively.

(b) We translate OQL's basic types as follows: numerical types are represented by type *Num*; character and string literals are of type *String*; *Bool* implements the type of the two boolean values. Object types map into the abstract type *Obj*.

(c) OQL comes equipped with a *record type constructor with tagged fields* and its corresponding value constructor $\texttt{struct}(l_1 : e_1, \ldots, l_n : e_n)$ in which the l_i denote unique field tags and the e_i are field entries (arbitrary OQL expressions). If we have $e_i : E_i$, the thus constructed value has the OQL type $\mathit{struct}(l_1 : E_1, \ldots, l_n : E_n)$.

Such record values are represented as right-deep nested products of the form $(e_1, (e_2, (\cdots (e_n, ()) \cdots)))$ whose types are $E_1 \times (E_2 \times (\cdots (E_n \times \mathit{1}) \cdots))$. During this translation we maintain a mapping $\mathit{tag} = \{l_i \mapsto i\}, i = 1 \ldots n$, that keeps track of tag positions. We may, however, safely drop the rigid parenthesized notation and write $(e_1, \ldots, e_n) : E_1 \times \cdots \times E_n$ instead, *i. e.*, we may act as if we were operating with n-ary product types and values, the latter also called *tuples* from now on. These simplifications are justified by a simple observation about isomorphisms: in a category with products, objects $E_1 \times (E_2 \times E_3)$ and $(E_1 \times E_2) \times E_3$ are isomorphic by means of the arrows

$$\begin{array}{lcl} (\mathit{id} \times \mathit{outl}) \vartriangle (\mathit{outr} \cdot \mathit{outr}) & : & E_1 \times (E_2 \times E_3) \to (E_1 \times E_2) \times E_3 \\ (\mathit{outl} \cdot \mathit{outl}) \vartriangle (\mathit{outr} \times \mathit{id}) & : & (E_1 \times E_2) \times E_3 \to E_1 \times (E_2 \times E_3)\ , \end{array}$$

which renders $\times$ associative so that the absence or presence of parentheses does not alter the meaning of a nested product. Likewise, $E \times \mathit{1}$ and E are isomorphic because we have that $\mathit{outl} \cdot (\mathit{id} \vartriangle ()) = \mathit{id}_E$ and $(\mathit{id} \vartriangle ()) \cdot \mathit{outl} = \mathit{id}_{E \times 1}$. This justifies the omission of the $()$ in tuple notation.

(d) All types E built from the basic types *Num*, *String*, *Bool*, *Obj*, and categorical datatype constructors are *equality types* in the ML sense [85], *i. e.*, they admit equality tests by means of the overloaded predicate $= : E \times E \to \mathit{Bool}$. Equality of products is defined component-wise. Equality on $\mathsf{T}E$ is decided by the initial algebra that induces T (see Section 2.7).

Types *Num* and *String* are taken to be *ordered*. The type $E_1 \times \cdots \times E_n$ is ordered if the E_i are ordered (in which case the ordering of the tuple type is defined lexicographically). If E is ordered, we are admitted to apply the overloaded predicate $<\!: E \times E \to \mathit{Bool}$ to check if its argument pair is ordered.

(e) To summarize, the following grammar yields all valid query types ty:

ty	$\to$	$\mathit{Num} \mid \mathit{String} \mid \mathit{Bool} \mid \mathit{Obj}$	basic types
	$\mid$	$\mathsf{set}\ ty \mid \mathsf{bag}\ ty \mid \mathsf{list}\ ty$	type functors
	$\mid$	$ty \times ty$	products
	$\mid$	$(\ ty\ )$	

79 Typing OQL fragments. (f) Designed as a syntax-directed mapping, $\mathscr{Q}$ descends the parse tree of its argument query, starting with the root which represents the entire query text. $\mathscr{Q}$ operates in a *bottom-up* manner: subquery fragments are translated before these translations are assembled to emit a translation of the enclosing query. During its walk of the parse tree, $\mathscr{Q}$ consequently encounters *open* OQL fragments, *i. e.*, OQL subqueries containing *free query variables* whose binding sites are located further up the parse tree (the root fragment is expected to be closed).

In the OQL query below, variable y appears free in three subqueries: y (in the `select` clause), `forall` x `in` xs`:` x `=` y, and x `=` y. Variable x appears free in x `=` y. The binding sites of y and x are the `from` clause and the `forall` quantifier, respectively:

```
select y
  from ys as y
 where forall x in xs: x = y .
```

To cope with the typing of OQL fragments in order to clearly define query variable scoping, we will interpret open OQL fragments as *functions of their free variables*. To illustrate, the above query will be understood as

```
select (λv.v) y
  from ys as y
 where (λv.forall x in xs: (λv′.v′ = v) x) y ,
```

so that, for example, the `where` clause of an OQL `select-from-where` block is a function of type $E \to \mathit{Bool}$ (where type E is determined by the query's `from` clause).

(g) Query types will be given and derived using standard notation [22]: the *type judgment* $\Gamma \vdash e\!: E$ asserts the type of term e to be E, given that

the free variables in e are typed according to the *type environment* Γ ("*e has type E in* Γ"). A type environment is of the form $\varnothing, v_1 : E_1, \ldots, v_n : E_n$ (with $n \geq 0$, $\varnothing$ denotes the empty environment).

A *typing rule* derives the type judgment for a term (below the horizontal rule) under the premise of a number (maybe zero) of conditions—typically other type judgments—listed above the rule. The typing rule for OQL's `forall` quantifier reads (note that v may appear free in p and the predicate is thus typed $\lambda v.p : E \to \mathit{Bool}$):

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma, v : E \vdash p : \mathit{Bool}}{\Gamma \vdash \texttt{forall}\ v\ \texttt{in}\ e\texttt{:}\ p : \mathit{Bool}}\ (\textsc{Ty-Forall})$$

where $\Gamma, v : E$ denotes Γ after the addition of variable-type assignment $v : E$. To render environment updates more compact, let $\Gamma, \overline{v}_n : \overline{E}_n$ denote the environment $\Gamma, v_1 : E_1, v_2 : E_2, \ldots v_n : E_n$ in which n will be determined by the context. For $n = 0$, this denotes Γ itself.

8o Miscellaneous. (h) The term $e[x/y]$ denotes term e with all free occurrences of variable y in e replaced by term x. Let $e[x/\overline{y}_n]$ be a shorthand for $e[x/y_1][x/y_2]\cdots[x/y_n]$ if the y_i are in context. Likewise, we write $e[\overline{x}_n/\overline{y}_n]$ to abbreviate the replacement $e[x_1/y_1][x_2/y_2]\cdots[x_n/y_n]$.

(i) During the translation, we will identify basic OQL constants with their obvious meaning in the categorical world, *e. g.* we do not distinguish `+` and $+ = {}^{\mathsf{sum}}_{\vdots}$ or `true` and $\mathit{true} = [\,]^{\mathsf{all}}$, respectively.

81 OQL translation. We will define $\mathscr{Q}$ by case selection on the various query formers provided by OQL. Much of the simplicity of $\mathscr{Q}$ is due to its *uniform* translation strategy [111]: a query e may be compiled independently from subqueries occurring in it. During the translation of e, the subquery fragments are treated as free variables that may be instantiated later to complete the translation. In the variant described in this text, $\mathscr{Q}$ calls itself recursively to perform these instantiations.

Nevertheless, the strategy may also be used in system contexts that call for *separate compilation* of query fragments. Such situations typically arise if the query language is embedded into a host programming language: parts of the query text are dependent on host language variables or are even computed by host language routines and are not known beforehand. Despite the presence of such dynamically generated query parts, we could translate the statically known query fragments (under the premise that the dynamic parts themselves will be correctly typed) at compile-time of the host program.

This completes the preliminaries, so let us now turn to the actual definition of the OQL translation function $\mathscr{Q}$.

82 Definition. At several places we provide notes concerning simplifications of the translation. Skipping these should not affect the understanding of $\mathscr{Q}$.

Query variables. We use $\Gamma, v:E, \Gamma'$ to denote an environment that contains the variable-type-assignment $v:E$.

$$\frac{}{\Gamma, v:E, \Gamma' \vdash v:E}\ (\textsc{Ty-Var})$$

$$\mathscr{Q}\, v = v \qquad \mathscr{Q}\text{-}\textsc{Var}$$

Constants of atomic type. Let *Basic* denote one of *Num*, *String*, *Bool*, or *Obj*.

$$\frac{c \in \mathit{Basic}}{\Gamma \vdash c:\mathit{Basic}}\ (\textsc{Ty-Basic})$$

$$\mathscr{Q}\, c = c \qquad \mathscr{Q}\text{-}\textsc{Basic}$$

Record construction (`struct`). As already noted in Paragraph 78, we maintain a $tag = \{l_i \mapsto i\}$ mapping as a side-effect of the `struct` constructor translation. For mere simplicity we assume that record tags are database-wide uniquely determined.

$$\frac{\Gamma \vdash e_1:E_1 \quad \cdots \quad \Gamma \vdash e_n:E_n}{\Gamma \vdash \texttt{struct}(l_1:e_1, \ldots, l_n:e_n) : E_1 \times \cdots \times E_n}\ (\textsc{Ty-Struct})$$

$$\mathscr{Q}\,(\texttt{struct}(l_1:e_1, \ldots, l_n:e_n)) = (\mathscr{Q}\, e_1, \ldots, \mathscr{Q}\, e_n) \qquad \mathscr{Q}\text{-}\textsc{Struct}$$

Record access. For a function f, define its iterated application by $f^0 = id$ and $f^n = f \cdot f^{(n-1)}$.

$$\frac{\Gamma \vdash e : E_1 \times \cdots \times E_n \qquad tag\, l = i \quad (1 \le i \le n)}{\Gamma \vdash e\,.\,l : E_i}\ (\textsc{Ty-Dot})$$

$$\mathscr{Q}\,(e\,.\,l) = (outl \cdot outr^{(i-1)})\,(\mathscr{Q}\,e) \qquad \mathscr{Q}\text{-}\textsc{Dot}$$

Collection-typed constants (`set`, `bag`, `list`). Let *coll* be one of `set`, `bag`, or `list` and define $\mathsf{T}^{\mathtt{set}} = \mathsf{set}$, $\mathsf{T}^{\mathtt{bag}} = \mathsf{bag}$, and $\mathsf{T}^{\mathtt{list}} = \mathsf{list}$.

$$\frac{\Gamma \vdash e_1 : E \quad \cdots \quad \Gamma \vdash e_n : E}{\Gamma \vdash coll(e_1, \ldots, e_n) : \mathsf{T}^{coll} E}\ (\textsc{Ty-Coll})$$

$$\begin{array}{rl} & \mathscr{Q}\,(coll(e_1, \ldots, e_n)) \\ = & cons^{\mathsf{T}^{coll}}(\mathscr{Q}\,e_1, cons^{\mathsf{T}^{coll}}(\cdots cons^{\mathsf{T}^{coll}}(\mathscr{Q}\,e_n, nil^{\mathsf{T}^{coll}})\cdots)) \end{array} \qquad \mathscr{Q}\text{-}\textsc{Coll}$$

Arithmetic operators. Let symbol $\ominus$ denote `-` (unary minus) or `abs`, and let $\oplus$ represent one of `+`, `-` (subtraction), `*`, `/`, or `mod`.

$$\frac{\Gamma \vdash e : Num}{\Gamma \vdash \ominus e : Num}\ (\textsc{Ty-Arith-1})$$

$$\frac{\Gamma \vdash e : Num \qquad \Gamma \vdash e' : Num}{\Gamma \vdash e \oplus e' : Num}\ (\textsc{Ty-Arith-2})$$

$$\mathscr{Q}\,(\ominus e) = \ominus\,(\mathscr{Q}\,e) \qquad \mathscr{Q}\text{-}\textsc{Arith-1}$$

$$\mathscr{Q}\,(e \oplus e') = (\mathscr{Q}\,e) \oplus (\mathscr{Q}\,e') \qquad \mathscr{Q}\text{-}\textsc{Arith-2}$$

Equality and comparison operators. Let $\otimes$ stand for one of the infix operators `<`, `<=`, `=`, `!=`, `>=`, or `>`. We then have that $\otimes : E \times E \to Bool$ (E must be ordered if $\otimes$ represents one of the relational comparison operators).

$$\frac{\Gamma \vdash e : E \qquad \Gamma \vdash e' : E}{\Gamma \vdash e \otimes e' : Bool}\ (\textsc{Ty-Comp})$$

$$\mathscr{Q}\,(e \otimes e') = (\mathscr{Q}\,e) \otimes (\mathscr{Q}\,e') \qquad \mathscr{Q}\text{-}\textsc{Comp}$$

Boolean connectives. We use symbol $\oslash$ to denote either `and` or `or`.

$$\frac{\Gamma \vdash e : \mathit{Bool}}{\Gamma \vdash \texttt{not}\, e : \mathit{Bool}}\ (\textsc{Ty-Bool-1})$$

$$\frac{\Gamma \vdash e : \mathit{Bool} \qquad \Gamma \vdash e' : \mathit{Bool}}{\Gamma \vdash e \oslash e' : \mathit{Bool}}\ (\textsc{Ty-Bool-2})$$

$$\mathcal{Q}\,(\texttt{not}\, e) = \neg\,(\mathcal{Q}\, e) \qquad \mathcal{Q}\text{-Bool-1}$$
$$\mathcal{Q}\,(e \oslash e') = (\mathcal{Q}\, e) \oslash (\mathcal{Q}\, e') \qquad \mathcal{Q}\text{-Bool-2}$$

Selecting into a set (`select-distinct-from-where`). The types of $e_2, \ldots, e_n$ express the possibility to specify *dependent joins* in the `from` clause, *i. e.*, $x_1, \ldots, x_{i-1}$ may occur free in e_i ($\mathcal{Q}$ accounts for this by placing the e_i as generator ranges in order of their appearance in the `from` clause). We adopt a widespread convention and assume $\mathcal{Q}\, p = \mathit{true}$ should the query expression lack the `where` clause.

$$\frac{\begin{array}{l}\Gamma, \overline{x}_n : \overline{E}_n \vdash e : E \\ \Gamma, \overline{x}_{i-1} : \overline{E}_{i-1} \vdash e_i : \mathsf{T}_i E_i \qquad (1 \leq i \leq n) \\ \Gamma, \overline{x}_n : \overline{E}_n \vdash p : \mathit{Bool}\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{select distinct}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \texttt{where}\ p\end{array}\right) : \mathsf{set} E}\ (\textsc{Ty-Set-Sfw})$$

$$\begin{array}{rl} & \mathcal{Q}\left(\begin{array}{l}\texttt{select distinct}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \texttt{where}\ p\end{array}\right) \\ = & [\![\, \mathcal{Q}\, e \,]\!]\, x_1 \leftarrow \mathcal{Q}\, e_1, \ldots, x_n \leftarrow \mathcal{Q}\, e_n, \mathcal{Q}\, p \,]\!]^{\mathsf{set}} \qquad \mathcal{Q}\text{-Set-Sfw}\end{array}$$

Selecting into a bag (`select-from-where`). Note that, to render the translation well-defined, we require $\mathsf{T}_i \neq \mathsf{set}$ as the `bag` algebra obeys equation Eq-Comm only (*cf.* Paragraph 70).

$$\frac{\begin{array}{l}\Gamma, \overline{x}_n : \overline{E}_n \vdash e : E \\ \Gamma, \overline{x}_{i-1} : \overline{E}_{i-1} \vdash e_i : \mathsf{T}_i E_i \qquad \mathsf{T}_i \neq \mathsf{set} \qquad (1 \leq i \leq n) \\ \Gamma, \overline{x}_n : \overline{E}_n \vdash p : \mathit{Bool}\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{select}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \texttt{where}\ p\end{array}\right) : \mathsf{bag} E}\ (\textsc{Ty-Sfw})$$

$$
\begin{aligned}
& \mathscr{Q}\left(\begin{array}{l}\texttt{select}\ e\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1,\ldots,e_n\ \texttt{as}\ x_n\\ \texttt{where}\ p\end{array}\right)\\
= \quad & [\![\, \mathscr{Q}\, e \,]\!]\, x_1 \leftarrow \mathscr{Q}\, e_1, \ldots, x_n \leftarrow \mathscr{Q}\, e_n, \mathscr{Q}\, p \,]\!]^{\mathsf{bag}} \qquad\qquad \mathscr{Q}\text{-SFW}
\end{aligned}
$$

Grouping (`select-from-where-group by`). OQL's grouping construct partitions the n-fold product of the $e_1, \ldots, e_n$ with respect to the grouping criteria $g_1, \ldots, g_m$ (accessible in the `select` clause e under names $l_1, \ldots, l_m$): n-tuples that agree on all criteria are collected into a bag (accessible in e by referring to name *partition*). References to these names are replaced with corresponding tuple accesses by function *tuplify*. During the recursive descent of $\mathscr{Q}$, the thus introduced tuple accesses are subsequently translated into sequences of projections by $\mathscr{Q}$-DOT.

$$
\frac{\begin{array}{l}\Gamma, \overline{l}_g : \overline{E'}_g, \mathit{partition} : \mathsf{bag}(E_1 \times \cdots \times E_n) \vdash e : E\\ \Gamma, \overline{x}_{i-1} : \overline{E}_{i-1} \vdash e_i : \mathsf{T}_i E_i \qquad \mathsf{T}_i \neq \mathsf{set} \qquad (1 \leq i \leq n)\\ \Gamma, \overline{x}_n : \overline{E}_n \vdash p : \mathit{Bool} \qquad \Gamma, \overline{x}_n : \overline{E}_n \vdash g_j : E'_j \qquad (1 \leq j \leq m)\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{select}\ e\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1,\ldots,e_n\ \texttt{as}\ x_n\\ \texttt{where}\ p\\ \texttt{group by}\ l_1 : g_1, \ldots, l_m : g_m\end{array}\right) : \mathsf{bag} E} \ (\text{TY-GROUP})
$$

$$
\begin{aligned}
& \mathscr{Q}\left(\begin{array}{l}\texttt{select}\ e\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1,\ldots,e_n\ \texttt{as}\ x_n\\ \texttt{where}\ p\\ \texttt{group by}\ l_1 : g_1, \ldots, l_m : g_m\end{array}\right)\\
= \quad & [\![\, \mathscr{Q}\,(\mathit{tuplify}\, e) \,]\!]\\
& \quad y \leftarrow [\![(\mathscr{Q}\, g_1, \ldots, \mathscr{Q}\, g_m, [\![\, (x'_1, \ldots, x'_n) \,]\!]\, x'_1 \leftarrow \mathscr{Q}\, e_1, \ldots, x'_n \leftarrow \mathscr{Q}\, e_n,\\
& \qquad\qquad \mathscr{Q}\, g_1 = \mathscr{Q}\,(g_1[\overline{x'}_n/\overline{x}_n]), \ldots,\\
& \qquad\qquad \mathscr{Q}\, g_m = \mathscr{Q}\,(g_m[\overline{x'}_n/\overline{x}_n]) \,]\!]^{\mathsf{bag}}) \,]\!]\\
& \qquad x_1 \leftarrow \mathscr{Q}\, e_1, \ldots, x_n \leftarrow \mathscr{Q}\, e_n, \mathscr{Q}\, p \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{bag}}\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathscr{Q}\text{-GROUP}
\end{aligned}
$$

where

$$
\mathit{tuplify}\, t = t[y\,.\,1/l_1] \cdots [y\,.\,m/l_m][y\,.\,(m+1)/\mathit{partition}]\ .
$$

Constrained Grouping (`select-from-where-group by-having`). We have to call *tuplify* on both the `select` clause e and p'—a predicate to be evaluated against all partitions—to replace references by name into tuple projections.

$$\frac{\begin{array}{l}\Gamma, \overline{l}_m : \overline{E'}_m, \mathit{partition} : \mathsf{bag}(E_1 \times \cdots \times E_n) \vdash e : E \\ \Gamma, \overline{x}_{i-1} : \overline{E}_{i-1} \vdash e_i : \mathsf{T}_i E_i \qquad \mathsf{T}_i \neq \mathsf{set} \qquad (1 \leq i \leq n) \\ \Gamma, \overline{x}_n : \overline{E}_n \vdash p : \mathit{Bool} \qquad \Gamma, \overline{x}_n : \overline{E}_n \vdash g_j : E'_j \qquad (1 \leq j \leq m) \\ \Gamma, \overline{l}_m : \overline{E'}_m, \mathit{partition} : \mathsf{bag}(E_1 \times \cdots \times E_n) \vdash p' : \mathit{Bool}\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{select}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \quad \texttt{where}\ p \\ \texttt{group by}\ l_1 : g_1, \ldots, l_m : g_m \\ \quad \texttt{having}\ p'\end{array}\right) : \mathsf{bag} E} \text{ (Ty-Having)}$$

$$\begin{array}{ll} & \mathscr{Q}\left(\begin{array}{l}\texttt{select}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \quad \texttt{where}\ p \\ \texttt{group by}\ l_1 : g_1, \ldots, l_m : g_m \\ \quad \texttt{having}\ p'\end{array}\right) \\ = & [\![\, \mathscr{Q}\,(\mathit{tuplify}\, e)\,]\!] \\ & \quad y \leftarrow [\![(\mathscr{Q}\, g_1, \ldots, \mathscr{Q}\, g_g, [\![\, (x'_1, \ldots, x'_n)\,]\!] x'_1 \leftarrow \mathscr{Q}\, e_1, \ldots, x'_n \leftarrow \mathscr{Q}\, e_n, \\ & \qquad\qquad \mathscr{Q}\, g_1 = \mathscr{Q}\,(g_1[\overline{x'}_n/\overline{x}_n]), \ldots, \\ & \qquad\qquad \mathscr{Q}\, g_m = \mathscr{Q}\,(g_m[\overline{x'}_n/\overline{x}_n])\,]\!]^{\mathsf{bag}})\,]\!] \\ & \qquad x_1 \leftarrow \mathscr{Q}\, e_1, \ldots, x_n \leftarrow \mathscr{Q}\, e_n, \mathscr{Q}\, p\,]\!]^{\mathsf{bag}}, \\ & \quad \mathscr{Q}\,(\mathit{tuplify}\, p')\,]\!]^{\mathsf{bag}} \end{array}$$

$\mathscr{Q}$-Having

Sorting (`select-from-where-order by`). In the query below, let *ord* be a placeholder for `asc` or `desc`—a hint to sort the query result in ascending or descending order, respectively. The type E' of the sort criterion is required to be ordered.

$$\frac{\begin{array}{l}\Gamma, \overline{x}_n : \overline{E}_n \vdash e : E \\ \Gamma, \overline{x}_{i-1} : \overline{E}_{i-1} \vdash e_i : \mathsf{T}_i E_i \qquad \mathsf{T}_i \neq \mathsf{set} \qquad (1 \leq i \leq n) \\ \Gamma, \overline{x}_n : \overline{E}_n \vdash p : \mathit{Bool} \qquad \Gamma, \overline{x}_n : \overline{E}_n \vdash o : E'\end{array}}{\Gamma \vdash \left(\begin{array}{l}\texttt{select}\ e \\ \quad \texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n \\ \quad \texttt{where}\ p \\ \texttt{order by}\ o\ \mathit{ord}\end{array}\right) : \mathsf{list} E} \text{ (Ty-Sort)}$$

$$
\begin{aligned}
& \mathscr{Q}\left(\begin{array}{l}\texttt{select}\ e\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1, \ldots, e_n\ \texttt{as}\ x_n\\ \quad\texttt{where}\ p\\ \texttt{order by}\ o\ ord\end{array}\right)\\
= \quad & [\![\, \mathscr{Q}\,(e[y/\overline{x}_n])\,[\![\, y \leftarrow (\!|\, nil^{\mathsf{list}} \triangledown ins^{\mathsf{list}} \lessdot \,|\!)\ [\![\, \mathscr{Q}\, o\,[\![\, x_1 \leftarrow \mathscr{Q}\, e_1, \ldots,\\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad x_n \leftarrow \mathscr{Q}\, e_n, \mathscr{Q}\, p\,]\!]^{\mathsf{bag}}\,]\!]^{\mathsf{list}}
\end{aligned}
$$

$\mathscr{Q}$-Sort

where

$$
\lessdot = \begin{cases} < : E' \times E' \to \mathit{Bool} & \text{if } \mathit{ord} = \texttt{asc},\\ > : E' \times E' \to \mathit{Bool} & \text{else.}\end{cases}
$$

Notes. Coherent with our quest for a catamorphic model for OQL, $(\!|\, nil^{\mathsf{list}} \triangledown ins^{\mathsf{list}} \lessdot \,|\!)$ provides a purely (see below) catamorphic representation of *insertion sort*: F-algebra $nil^{\mathsf{list}} \triangledown (ins^{\mathsf{list}} \lessdot) : \mathsf{F}(E', \mathsf{list}E') \to \mathsf{list}E'$ is the algebra of sorted lists in insert representation. Parameterized with $\lessdot$, operator ins^{list} inserts an element into an already sorted list, respecting order $\lessdot$. Note that ins^{list} itself may be understood as a catamorphism:

$$
ins^{\mathsf{list}} \lessdot\ (x, xs) = (\!|\, [x]^{\mathsf{list}} \triangledown (swap \cdot (\lessdot \cdot (id \times hd))?) \,|\!)\ xs
$$

where

$$
\begin{aligned}
hd \cdot \dot{!}^{\top} &= outl\\
tl \cdot \dot{!}^{\top} &= outr\\
swap &= \dot{!}^{\top} \triangledown (\dot{!}^{\top} \cdot ((hd \cdot outr) \vartriangle (\dot{!}^{\top} \cdot (id \times tl))))
\end{aligned}
$$

(defined like this, $ins^{\mathsf{list}} \lessdot$ is left-commutative but not left-idempotent).

Catamorphic sorting does not stress the efficient implementation of sorting but rather its algebraic properties; it is these properties that we would like $\mathscr{Q}$ to make explicit (however, see [49] where merge sort is derived from a catamorphic insertion sort by program transformations alone). Applied to algebra $nil^{\mathsf{list}} \triangledown ins^{\mathsf{list}} \lessdot$, Lemma 60 yields a monad of sorted lists and thus its corresponding monad comprehension notion, but we are not following this route here.

Intersection (`intersect`).

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma \vdash e' : \mathsf{T}'E}{\Gamma \vdash e \texttt{ intersect } e' : \mathsf{T}E} \text{ (Ty-Intersect)}$$

$$\begin{aligned} & \mathscr{Q}\,(e \texttt{ intersect } e') \\ = \;& [\![\, x \,|\, x \leftarrow \mathscr{Q}\,e, [\![\, x = x' \,|\, x' \leftarrow \mathscr{Q}\,e' \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{T}} \qquad \mathscr{Q}\text{-Intersect} \end{aligned}$$

Difference (`except`). Note that this translation of `except` does not subtract multiplicities of elements (but rather removes all elements from the result that occur in both e and e') if $\mathsf{T} = \mathsf{T}' = \mathsf{bag}$.

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma \vdash e' : \mathsf{T}'E}{\Gamma \vdash e \texttt{ except } e' : \mathsf{T}E} \text{ (Ty-Except)}$$

$$\begin{aligned} & \mathscr{Q}\,(e \texttt{ except } e') \\ = \;& [\![\, x \,|\, x \leftarrow \mathscr{Q}\,e, [\![\, x \neq x' \,|\, x' \leftarrow \mathscr{Q}\,e' \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{T}} \qquad \mathscr{Q}\text{-Except} \end{aligned}$$

Union and concatenation (`union` and `+`). OQL designates symbol `+` to denote list concatenation if $\mathsf{T} = \mathsf{T}' = \mathsf{list}$. If (τ, T) and (τ', T') denote the initial types associated with type functors T and T', define $\mathsf{T}'' = \mathsf{T}$ if τ obeys all equations that τ' obeys, otherwise set $\mathsf{T}'' = \mathsf{T}'$.

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma \vdash e' : \mathsf{T}'E}{\Gamma \vdash e \texttt{ union } e' : \mathsf{T}''E} \text{ (Ty-Union)}$$

$$\begin{aligned} & \mathscr{Q}\,(e \texttt{ union } e') \\ = \;& \mathit{join}^{\mathsf{T}''}\left((\!|\, [\,] \,|\!)^{\mathsf{T}''} \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e) \;{}^{\mathsf{T}''}_{\vdots}\; (\!|\, [\,] \,|\!)^{\mathsf{T}''} \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e') \;{}^{\mathsf{T}''}_{\vdots}\; [\,]^{\mathsf{T}''} \right) \qquad \mathscr{Q}\text{-Union} \end{aligned}$$

Notes. By inlining the definition of $\mathit{join}^{\mathsf{T}''}$ and with the help of law Acid-Rain (Chapter 6), this initial translation may be simplified to read

$$(\!|\, (\!|\, [\,] \,|\!)^{\mathsf{T}''} \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e')) \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e)\ ,$$

which, for the case $\mathsf{T}'' = \mathsf{T} = \mathsf{T}'$, further reduces to

$$\begin{aligned} & (\!|\, (\!|\, [\,] \,|\!)^{\mathsf{T}''} \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e')) \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e) \\ = \;& \quad \{\ \text{Cata-Reflect}\ \} \\ & (\!|\, \mathscr{Q}\,e' \triangledown^{\mathsf{T}''}_{\vdots} \,|\!) \,(\mathscr{Q}\,e) \\ = \;& \quad \{\, \text{definition of } \# \text{ for } \mathsf{T}'' \text{ (Lemma 60)} \,\} \\ & (\mathscr{Q}\,e) \# (\mathscr{Q}\,e')\ . \end{aligned}$$

Universal quantification (`forall-in`).

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma, v : E \vdash p : \mathit{Bool}}{\Gamma \vdash \texttt{forall}\ v\ \texttt{in}\ e\texttt{:}\ p : \mathit{Bool}}\ (\textsc{Ty-Forall})$$

$$\mathscr{Q}\,(\texttt{forall}\ v\ \texttt{in}\ e\texttt{:}\ p) = [\![\, \mathscr{Q}\,p \,]\!]\, v \leftarrow \mathscr{Q}\,e \,]\!]^{\mathsf{all}} \qquad \mathscr{Q}\text{-}\textsc{Forall}$$

Existential quantification (`exists-in`).

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma, v : E \vdash p : \mathit{Bool}}{\Gamma \vdash \texttt{exists}\ v\ \texttt{in}\ e\texttt{:}\ p : \mathit{Bool}}\ (\textsc{Ty-Exists})$$

$$\mathscr{Q}\,(\texttt{exists}\ v\ \texttt{in}\ e\texttt{:}\ p) = [\![\, \mathscr{Q}\,p \,]\!]\, v \leftarrow \mathscr{Q}\,e \,]\!]^{\mathsf{exists}} \qquad \mathscr{Q}\text{-}\textsc{Exists}$$

OQL defines alternative syntactic forms for the quantifiers which we rewrite so that they become subject to $\mathscr{Q}$-Forall or $\mathscr{Q}$-Exists. As before, $\circledcirc$ serves as a placeholder for the operators `<`, `<=`, `=`, `!=`, `>=`, or `>`.

$$\frac{\Gamma \vdash e : E \qquad \Gamma \vdash e' : \mathsf{T}E}{\Gamma \vdash e \circledcirc \texttt{all}\ e' : \mathit{Bool}} \qquad \frac{\Gamma \vdash e : E \qquad \Gamma \vdash e' : \mathsf{T}E}{\Gamma \vdash e \circledcirc \texttt{some}\ e' : \mathit{Bool}}$$

$$\mathscr{Q}\,(e \circledcirc \texttt{all}\ e') = \mathscr{Q}\,(\texttt{forall}\ v\ \texttt{in}\ e'\texttt{:}\ e \circledcirc v)$$
$$\mathscr{Q}\,(e \circledcirc \texttt{some}\ e') = \mathscr{Q}\,(\texttt{exists}\ v\ \texttt{in}\ e'\texttt{:}\ e \circledcirc v)$$

$$\frac{\Gamma \vdash e : E \qquad \Gamma \vdash e' : \mathsf{T}E}{\Gamma \vdash e\ \texttt{in}\ e' : \mathit{Bool}}$$

$$\mathscr{Q}\,(e\ \texttt{in}\ e') = \mathscr{Q}\,(\texttt{exists}\ v\ \texttt{in}\ e'\texttt{:}\ e = v)$$

Test for non-empty collections (`exists`).

$$\frac{\Gamma \vdash e : \mathsf{T}E}{\Gamma \vdash \texttt{exists}(e) : \mathit{Bool}}\ (\textsc{Ty-Peek})$$

$$\mathscr{Q}\,(\texttt{exists}(e)) = [\![\, \mathit{true} \,|\, x \leftarrow \mathscr{Q}\,e \,]\!]^{\mathsf{exists}} \qquad \mathscr{Q}\text{-}\textsc{Peek}$$

Notes. This translation may suggest to scan collection e completely just to determine whether e contains at least one arbitrary element. There are two points to note here. First, remember that $\mathscr{Q}$'s purpose is to assign meaning to queries, not execution plans. Second, depending on the execution model implemented by the query engine, $\mathscr{Q}$-Peek actually is as efficient a translation as one may hope for. To see this, consider the calculation

$$\begin{aligned}
& (\mathscr{M} \cdot \mathscr{Q})\,(\texttt{exists}(e)) \\
= \quad & \{\, \mathscr{Q}\text{-}\textsc{Peek} \,\} \\
& \mathscr{M}\,[\![\, \mathit{true} \,|\, x \leftarrow \mathscr{Q}\,e \,]\!]^{\mathsf{exists}} \\
= \quad & \{\, \mathscr{M}\text{-}3 \,\} \\
& (\!|\mathit{false} \triangledown \vee|\!)\,(\mathscr{M}\,[\![\, \mathit{true} \,|\, x \leftarrow \mathscr{Q}\,e \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \mathscr{M}\text{-}2 \,\} \\
& ((\!|\mathit{false} \triangledown \vee|\!) \cdot (\mathsf{T}\,K_{\mathit{true}}))\,(\mathscr{Q}\,e) \\
= \quad & \{\, \textsc{Cata-Map-Fusion} \,\} \\
& (\!|(\mathit{false} \triangledown \vee) \cdot (\mathit{id} + K_{\mathit{true}} \times \mathit{id})|\!)\,(\mathscr{Q}\,e) \\
= \quad & \{\, \text{product},\ \mathit{true} \vee x = \mathit{true} \,\} \\
& (\!|\mathit{false} \triangledown K_{\mathit{true}}|\!)\,(\mathscr{Q}\,e)\ .
\end{aligned}$$

A database engine that implements query operators following a *lazy* (or *on-demand*) stream iterator model [18, 54] will demand evaluation of $\mathscr{Q}\,e$ not beyond the point at which it gets hold of the top-most constructor ($[\,]^{\mathsf{T}}$ or $\overset{!}{:}$) of the spine of $\mathscr{Q}\,e$. In terms of functional programming languages, it is sufficient to evaluate $\mathscr{Q}\,e$ to *weak head normal form* [18, 92]. We will have to say a lot more about streaming in Chapter 6.

Test for singleton collections (`unique`).

$$\frac{\Gamma \vdash e : \mathsf{T}E}{\Gamma \vdash \texttt{unique}(e) : \mathit{Bool}}\ (\textsc{Ty-Unique})$$

$$\mathscr{Q}\,(\texttt{unique}(e)) = (= \cdot (K_{[\,]^{\mathsf{T}}} \vartriangle \mathit{tl}))\,(\mathscr{Q}\,e) \qquad \mathscr{Q}\text{-}\textsc{Unique}$$

(tl defined as before).

Subset relationship (<=). The monad comprehension below implements the semantics of the subset relationship predicate $\subseteq$. The typing allows for the application to arbitrary collection type formers but multiplicity or order of elements is not accounted for.

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \Gamma \vdash e' : \mathsf{T}'E}{\Gamma \vdash e \texttt{ <= } e' : \mathit{Bool}} \text{ (Ty-Subset)}$$

$$\mathscr{Q}\,(e \texttt{ <= } e') = [\![\, [\![\, x = x' \,]\!] \; x' \leftarrow \mathscr{Q}\, e' \,]\!]^{\mathsf{exists}} \; [\![\, x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{all}} \qquad \mathscr{Q}\text{-Subset}$$

Summation (sum).

$$\frac{\Gamma \vdash e : \mathsf{T}\mathit{Num} \qquad \mathsf{T} \neq \mathsf{set}}{\Gamma \vdash \texttt{sum}(e) : \mathit{Num}} \text{ (Ty-Sum)}$$

$$\mathscr{Q}\,(\texttt{sum}(e)) = [\![\, x \,]\!] \; x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{sum}} \qquad \mathscr{Q}\text{-Sum}$$

Counting (count).

$$\frac{\Gamma \vdash e : \mathsf{T}E \qquad \mathsf{T} \neq \mathsf{set}}{\Gamma \vdash \texttt{count}(e) : \mathit{Num}} \text{ (Ty-Count)}$$

$$\mathscr{Q}\,(\texttt{count}(e)) = [\![\, K_1\, x \,]\!] \; x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{sum}} \qquad \mathscr{Q}\text{-Count}$$

Average (avg).

$$\frac{\Gamma \vdash e : \mathsf{T}\mathit{Num} \qquad \mathsf{T} \neq \mathsf{set}}{\Gamma \vdash \texttt{avg}(e) : \mathit{Num}} \text{ (Ty-Avg)}$$

$$\begin{aligned} & \mathscr{Q}\,(\texttt{avg}(e)) \\ = \; & \big(/ \cdot ((\lambda v.\mathscr{Q}\,(\texttt{sum}(v))) \vartriangle (\lambda v.\mathscr{Q}\,(\texttt{count}(v))))\big)\,(\mathscr{Q}\, e) \end{aligned} \qquad \mathscr{Q}\text{-Avg}$$

Notes. After recursive invocation of $\mathscr{Q}$ and subsequent monad comprehension desugaring, the translated query essentially exhibits the typical shape of a *tupling catamorphism* $(\!|f|\!) \vartriangle (\!|g|\!)$. Operationally speaking, tupling catamorphisms walk their argument twice but the equational theory for catamorphisms once more provides a hook for optimization, this time in terms of law

BANANA-SPLIT: for a tupling catamorphism $(\!|f|\!)_{\mathsf{F}} \vartriangle (\!|g|\!)_{\mathsf{F}}$ whose components are F-catamorphisms, it is true that

$$(\!|f|\!)_{\mathsf{F}} \vartriangle (\!|g|\!)_{\mathsf{F}} \quad = \quad (\!|(f \cdot \mathsf{F}\mathit{outl}) \vartriangle (g \cdot \mathsf{F}\mathit{outr})|\!)_{\mathsf{F}} \qquad \text{BANANA-SPLIT}$$

(a proof may be found in several places, including [67] and [42]).

For an `avg` query, BANANA-SPLIT yields the catamorphism

$$(\!|(K_0 \vartriangle K_0) \triangledown ((+ \cdot (\mathit{id} \times \mathit{outl})) \vartriangle (+ \cdot (K_1 \times \mathit{outr})))|\!) \,,$$

which, using a single traversal of $\mathscr{Q}\, e$, computes a (nominator, denominator) pair from which the average is immediate.

Maximum and minimum (`max` and `min`). Types instantiating E are required to be ordered.

$$\frac{\Gamma \vdash e : \mathsf{T}E}{\Gamma \vdash \mathtt{max}(e) : E}\,(\text{TY-MAX}) \qquad \frac{\Gamma \vdash e : \mathsf{T}E}{\Gamma \vdash \mathtt{min}(e) : E}\,(\text{TY-MIN})$$

$$\mathscr{Q}\,(\mathtt{max}(e)) = [\![\, x \mid\!]\, x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{max}} \qquad \mathscr{Q}\text{-MAX}$$
$$\mathscr{Q}\,(\mathtt{min}(e)) = [\![\, x \mid\!]\, x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{min}} \qquad \mathscr{Q}\text{-MIN}$$

Flattening (`flatten`). If (τ, T) and (τ', T') denote the initial types associated with type functors T and T', define $\mathsf{T}'' = \mathsf{T}$ if τ obeys all equations that τ' obeys, otherwise set $\mathsf{T}'' = \mathsf{T}'$.

$$\frac{\Gamma \vdash e : \mathsf{T}\mathsf{T}'E}{\Gamma \vdash \mathtt{flatten}(e) : \mathsf{T}''E}\,(\text{TY-FLATTEN})$$

$$\mathscr{Q}\,(\mathtt{flatten}(e)) = [\![\, x' \mid\!]\, x \leftarrow \mathscr{Q}\, e, x' \leftarrow x \,]\!]^{\mathsf{T}''} \qquad \mathscr{Q}\text{-FLATTEN}$$

Note. An appeal to $\mathscr{M}$ and the functoriality of T'' reveals OQL's `flatten` operator to simply be the monadic $\mathit{join}^{\mathsf{T}''}$ if $\mathsf{T}'' = \mathsf{T} = \mathsf{T}'$.

Duplicate elimination (`listtoset` and `distinct`).

$$\frac{\Gamma \vdash e : \mathsf{list}E}{\Gamma \vdash \mathtt{listtoset}(e) : \mathsf{set}E}\,(\text{TY-DUPELIM-1})$$

$$\frac{\Gamma \vdash e : \mathsf{bag}E}{\Gamma \vdash \mathtt{distinct}(e) : \mathsf{set}E}\,(\text{TY-DUPELIM-2})$$

$$\mathscr{Q}\,(\mathtt{listtoset}(e)) = [\![\, x \mid\!]\, x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{set}} \qquad \mathscr{Q}\text{-DUPELIM-1}$$
$$\mathscr{Q}\,(\mathtt{distinct}(e)) = [\![\, x \mid\!]\, x \leftarrow \mathscr{Q}\, e \,]\!]^{\mathsf{set}} \qquad \mathscr{Q}\text{-DUPELIM-2}$$

Extraction (`element` and `first`). OQL designates the keyword `first` for the case $\mathsf{T} = \mathsf{list}$.

$$\frac{\Gamma \vdash e : \mathsf{T}E}{\Gamma \vdash \texttt{element(}e\texttt{)} : E}\,(\textsc{Ty-Element})$$

$$\mathscr{Q}\,(\texttt{element(}e\texttt{)}) = (\!|\, \bot \triangledown \mathit{outl} \,|\!)\,(\mathscr{Q}\,e) \qquad \mathscr{Q}\textsc{-Element}$$

Notes. The catamorphism yields $\bot$ should e evaluate to $[\,]^{\mathsf{T}}$. Here, the *bottom* symbol $\bot$ (denoting a completely undefined value) signals a situation in which the query engine should raise an exception according to the specification in [23]. For the above to work we require *outl* to be *non-strict* in the second component of its argument, *i. e.*, $\mathit{outl}\,(x, \bot) = x$. //

83 Arrays. As presented in Definition 82, mapping $\mathscr{Q}$ lacks support for OQL's `array` type former and its associated query primitives, most notably array subscripting via `[·]`. Support for arrays in a setting of categorical datatypes and catamorphic computation raises no principal problems—indeed, a number of efforts in that direction have already been undertaken [5, 16, 40, 77].

An insert representation of arrays is conceivable but leads to a rather awkward handling of arrays as either (a) each insertion affects the array *shape* (dimensionality and extension of each dimension) or (b) the array needs to hold *empty slots* on creation into which array elements are to be inserted later. Arrays are more naturally constructed in a bulk-oriented fashion. In [5], arrays with element type E are constructed by the initial algebra

$$\sigma \triangledown \rho : \mathsf{list}E + \mathit{Num} \times \mathsf{array}E \to \mathsf{array}E$$

(where array denotes the array type functor). An array is constructed from a potentially infinite list of values of type E. Rather than on the array itself, the constructors operate on the shape of the array and thus describe how the array is to be built from the infinite pool of values supplied by the list: an application of the *scalarize* constructor $\sigma\,[x_0, x_1, \ldots]^{\mathsf{list}}$ builds the 0-dimensional (scalar) array consisting of element x_0 only; *redimensioning* $\rho\, n$ raises the dimensionality of the array by 1 (the new dimension has extension n): $((\rho\, 3) \cdot \sigma)\,[x_0, x_1, \ldots]^{\mathsf{list}}$ constructs a 1-dimensional array containing elements x_0, x_1, x_2; $(\rho\, m) \cdot (\rho\, n) \cdot \sigma$ denotes the $(n \times m)$-matrix constructor, and so on.

It is this rather different nature of arrays which led us to exclude their treatment in this text. The details of their coverage would blur much of the conceptual simplicity (*e. g.*, the need for the single insert representation functor $\mathsf{F} = \mathsf{K}_1 + \mathsf{K}_E \times \mathsf{Id}$, only) we benefit from in this stage of discussion.

84 The catamorphisms (primarily in the disguise of monad comprehensions) emitted by *2* provide a somewhat unfamiliar view of database queries. Much of this is due to the uniform catamorphic representation of query primitives: the encoding of projections, joins, and selections (*e. g.*, the translation of a `select-from-where` block) is subject to inspection as is the representation of quantification or element extraction.

This is unlike the situation we face with "classical" query formalisms, relational algebra or calculus, say. User-level query language primitives that lie outside the reach—in terms of expressiveness—of the relational operators find their way into the relational query representation as *black box* operators or function calls. During the query rewrite phase these black boxes may be merely moved around as their algebraic properties are not known. Their interaction with the well-understood optimization rules in the relational domain remains unclear. Much may be missed that way.

To prove the catamorphic (monadic) comprehension of queries viable, there are thus two questions on our agenda:

- How is already well-established and provably useful knowledge on query optimization expressed in the catamorphic model? Can we mimic these approaches? We try to find answers to these questions starting with the section on query normalization in the current chapter and the material on query combinators in Chapter 4. Actually, we hope to provide more than just mimicry: in places, the categorical query model can simplify and unify previous ideas.

- Where does the catamorphic comprehension of queries give additional insight? Can the categorical query model answer previously open questions? We explore the now newly opened opportunities for query comprehension throughout this text, especially in Chapters 5 and 6.

Before we go on to work on the items of this agenda, let us spend the next few paragraphs and try to give a taste of what kind of possibilities for optimization show up if queries are tackled using a categorical toolbox.

85 Different placements of OQL's extraction operator `element` (or `first`, respectively) in a query may obviously influence the quality of a query plan in dramatic ways. Early execution of element extraction can lead to removal of joins or even query unnesting. Join removal, *e. g.*, is possible for the OQL query below (remember the convention that a query expression like $f\,x\,y$ denotes a query f containing free variables whose occurences are replaced by

x, y):

$$\texttt{element(select } f\, x\, y \texttt{ from } xs \texttt{ as } x\texttt{, } ys \texttt{ as } y\texttt{)} \ .$$

Computing the cartesian product of xs and ys is wasted work as we are throwing the result away should the product unexpectedly contain more than a single element (in which case the query raises an exception). An `element`-aware query optimizer could emit the equivalent

$$f\,(\texttt{element(}xs\texttt{)})\,(\texttt{element(}ys\texttt{)})$$

which removes the need for the cartesian product and thus never wastes work. Note that the simplified query raises an exception if and only if the original form would. However, `element` pushdown has a perilous nature:

- The above rewrite does not preserve equivalence if we are computing with sets (replace `select` by `select distinct`): function f might not be one-to-one and duplicate removal could potentially reduce the product to hold exactly one element.
- We must not push element extraction beyond a selection as the selection might filter its input so that exactly one element passes (*e. g.*, selection on a key).

This raises at least two questions. How are the safe rewrites recognizable and, in safe cases, how do we obtain the optimized form?

This is where the theory jumps in. Once we have observed that OQL's element extraction operators act like *monad morphisms* we can answer both questions.

86 Definition. Given two monads T, T', an arrow $h\colon \mathsf{T}E \to \mathsf{T}'E$ is called a **monad morphism** [117] if it interacts with the operations of T and T' as follows:

$$h \cdot \mathsf{T}f = \mathsf{T}'f \cdot h \qquad \textsc{Monad-Morph-1}$$
$$h \cdot unit^{\mathsf{T}} = unit^{\mathsf{T}'} \qquad \textsc{Monad-Morph-2}$$
$$h \cdot join^{\mathsf{T}} = join^{\mathsf{T}'} \cdot \mathsf{T}'h \cdot h \ . \qquad \textsc{Monad-Morph-3}$$

Should T and T' be monads with zero, we additionally require

$$h \cdot zero^{\mathsf{T}} = zero^{\mathsf{T}'} \ . \qquad \textsc{Monad-Morph-4}$$

//

87 Element extraction acts like a monad morphism from a collection monad T to the identity monad Id (here we take the freedom to mix OQL keywords and formal notation, having in mind that $\mathscr{Q}$-Element establishes the correspondence $\texttt{element} = (\!|\, \bot \triangledown \mathit{outl} \,|\!)$):

$$\texttt{element} \cdot \mathsf{T} f = f \cdot \texttt{element} \qquad \text{Elem-Pushdown-1}$$
$$\texttt{element} \cdot \mathit{unit}^{\mathsf{T}} = \mathit{id} \qquad \text{Elem-Pushdown-2}$$
$$\texttt{element} \cdot \mathit{join}^{\mathsf{T}} = \texttt{element} \cdot \texttt{element}\,. \qquad \text{Elem-Pushdown-3}$$

Two restrictions apply. First, Elem-Pushdown-1 does not hold for $\mathsf{T} = \mathsf{set}$ in general but only for constant f (*i. e.*, $f = K_x$ for some x). Second, `element` is not a morphism between monads with zero since $\texttt{element} \cdot \mathit{zero}^{\mathsf{T}} = \bot \neq \mathit{zero}^{\mathsf{Id}}$.

Observe that `element` fails to act like a monad morphism in exactly those two situations that we have identified as unsafe pushdown rewritings in Paragraph 85. This suggests to exploit the above three equivalences as rewritings that propagate `element` through the monad operations and use these with $\mathscr{Q}$ as a basis for safe `element` pushdown.

For the example query of Paragraph 85 we get

$$\begin{aligned}
& (\mathscr{M} \cdot \mathscr{Q}) \left(\begin{array}{l} \texttt{element(select } f\ x\ y \\ \qquad\qquad \texttt{from } xs \texttt{ as } x\texttt{, } ys \texttt{ as } y\texttt{)} \end{array} \right) \\
= \quad & \{\, \mathscr{Q}\text{-Element}, \mathscr{Q}\text{-Sfw} \,\} \\
& \texttt{element}\, (\mathscr{M}\, [\![\, f\ x\ y \mid x \leftarrow xs, y \leftarrow ys \,]\!]^{\mathsf{bag}}) \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& (\texttt{element} \cdot \mathit{join}^{\mathsf{bag}})\, (\mathscr{M}\, [\![\, [\![\, f\ x\ y \mid y \leftarrow ys \,]\!]^{\mathsf{bag}} \mid x \leftarrow xs \,]\!]^{\mathsf{bag}}) \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& (\texttt{element} \cdot \mathit{join}^{\mathsf{bag}})\, (\mathsf{bag}\, (\lambda x.\mathscr{M}\, [\![\, f\ x\ y \mid y \leftarrow ys \,]\!]^{\mathsf{bag}})\, xs) \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& (\texttt{element} \cdot \mathit{join}^{\mathsf{bag}})\, (\mathsf{bag}\, (\lambda x.(\mathsf{bag}\, (\lambda y.f\ x\ y)\ ys))\, xs) \\
= \quad & \{\, \text{Elem-Pushdown-3} \,\} \\
& (\texttt{element} \cdot \texttt{element} \cdot \mathsf{bag})\, ((\lambda x.(\mathsf{bag}\, (\lambda y.f\ x\ y)\ ys))\, xs) \\
= \quad & \{\, \text{Elem-Pushdown-1} \,\} \\
& \texttt{element}\, ((\lambda x.(\mathsf{bag}\, (\lambda y.f\ x\ y)\ ys))\, (\texttt{element}\ xs)) \\
= \quad & \{\, \beta\text{-reduction} \,\} \\
& (\texttt{element} \cdot \mathsf{bag})\, ((\lambda y.f\, (\texttt{element}\ xs)\ y)\ ys) \\
= \quad & \{\, \text{Elem-Pushdown-1} \,\} \\
& (\lambda y.f\, (\texttt{element}\ xs)\ y)\, (\texttt{element}\ ys)
\end{aligned}$$

$$
\begin{array}{ll}
= & \quad \{\, \beta\text{-reduction} \,\} \\
& f\,(\texttt{element}\;xs)\,(\texttt{element}\;ys)
\end{array}
$$

and thus obtain the optimized form we were after. The propagation rules (Elem-Pushdown-1 through Elem-Pushdown-3) push element extraction down as far as possible but not beyond filters (which are implemented with the help of monadic zeroes for which no propagation rule applies). We can see this even more easily if we exploit the syntactic sugar we have introduced for monadic computations: comprehension syntax.

In [117], Wadler observed that the action of a monad morphism on a monadic computation may be concisely described by way of comprehension syntax. A monad morphism $h : \mathsf{T}E \to \mathsf{T}'E$ acts on a T-monad comprehension as follows:

$$
h\,[\![\, e \mid qs \,]\!]^{\mathsf{T}} = [\![\, e \mid h\;qs \,]\!]^{\mathsf{T}'} \qquad \text{Monad-Morph-5}
$$

where h's extension on a qualifier list qs is defined by (ε and identifiers v, e, qs, qs' represent the syntactic categories of Definition 67)

$$
\begin{array}{rcll}
h\,\varepsilon & = & \varepsilon & \text{empty} \\
h\,(v \leftarrow e) & = & v \leftarrow h\,e & \text{generators} \\
h\,(qs, qs') & = & h\;qs, h\;qs' & \text{qualifier lists} \\
h\,e & = & e\,. & \text{filters}
\end{array}
$$

For $h = \texttt{element}$, these simple syntactical rewritings exactly express the `element` pushdown optimization and are therefore especially suitable for inclusion into a rule-based optimizer. For our former example we calculate:

$$
\begin{array}{ll}
& \texttt{element}\,[\![\, f\;x\;y \mid x \leftarrow xs, y \leftarrow ys \,]\!]^{\mathsf{bag}} \\
= & \quad \{\, \text{Monad-Morph-5} \,\} \\
& [\![\, f\;x\;y \mid \texttt{element}\,(x \leftarrow xs, y \leftarrow ys) \,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \text{Monad-Morph-5} \,\} \\
& [\![\, f\;x\;y \mid \texttt{element}\,(x \leftarrow xs), \texttt{element}\,(y \leftarrow ys) \,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \text{Monad-Morph-5} \,\} \\
& [\![\, f\;x\;y \mid x \leftarrow \texttt{element}\;xs, y \leftarrow \texttt{element}\;ys \,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \text{comprehension in } \mathsf{Id} \text{ monad} \,\} \\
& f\,(\texttt{element}\;xs)\,(\texttt{element}\;ys)
\end{array}
$$

(for the last step remember that a monad comprehension in the Id-monad essentially describes a nested *let*-expression, *cf.* Example 71).

Optimizations like these are indeed relevant. Early execution of element extraction can decrease the nesting depth of a query as the final example shows:

$$
\begin{array}{ll}
 & \mathscr{2}\left(\begin{array}{l}\texttt{element(select (select } f\ x\ y \\ \qquad\qquad\qquad\qquad \texttt{from } ys \texttt{ as } y) \\ \qquad\qquad \texttt{from } xs \texttt{ as } x)\end{array}\right) \\
= & \quad \{\, \mathscr{2} \,\} \\
 & \texttt{element}\, [\![\, [\![\, f\ x\ y \,]\!]\ y \leftarrow ys \,]\!]^{\mathsf{bag}}\ [\![\, x \leftarrow xs \,]\!]^{\mathsf{bag}} \\
= & \quad \{\, \text{MONAD-MORPH-5} \,\} \\
 & [\![\, [\![\, f\ x\ y \,]\!]\ y \leftarrow ys \,]\!]^{\mathsf{bag}}\ [\![\, \texttt{element}\ (x \leftarrow xs) \,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \text{MONAD-MORPH-5} \,\} \\
 & [\![\, [\![\, f\ x\ y \,]\!]\ y \leftarrow ys \,]\!]^{\mathsf{bag}}\ [\![\, x \leftarrow \texttt{element}\ xs \,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \text{comprehension in } \mathsf{Id} \text{ monad} \,\} \\
 & [\![\, f\ (\texttt{element}\ xs)\ y \,]\!]\ y \leftarrow ys \,]\!]^{\mathsf{bag}}\ .
\end{array}
$$

The last monad comprehension has the equivalent OQL form

$$
\begin{array}{l}\texttt{select } f\ (\texttt{element(}xs\texttt{)})\ y \\ \quad \texttt{from } ys \texttt{ as } y\ ,\end{array}
$$

i. e., a query which simply maps f over collection ys instead of creating a nested bag of bags like the original form did.

88 It is rather straightforward to check that OQL's duplicate elimination primitives `listtoset` and `distinct` act like monad morphisms from the list and bag monads to the set monad, respectively. In other words, the function $\texttt{listtoset} = \lambda e.[\![\, x \,]\!]\ x \leftarrow e \,]\!]^{\mathsf{set}} : \mathsf{list}E \to \mathsf{set}E$ fulfills laws MONAD-MORPH-1 through MONAD-MORPH-4 and thus law MONAD-MORPH-5. An analogous remark applies to `distinct`.

We can exploit this observation to *delay duplicate removal* inside the query engine.

- To delay the removal of duplicates may seem counter-productive as, in general, duplicate elimination reduces operand sizes and thus might save execution costs. Duplicate removal, however, is a rather expensive operation to perform as it requires a sort or hash pass to identify duplicate elements. This, in turn, may disrupt the desirable stream-based processing of queries (to which we will turn in Chapter 6).

- Second, if we take a look at the innards of a query engine implemented on a common platform, actual implementations are almost exclusively list-based. This is partly due to the just mentioned cost inherent to duplicate removal (which manifests itself in the $\overset{\mathsf{set}}{\vdots}$ constructor) but additionally enables the engine to reason about interesting (sort) orders [102] of the streams it processes.

The query rewrite that enables delay of duplicate elimination is justified by the equivalence (let $xs : \mathsf{list}E$, we could reason analogously for $xs : \mathsf{bag}E$ and `distinct`)

$$[\![\, e \mid qs, x \leftarrow xs, qs' \,]\!]^{\mathsf{set}} \quad = \quad \texttt{listtoset}\, [\![\, e \mid qs, x \leftarrow xs, qs' \,]\!]^{\mathsf{list}} .$$

The right-hand side evaluates the comprehension in the list monad which preserves an potentially interesting sort order of xs until duplicates are finally removed by the outer `listtoset` primitive.

We could verify the correctness of this transformation by means of the monad morphism properties of `listtoset`. We will not do so here but rather appeal to monad comprehension *unnesting* which will be the focus of the forthcoming section. The correctness of delayed duplicate removal is immediately asserted by comprehension *nesting*:

$$\begin{aligned}
& [\![\, e \mid qs, x \leftarrow xs, qs' \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{nesting via } \mathscr{M}\text{-Norm-3} \,\} \\
& [\![\, x' \mid x' \leftarrow [\![\, e \mid qs, x \leftarrow xs, qs' \,]\!]^{\mathsf{list}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{definition of } \texttt{listtoset} \,\} \\
& \texttt{listtoset}\, [\![\, e \mid qs, x \leftarrow xs, qs' \,]\!]^{\mathsf{list}} .
\end{aligned}$$

Unnesting plays a crucial role during monad comprehension *normalization* to which we will turn now.

3.2 Monad Comprehension Normalization

89 Being a completely syntax-directed translation, $\mathscr{Q}$ rather directly reflects the nested structure of an OQL query in the comprehensions it emits. Deriving anything but nested loop forms from a deeply nested query expression is a considerably hard task and a widely recognized challenge in the query processing literature [35, 48, 69, 105, 107]. In order to make good use of the efficient processing primitives of an underlying query engine we are better off to unravel nested queries.

The monad comprehension calculus provides particularly efficient yet simple handles to attack the problem.

- Different types of query nesting lead to similar, if not identical, nested forms of monad comprehensions. Rather than to maintain and identify a number of special cases of nesting—this route has been taken by numerous approaches, notably Kim's original and followup work on classifying nested SQL queries [48, 69]—we can concentrate on unnesting the relatively few comprehension forms.
- Much of the actual unnesting work can be achieved by the application of a small set of *normalization rules* to monad comprehensions. Despite its simplicity, comprehension normalization covers a wide range of query unnesting techniques.

The normalization rules will prove to be valuable transformation tools in general (above we have already used one of these rules—more specifically its dual, nesting rule—to justify delayed duplicate removal).

90 The normalization rule set is assembled from a number of sources. Wadler already described the core rule $\mathscr{M}$-Norm-3 in his original work on endo-monadic comprehensions [117]. We augment this rule with a number of comprehension transformations that have appeared in the work of Wong and Fegaras [38, 124].

Here, we will put these rules to use and confirm their correctness in the multi-monad comprehension framework. An assessment of the soundness of the rules is crucial. The past has shown that relational unnesting is a query transformation already complex enough to let incorrect rewritings—*e. g.*, the infamous `count` bug of [69]—remain undiscovered for long.

91 Definition. The following set of equivalences is referred to as the **monad comprehension normalization rules** (an equivalence defines a normalization rule if it is read from left to right—interpreted from right to left we obtain the *dual* or *nesting* rule). Let $\mathsf{T}' \neq \mathsf{Id}$.

$$
\begin{aligned}
[\![\, e \mid qs, v \leftarrow zero^{\mathsf{T}'}, qs' \,]\!]^{\mathsf{T}} &= zero^{\mathsf{T}} && \mathscr{M}\text{-Norm-1}\\
[\![\, e \mid qs, v \leftarrow unit^{\mathsf{T}'}\, e', qs' \,]\!]^{\mathsf{T}} &= [\![\, e[e'/v] \mid qs, qs'[e'/v] \,]\!]^{\mathsf{T}} && \mathscr{M}\text{-Norm-2}\\
[\![\, e \mid qs, v \leftarrow [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'}, qs' \,]\!]^{\mathsf{T}} &= [\![\, e[e'/v] \mid qs, qs'', qs'[e'/v] \,]\!]^{\mathsf{T}} && \mathscr{M}\text{-Norm-3}\\
[\![\, e \mid qs, [\![\, e' \mid qs'' \,]\!]^{\mathsf{exists}}, qs' \,]\!]^{\mathsf{T}} &= [\![\, e \mid qs, qs'', e', qs' \,]\!]^{\mathsf{T}} && \mathscr{M}\text{-Norm-4}
\end{aligned}
$$

$$[\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{Id}} \mid qs \,]\!]^{\mathsf{Id}} \quad = \quad [\![\, e \mid qs, qs' \,]\!]^{\mathsf{Id}} \qquad \begin{matrix}(\overset{\mathsf{T}}{!}\text{ left-idempotent}) \\ \mathscr{M}\text{-Norm-5}\end{matrix}$$

∎

92 Theorem. The set of monad comprehension normalization rules is sound, *i. e.*, the left-hand and right-hand sides of each rule map to equivalent monadic values under $\mathscr{M}$.

Proof. We will only prove rules $\mathscr{M}$-Norm-3 and $\mathscr{M}$-Norm-5 valid here. The structure of the proofs for the other three rules closely follows that of $\mathscr{M}$-Norm-3 and thus provide few new insights. Along the way of establishing the correctness of $\mathscr{M}$-Norm-3, however, we will invoke a transformation (validated in Lemma 93) that will be useful in its own right later on.

If we slightly specialize the comprehension qualifier list by setting $qs = \varepsilon$ we can find a rather simple proof for rule $\mathscr{M}$-Norm-3. Being so far, we establish the proof in a second step by generalizing from $qs = \varepsilon$ to arbitrary qs, assuming the specialized rule to be valid.

Set $qs = \varepsilon$. Let, as usual, τ denote the initial algebra of the initial type (τ, T).

$$\begin{aligned}
& \mathscr{M}\, [\![\, e \mid v \leftarrow [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'}, qs' \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& \mathit{join}^{\mathsf{T}}\, (\mathscr{M}\, [\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'} \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& (\mathit{join}^{\mathsf{T}} \cdot (\!|\tau|\!))\, (\mathscr{M}\, [\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'} \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \mathscr{M}\text{-2} \,\} \\
& (\mathit{join}^{\mathsf{T}} \cdot (\!|\tau|\!))\, (\mathsf{T}'\, (\lambda v.\mathscr{M}\, [\![\, e \mid qs' \,]\!]^{\mathsf{T}})\, (\mathscr{M}\, [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'})) \\
= \quad & \{\, \text{Functor-}\mathscr{M}\text{-Fusion}, \text{ established below} \,\} \\
& (\mathit{join}^{\mathsf{T}} \cdot (\!|\tau|\!))\, (\mathscr{M}\, [\![\, [\![\, e[e'/v] \mid qs'[e'/v] \,]\!]^{\mathsf{T}} \mid qs'' \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& \mathit{join}^{\mathsf{T}}\, (\mathscr{M}\, [\![\, [\![\, e[e'/v] \mid qs'[e'/v] \,]\!]^{\mathsf{T}} \mid qs'' \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& \mathscr{M}\, [\![\, e[e'/v] \mid qs'', qs'[e'/v] \,]\!]^{\mathsf{T}}\,.
\end{aligned}$$

Generalize to arbitrary qualifier list qs:

$$\mathscr{M}\, [\![\, e \mid qs, v \leftarrow [\![\, e' \mid qs'' \,]\!]^{\mathsf{T}'}, qs' \,]\!]^{\mathsf{T}}$$

$$
\begin{array}{ll}
= & \quad \{\, \mathscr{M}\text{-5} \,\} \\
& \mathit{join}^{\mathsf{T}}\,(\mathscr{M}\,[\![\,[\![\,e \parallel v \leftarrow [\![\,e' \parallel qs''\,]\!]^{\mathsf{T}'}, qs'\,]\!]^{\mathsf{T}} \parallel qs\,]\!]^{\mathsf{T}}) \\
= & \quad \{\, \mathscr{M}\text{-Norm-3 with } qs = \varepsilon \,\} \\
& \mathit{join}^{\mathsf{T}}\,(\mathscr{M}\,[\![\,[\![\,e[e'/v] \parallel qs'', qs'[e'/v]\,]\!]^{\mathsf{T}} \parallel qs\,]\!]^{\mathsf{T}}) \\
= & \quad \{\, \mathscr{M}\text{-5} \,\} \\
& \mathscr{M}\,[\![\,e[e'/v] \parallel qs, qs'', qs'[e'/v]\,]\!]^{\mathsf{T}}\,.
\end{array}
$$

Following the course of the proof for rule $\mathscr{M}$-Norm-3, establishing rule $\mathscr{M}$-Norm-1 then is immediate by appealing to the functoriality of T' and law Monad-4. A similar remark applies to rule $\mathscr{M}$-Norm-2 and monad law Monad-2. For $\mathscr{M}$-Norm-4 note that if $\dot{+}$ is left-idempotent we have (for arbitrary x)

$$
\begin{array}{ll}
& \mathit{if}\ (p \vee q)\ \mathit{then}\ \mathit{unit}^{\mathsf{T}}\ x\ \mathit{else}\ \mathit{zero}^{\mathsf{T}} \\
= & \quad \{\, \dot{+} \text{ idempotent} \,\} \\
& (\mathit{if}\ p\ \mathit{then}\ \mathit{unit}^{\mathsf{T}}\ x\ \mathit{else}\ \mathit{zero}^{\mathsf{T}}) \dot{+} (\mathit{if}\ q\ \mathit{then}\ \mathit{unit}^{\mathsf{T}}\ x\ \mathit{else}\ \mathit{zero}^{\mathsf{T}})\,.
\end{array}
$$

To prove $\mathscr{M}$-Norm-5 simply observe that in the Id-monad we have $\mathit{join}^{\mathsf{Id}} = \mathit{id}$ and thus

$$
\begin{array}{ll}
& \mathscr{M}\,[\![\,[\![\,e \parallel qs'\,]\!]^{\mathsf{Id}} \parallel qs\,]\!]^{\mathsf{Id}} \\
= & \quad \{\, \mathit{join}^{\mathsf{Id}} = \mathit{id} \,\} \\
& \mathit{join}^{\mathsf{Id}}\,(\mathscr{M}\,[\![\,[\![\,e \parallel qs'\,]\!]^{\mathsf{Id}} \parallel qs\,]\!]^{\mathsf{Id}}) \\
= & \quad \{\, \mathscr{M}\text{-5} \,\} \\
& \mathscr{M}\,[\![\,e \parallel qs, qs'\,]\!]^{\mathsf{Id}}\,.
\end{array}
$$

□

93 Lemma. The application of a type functor $\mathsf{T} \neq \mathsf{Id}$ may be fused with a comprehension evaluated in the associated T-monad (no variable bound in qs may appear free in f):

$$
\mathsf{T}f\,(\mathscr{M}\,[\![\,e \parallel qs\,]\!]^{\mathsf{T}}) \quad = \quad \mathscr{M}\,[\![\,f\,e \parallel qs\,]\!]^{\mathsf{T}}\,. \qquad \text{Functor-}\mathscr{M}\text{-Fusion}
$$

Proof. Induction over qs. The equation holds for $qs = \varepsilon$ due to $\mathscr{M}$-1 and the functoriality of T. For non-empty qs, we may safely assume that qs starts with a generator, *i. e.*, $qs = v \leftarrow e', qs'$. Should qs start with a filter p we simply move p to the end of qs. (It is straightforward to see that filters may

take an arbitrary place in the qualifier list as long as they are not moved out of the scope of the free variables they refer to. In our case, p is constant with respect to the variables bound in qs.)

Quite like in the validation of the normalization rules, we will first establish FUNCTOR-$\mathscr{M}$-FUSION for $qs' = \varepsilon$ and later generalize our findings to arbitrary qs'.

Induction base case. Set $qs' = \varepsilon$. In general, $e : \mathsf{T}'E$ with $\mathsf{T} \neq \mathsf{T}'$, E arbitrary:

$$
\begin{aligned}
& \mathscr{M}\,[\![\, f\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& (\!|\tau|\!)\,(\mathscr{M}\,[\![\, f\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \mathscr{M}\text{-2} \,\} \\
& ((\!|\tau|\!) \cdot \mathsf{T}'(\lambda v.f\, e))\,(\mathscr{M}\, e') \\
= \quad & \{\, \text{CATA-MAP-FUSION} \,\} \\
& \mathsf{T}(\lambda v.f\, e)\,(\mathscr{M}\, e') \\
= \quad & \{\, v \text{ not free in } f, \mathsf{T} \text{ functor} \,\} \\
& (\mathsf{T} f \cdot \mathsf{T}(\lambda v.e))\,(\mathscr{M}\, e') \\
= \quad & \{\, \mathscr{M}\text{-2} \,\} \\
& \mathsf{T} f\,(\mathscr{M}\,[\![\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}})\,.
\end{aligned}
$$

Assuming that the lemma is true for qs', perform the induction step. This will complete the proof.

$$
\begin{aligned}
& \mathscr{M}\,[\![\, f\, e \mid v \leftarrow e', qs' \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& \mathit{join}^{\mathsf{T}}\,(\mathscr{M}\,[\![\, [\![\, f\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow e' \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \text{induction hypothesis} \,\} \\
& \mathit{join}^{\mathsf{T}}\,(\mathscr{M}\,[\![\, \mathsf{T} f\,[\![\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow e' \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \text{induction base case} \,\} \\
& (\mathit{join}^{\mathsf{T}} \cdot \mathsf{T}\mathsf{T} f)\,(\mathscr{M}\,[\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow e' \,]\!]^{\mathsf{T}}) \\
= \quad & \{\, \mathit{join}^{\mathsf{T}} \text{ is natural transformation } \mathsf{T}\mathsf{T} \dot{\to} \mathsf{T} \,\} \\
& \mathsf{T} f\,(\mathit{join}^{\mathsf{T}}\,(\mathscr{M}\,[\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{T}} \mid v \leftarrow e' \,]\!]^{\mathsf{T}})) \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& \mathsf{T} f\,(\mathscr{M}\,[\![\, e \mid v \leftarrow e', qs' \,]\!]^{\mathsf{T}})\,.
\end{aligned}
$$

See Lemma 114 for a related transformation applicable to Id-monad comprehensions. □

94 Rules $\mathscr{M}$-Norm-1 through $\mathscr{M}$-Norm-5 form a confluent and terminating set of rewriting rules which is the main incentive to refer to them as *normalization* rules. Given a monad comprehension, repeated application of the rules derives an equivalent syntactic *normal form*

$$[\![\, e \mid v_1 \leftarrow e_1,\ v_2 \leftarrow e_2,\ \ldots,\ v_n \leftarrow e_n,\ p \,]\!]^{\mathsf{T}}$$

for the original comprehension. This is achieved by rules $\mathscr{M}$-Norm-3 and $\mathscr{M}$-Norm-4. Normalization gives an unnesting procedure that is *complete* in the sense that an exhaustive application of the rules leads to a comprehension in which all semantically sound unnestings have been performed [40, 41]. If a query is evaluated in idempotent monads only (*cf.* the proviso of $\mathscr{M}$-Norm-4), then this can go as far as that all e_i are atomic expressions with respect to the monad comprehension syntax given in Definition 67, *i. e.*, database entry points or constants. Nested terms may still occur in the filter p and the comprehension's head e. We will approach these types of nesting when we have introduced *algebraic combinators* in the upcoming Chapter 4.

The comprehension normal form has been used by Fegaras and Maier as a canonical starting point for query optimization [40, 41]. Scanning the comprehension qualifier list from left to right, the algorithm described in [41] introduces *outerjoins* and algebraic grouping operators to cope with nested comprehensions that could not be flattened by normalization.

Unnesting disentangles queries and makes operands of formerly inner queries accessible in the outer enclosing comprehension. This, in turn, provides new possibilities for further rewritings and optimizations that will prove especially useful in the course of Chapters 4 and 5. Unnesting principally facilitates the applicability of other optimizing transformations that may help to, *e. g.*, trade a nested loop for a more efficient join.

Comprehension syntax provides a rather poor variety of syntactical forms, but in the early query translation phase this is more of a virtue than a shortcoming. Monad comprehensions extract and emphasize the *structural* information contained in a query rather than to stress diversity of query clauses. It is this uniformity that facilitates deep query analysis like the completeness result for query unnesting via comprehension normalization which we have mentioned in Paragraph 94. This can lead to new insights and simplifications which is the point the three concluding paragraphs try to make.

95 In [106], Steenhagen, Apers, and Blanken analyzed a class of SQL-like queries which exhibit correlated nesting in the `where`-clause, more specifically

$$\begin{array}{l}\texttt{select distinct } f\,x \\ \quad\texttt{from } xs\ \texttt{as } x \\ \quad\texttt{where } p\,x\,z \end{array} \qquad \text{with } z = \begin{pmatrix}\texttt{select } g\,x\,y \\ \quad\texttt{from } ys\ \texttt{as } y \\ \quad\texttt{where } q\,x\,y\end{pmatrix} .$$

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

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

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* (see Paragraph 108 and [48, 69, 105, 106, 107]). Whether we can derive a flat join query is, naturally, dependent on the nature of the yet unspecified predicate p.

Steenhagen et al. state the following theorem—reproduced here using monadic terminology—which provides a partial answer to the question:

> *Whenever $p\,x\,z$ can be rewritten into $[\![\, p'\,x\,v \mid v \leftarrow z \,]\!]^{\mathsf{exists}}$ (for some p') the original query may be evaluated by a flat join.*

For space reasons a proof is omitted in [106]. After the original query is compiled via $\mathscr{Q}$, however, normalization can provide a succinct proof in a few lines (collection monad T is arbitrary):

$$\begin{array}{ll} & [\![\, f\,x \mid x \leftarrow xs, p\,x\,z \,]\!]^{\mathsf{set}} \\ = & \quad \{\,\text{unfold } p\,\} \\ & [\![\, f\,x \mid x \leftarrow xs, [\![\, p'\,x\,v \mid v \leftarrow z \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\ = & \quad \{\, \mathscr{M}\text{-}\textsc{Norm}\text{-}4 \,\} \\ & [\![\, f\,x \mid x \leftarrow xs, v \leftarrow z, p'\,x\,v \,]\!]^{\mathsf{set}} \\ = & \quad \{\,\text{unfold } z\,\} \\ & [\![\, f\,x \mid x \leftarrow xs, v \leftarrow [\![\, g\,x\,y \mid y \leftarrow ys, q\,x\,y \,]\!]^{\mathsf{T}}, p'\,x\,v \,]\!]^{\mathsf{set}} \end{array}$$

$$= \quad \{\ \mathscr{M}\text{-Norm-3}\ \}$$
$$[\![\, f\, x \mid x \leftarrow xs,\ y \leftarrow ys,\ q\, x\, y,\ p'\, x\, (g\, x\, y) \,]\!]^{\mathsf{set}}\ .$$

A flat join between xs and ys can evaluate the resulting comprehension (this will become more apparent in Chapter 4).

But we can say even more and strengthen the statement of the theorem (thus answering an open question that has been put by Steenhagen et al. in [106]):

> *If p is* not *rewriteable into an existential quantifier like above, then we can conclude—based on the completeness result for comprehension normalization—that unnesting will in fact be impossible whatsoever.*

Note. In [106], the *antijoin* (*cf.* Chapter 4) is also considered a flat join operator and we will do so in the upcoming chapters, too. This turns $[\![\, \neg p'\, x\, v \mid v \leftarrow z \,]\!]^{\mathsf{all}}$ into an additionally acceptable form for predicate p.

96 Kim's seminal work on the unnesting of SQL queries [69] may largely be understood in terms of normalization if queries are interpreted in the monad comprehension calculus. At the same time we can generalize and avoid the treatment of special cases. We additionally gain insight into questions on the validity of these unnesting strategies in the context of a complex object model featuring diverse collection type formers beyond `set`.

In [69], a classification of query nesting was proposed which associates each type of nesting with an equivalence-preserving unnesting transformation (modulo the infamous `count` bug whose treatment requires the introduction of the relational outerjoins or nestjoins [48, 107]) whose primary goal is to rewrite the original nested expression into a flat join query. Monad comprehension normalization readily unnests queries of Kim's *type J*, *i. e.*, SQL queries of the form

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

Note that predicate q refers to the query variable x so that outer and nested query are correlated; uncorrelated (*type N*) queries need no special treatment and are transformed likewise. We will turn to a considerable generalization

of *type D* queries—containing universal quantifiers and thus essentially performing relational division—in Paragraph 116 when the associated notions are in context.

Normalization deduces an equivalent join query for a *type J* expression in two steps:

$$
\begin{aligned}
& [\![\, f\, x \mid x \leftarrow xs, [\![\, p\, x = v \mid v \leftarrow [\![\, g\, y \mid y \leftarrow ys, q\, x\, y \,]\!]^{\mathsf{T}} \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-}\textsc{Norm}\text{-}3 \,\} \\
& [\![\, f\, x \mid x \leftarrow xs, [\![\, p\, x = g\, y \mid y \leftarrow ys, q\, x\, y \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-}\textsc{Norm}\text{-}4 \,\} \\
& [\![\, f\, x \mid x \leftarrow xs, y \leftarrow ys,\ q\, x\, y,\ p\, x = g\, y \,]\!]^{\mathsf{set}} \,.
\end{aligned}
$$

The resulting comprehension is Kim's *canonical 2-relation query* (*i. e.*, the join)

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

97 Above, we have chosen the collection monads as general possible: this form of *type J full unnesting* is sound only if the outer query is evaluated in the set monad; no such restriction is necessary for the inner query.

This restriction on the type formers becomes void, however, if we already stop the unnesting process after the application of rule $\mathscr{M}$-Norm-3. As we will see in the next chapter, this comes with the extra benefit that the query becomes subject to evaluation by an efficient *semijoin* ($\ltimes$) while the fully unnested query calls for a general θ-join. Full unnesting may blur the detection of beneficial query execution alternatives.

On these grounds, while we believe in monad comprehension normalization as an important *preprocessing step* that should precede other query optimization phases, we do not subscribe to full comprehension unnesting as it was argued for in [38, 40, 41].

Chapter 4

Combinators

98 Given a simple spine transformer-based query engine as well as the mapping $\mathscr{M} \cdot \mathscr{Q}$ we are already able to rapidly prototype a fully functional complex value database system with OQL support. Based on the work reported in the two previous chapters, a spine transformer database engine has actually been manufactured from scratch and successfully operated in the realm of the *CROQUE* object database project [52, 58]. For efficiency reasons we have discussed in the introduction to Chapter 3, this is, however, clearly *not* the route we want to take. We will nevertheless briefly take up the idea in Chapter 6.

On the other hand, from the viewpoint of the catamorphic query representation, we know that *any* action executed in the query engine may eventually be broken down into walks of a spine. It is only that a classical database query engine implements algorithms—*physical operators* [66]—which allow for a significantly more efficient, possibly parallel, execution of a potentially large series of spine transformations. Typically this efficiency is accomplished by means of more or less involved support data structures, *e. g.*, *indices*. Although these physical operators construct spines they generally do so in completely different ways (random access to spine nodes, block-wise processing, and so on) than in adherence to the primitive catamorphic recursion pattern.

The principle observation that motivates the present chapter is that it is nevertheless possible to *encode the action* of a physical operator using our abstract query representation, *i. e.*, via monad comprehensions and thus catamorphisms. This encoding emphasizes the algebraic properties of the operator rather than its—potentially many—underlying physical realizations. 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).

We refer to these encodings as *combinators* for reasons which will become clear soon. Unlike in the monad comprehension calculus, expressing a query

in terms of combinators provides the query engine with clear hints which algorithms, besides nested loops, may be enlisted to execute the query. The set of combinators encoding the physical algorithms of a query engine forms the *logical algebra* associated with this engine [55].

99 Combinators. We have just said that a *combinator* encodes a series of actions on a spine which, when viewed from the outside, *act like* a possibly complex catamorphic spine transformer. For our aims, this is an operator's prominent property. Its actual physical realization is going on behind the scenes. This opacity is reflected by the means we use to model operators, namely as *closed expressions*, *i. e.*, *combinators*, in the sense of the λ-calculus [92, 95]. As an example, the relational *selection* operator σ is represented by the closed expression

$$\begin{array}{rcl} \sigma & : & (E \to Bool) \to \mathsf{set}E \to \mathsf{set}E \\ \sigma & = & \underbrace{\lambda p.\lambda xs.[\![\, x \mid x \leftarrow xs, p\, x \,]\!]^{\mathsf{set}}}_{\text{closed}} \; . \end{array}$$

As the set of free variables of a combinator expression is empty by definition we do not encounter inter-combinator variable dependencies in a composition of combinators. This facilitates the reordering and transformation of pure combinator expressions, a property that has long been exploited in algebraic query rewriting frameworks. (Although surprisingly many proposed "query algebras" heavily depend on the presence of free variables and thus do not qualify as algebras in the strict sense of the word.)

To reach specific optimization goals, notably the generation of stream-based programs for queries (query execution plans that do not allocate memory to buffer intermediate results), it pays off to break the encapsulation the combinator notion provides. We will not do so until Chapter 6, however.

100 Combinators from monad comprehensions. The path is clear: given a query engine's logical algebra we encode the actions of its operators as closed monad comprehension expressions or, in rare cases, catamorphisms. The PTIME expressive power of the catamorphic query model makes this possible for query engines implementing classical relational algebra as well as a broad range of advanced query algebras, from various proposals for the non-first normal form (NF^2) relational model [1, 12, 35, 99, 100, 105] to algebras for complex value databases [6, 24, 30, 32, 76, 115]. Here, we will focus on a variant of the latter.

The equational theory of monads and the initial algebras the monads were derived from will provide the equational theory for the combinator algebra.

Since a combinator is, in principle, nothing else than a monad comprehension *macro* (built adhering to the closed expression property) its interaction with an enclosing comprehension expression is well-defined.

This paves the way for a step-wise or incremental discipline for the derivation of a pure combinator expression from a given monad comprehension: the combinator definitions serve as *patterns* which may be used to trigger a replacement trading the matched comprehension subexpression for an equivalent combinator. Along the way we are left with *hybrid* query expressions of which parts are either built using comprehension syntax or combinators. If the underlying query engine implements the catamorphism combinator $(\!|\cdot|\!)$ for the initial algebras its data model rests upon—this is actually a light requirement—this step-wise replacement process is guaranteed to generate a pure combinator expression. Falling back onto catamorphisms, however, should clearly be the last resort.

Each replacement step enriches the original query expression by hints on which strategies for query execution may be profitable. This does not yet select the physical operators themselves, but (a) the introduction of a combinator seeds the search for the actual execution plan, and (b) a combinator comes with a limited number of physical implementation alternatives. In this sense we decrease the level of abstraction by placing combinators inside a formerly comprehension-based query.

101 An exploration of a mixture of the calculus and algebra styles of query rewriting over monadic types forms the core of this chapter. Query transformations may be formulated using the style in which their applicability is most easily detected and expressed.

This procedure has much in common with the work of Nakano as well as Steenhagen, Apers, Blanken, and de By in the relational and NF^2 domains, respectively [88, 105, 106, 107]. There, logical algebra semantics were specified using specific instances of the relational tuple calculus. As already described in the previous paragraph, this opens the scene for a hybrid query rewriting framework. Steenhagen et al. make significant use of relational calculus to prove correct and justify rewritings, especially when it comes to the processing of complex predicates containing quantifiers. The actual query transformation rules (and thus the actual query rewriting), however, are formulated at the algebra-level.

Here, we employ the monad comprehension calculus also as a tool to calculate on the "semantics level", *i. e.*, the core query transformation work is performed on the calculus-level. Once again, comprehension normalization greatly simplifies our task. (The explicit treatment of *conjunctive path*

formulas in [105, p. 67 ff.] is, for example, an algebra-level formulation of unnesting rule $\mathscr{M}$-Norm-4).

It is our aim *not* to restrict our attention to queries that originate from the use of a specific subset of the user-level query language (OQL, say). The `select-from-where` block is the principle query construct, but we believe that studying its efficient translation in isolation from the rest of the query language comes with the danger of missing profitable plans. The uniformity of monad comprehension-based query representation has something to offer here. (Chapter 5 presents extended examples of the treatment of queries other than of the `select-from-where` type.)

4.1 Defining and Detecting Combinators

102 The following definition sets up a combinator algebra for a query engine that can operate over monadic types. It is our assumption that the query engine copes with set, bag, as well as list values and supports similar facilities for the three collection formers. At the combinator level this is reflected by polymorphic typing: the combinators are parametric in their monadic input and result types which implies that they behave similarly for the different type instantiations in which they are going to be used. At the same time this prevents the introduction of a large number of families of combinators in which the members of a family implement a common operator but each member for a specific type only.

103 Definition. Let T, T' denote monadic types and let E, E', E'' be placeholders for valid query types (*cf.* Paragraph 78). The list below defines the set of combinators that we will refer to in the sequel. Because we will encounter *curried* (*i. e.*, partial) applications of the combinators, this definition displays them in prefix notation instead of the "classical" infix form. (Fully applied binary operators are also written between their collection-typed arguments, *i. e.*, we consider

$$\bowtie\, p\, f\, xs\, ys \qquad \text{and} \qquad xs \underset{p\,|\,f}{\bowtie} ys$$

to be equivalent notations.) It goes without saying that a combinator application is sensible only if—given the monadic types of its arguments—its defining monad comprehension is well-defined for these type instantiations.

$$\pi \qquad : \quad (E \to E') \to \mathsf{T}E \to \mathsf{T}E'$$

$$
\begin{array}{lcll}
\pi\ f\ xs & = & \mathsf{T} f\ xs \quad = \quad [\![\, f\ x \mid x \leftarrow xs \,]\!]^{\mathsf{T}} & \textsc{Project} \\
\\
\alpha^{\mathsf{T}} & : & (E \to E') \to \mathsf{T}'E \to \mathsf{T}E' & \\
\alpha^{\mathsf{T}}\ f\ xs & = & [\![\, f\ x \mid x \leftarrow xs \,]\!]^{\mathsf{T}} & \textsc{Aggregate} \\
\\
\sigma & : & (E \to \mathit{Bool}) \to \mathsf{T}E \to \mathsf{T}E & \\
\sigma\ p\ xs & = & [\![\, x \mid x \leftarrow xs, p\ x \,]\!]^{\mathsf{T}} & \textsc{Select} \\
\\
\times & : & \mathsf{T}E \to \mathsf{T}'E \to \mathsf{T}(E \times E') & \\
\times\ xs\ ys & = & [\![\, (x, y) \mid x \leftarrow xs, y \leftarrow ys \,]\!]^{\mathsf{T}} & \textsc{Cross} \\
\\
\bowtie & : & (E \to E' \to \mathit{Bool}) \to (E \to E' \to E'') & \\
 & & \quad \to \mathsf{T}E \to \mathsf{T}'E' \to \mathsf{T}E'' & \\
\bowtie\ p\ f\ xs\ ys & = & [\![\, f\ x\ y \mid x \leftarrow xs, y \leftarrow ys, p\ x\ y \,]\!]^{\mathsf{T}} & \textsc{Join} \\
\\
\triangle & : & (E \to E' \to \mathit{Bool}) \to (E \to \mathsf{T}'E' \to E'') & \\
 & & \quad \to \mathsf{T}E \to \mathsf{T}'E' \to \mathsf{T}E'' & \\
\triangle\ p\ f\ xs\ ys & = & [\![\, f\ x\ [\![\, y \mid y \leftarrow ys, p\ x\ y \,]\!]^{\mathsf{T}'} \mid x \leftarrow xs \,]\!]^{\mathsf{T}} & \textsc{Nestjoin} \\
\\
\ltimes & : & (E \to E' \to \mathit{Bool}) \to \mathsf{T}E \to \mathsf{T}'E' \to \mathsf{T}E & \\
\ltimes\ p\ xs\ ys & = & [\![\, x \mid x \leftarrow xs, [\![\, p\ x\ y \mid y \leftarrow ys \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{T}} & \textsc{Semijoin} \\
\\
\not\ltimes & : & (E \to E' \to \mathit{Bool}) \to \mathsf{T}E \to \mathsf{T}'E' \to \mathsf{T}E & \\
\not\ltimes\ p\ xs\ ys & = & [\![\, x \mid x \leftarrow xs, [\![\, \neg p\ x\ y \mid y \leftarrow ys \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{T}} & \textsc{Antijoin} \\
\\
\mu & : & \mathsf{T}\mathsf{T}'E \to \mathsf{T}E & \\
\mu\ xss & = & [\![\, x \mid xs \leftarrow xss, x \leftarrow xs \,]\!]^{\mathsf{T}} & \textsc{Unnest} \\
\\
\varsigma & : & (E \times E \to \mathit{Bool}) \to \mathsf{T}E \to \mathsf{list}E & \\
\varsigma\ \circledcirc\ xs & = & (\!|\, \mathit{nil}^{\mathsf{list}} \triangledown \mathit{ins}^{\mathsf{list}} \circledcirc |\!)\ xs & \textsc{Sort}
\end{array}
$$

//

104 **Notes.** (a) Issues of polymorphism aside, the combinators are standard and should explain themselves. The nestjoin $\triangle$ combines joining with grouping: for each x in xs a group from those y in ys is built that make predicate $p\,x\,y$ evaluate to *true*. Note that nestjoin provides an abstraction for nesting in a comprehension's *head*, a case the normalization rules of Definition 91 do not account for. Nestjoin thus plays a central role in nested

query processing. Variants of nestjoin have appeared as, *e. g.*, the *binary grouping* operator of [35] or the *hierarchical join* of [96].

(b) SEMIJOIN and ANTIJOIN define the *left* variants of the semijoin and antijoin operators, respectively. The right variants $\rtimes$ and $\not\rtimes$ are dually defined. Semijoin $\ltimes\ p\,xs\,ys$ emits all x in xs for which some (arbitrary) y in ys with $p\,x\,y$ can be found. The corresponding antijoin computes the complement.

(c) In a setting of monadic types, combinator α^{T} has two facets: evaluated in an identity monad, *i. e.*, a monad that has been derived from the algebras of Definition 64, α^{T} acts like a quantifier or aggregation operator; evaluated in a collection monad the combinator coerces between collection types (*e. g.*, duplicates are eliminated for $\mathsf{T} = \mathsf{set}$).

(d) Remember that it has not been our goal to define a *minimal* algebra in the sense that each combinator adds expressive power (indeed, $(\!|\cdot|\!)$ is all what would be needed then). Highly specialized combinators like nestjoin rather flag the presence of an efficient execution algorithm inside the query engine and are thus introduced for optimization purposes only.

105 Partial matches. It is actually only a small step from a combinator definition to its associated detection rule: reading a definition from right to left specifies a rewriting rule that spots the opportunity to *completely* replace the right-hand side by a combinator. Note that such a rule application is valid only if the lhs combinator arguments are closed expressions (this is a requirement inherent to the combinator notion). Rarely, however, will such perfect matches occur.

The detection process is significantly enhanced if we provide (rule-based) knowledge about *partial* matches and replacements. Rules of this type obviously lead to hybrid query expressions as we have discussed them in Paragraph 100. As the thus induced rules exhibit a similar structure let us list only a selection of them below.

Most of the rules rely on the simple syntactic recipe after which monad comprehensions are constructed: as before, let qs, qs', qs'' represent the syntactic category of possibly empty lists of qualifiers (*cf.* Definition 67). The left-hand sides thus specify (qualifier) list patterns that match the specified template of generators and filters. We will assess the rules' correctness in the next paragraph.

Note. The uses of $p\,x$ or $p\,x\,y$ below denote predicates p in which no variables besides x (or x and y, respectively) may occur free. This includes closed predicates. The replacements in the rhs of rules $\mathscr{M}$-JOIN and $\mathscr{M}$-NESTJOIN implement the deconstruction of the tuple-typed elements found in the join

results. The introduction of pattern-matching capabilities for comprehension generators—which we have not done for simplicity reasons, but see [92, 125]—make such replacements obsolete.

$$[\![\, f\, e \mid qs \,]\!]^{\mathsf{T}} \quad \to \quad \pi\ f\ [\![\, e \mid qs \,]\!]^{\mathsf{T}} \qquad \mathscr{M}\text{-}\textsc{Project}$$
$$(f \text{ closed})$$

$$[\![\, e \mid qs : \mathsf{T}' \,]\!]^{\mathsf{T}} \quad \to \quad \alpha^{\mathsf{T}}\ id\ [\![\, e \mid qs \,]\!]^{\mathsf{T}'} \qquad \mathscr{M}\text{-}\textsc{Aggregate}$$

$$[\![\, e \mid qs,\ x \leftarrow xs,\ qs',\ p\, x,\ qs'' \,]\!]^{\mathsf{T}} \quad \to \quad [\![\, e \mid qs,\ x \leftarrow \sigma\ p\, xs,\ qs',\ qs'' \,]\!]^{\mathsf{T}} \qquad \mathscr{M}\text{-}\textsc{Select}$$

$$[\![\, e \mid qs, x \leftarrow xs, y \leftarrow ys,\ qs',\ p\, x\, y,\ qs'' \,]\!]^{\mathsf{T}} \quad \to \quad [\![\, e[outl\, v/x][outr\, v/y] \mid qs,\ v \leftarrow xs \underset{p \mid \lambda x.\lambda y.(x,y)}{\bowtie} ys,\ qs'[outl\, v/x][outr\, v/y],\ qs''[outl\, v/x][outr\, v/y] \,]\!]^{\mathsf{T}}$$
$$\mathscr{M}\text{-}\textsc{Join}$$

$$[\![\, e \mid qs,\ x \leftarrow xs,\ qs',\ p\, x\, e',\ qs'' \,]\!]^{\mathsf{T}} \quad \to \quad [\![\, e[outl\, v/x] \mid qs,\ v \leftarrow xs \underset{q \mid \lambda x.\lambda y.(x, g\, x\, y)}{\triangle} ys,\ qs'[outl\, v/x],\ p[outl\, v/x][outr\, v/e'],\ qs''[outl\, v/x] \,]\!]^{\mathsf{T}}$$
$$\text{with } e' = [\![\, g\, x\, y \mid y \leftarrow ys, q\, x\, y \,]\!]^{\mathsf{T}'}$$
$$\mathscr{M}\text{-}\textsc{Nestjoin}$$

$$[\![\, e \mid qs,\ x \leftarrow xs,\ qs',\ [\![\, p\, x\, y \mid y \leftarrow ys \,]\!]^{\mathsf{exists}},\ qs'' \,]\!]^{\mathsf{T}} \quad \to \quad [\![\, e \mid qs,\ x \leftarrow xs \underset{p}{\ltimes} ys,\ qs', qs'' \,]\!]^{\mathsf{T}} \qquad \mathscr{M}\text{-}\textsc{Semijoin}$$

106 Lemma. The introduction of combinators via the rules listed in Paragraph 105 does not alter the meaning of the affected monad comprehensions.

Proof. The proof scheme is so simple that we establish the correctness of rules $\mathscr{M}$-Project and $\mathscr{M}$-Join only here: (a) in the rhs, unfold the combinator definition found in Definition 103, then (b) use monad comprehension normalization to derive the lhs.

$\mathscr{M}$-Project:

$$\pi\ f\ [\![\, e \mid qs \,]\!]^{\mathsf{T}}$$

$$
\begin{aligned}
&= \quad \{\,\text{unfold } \pi \text{ via } \textsc{Project}\,\} \\
&\quad [\![\, f\, x \mid\!]\, x \leftarrow [\![\, e \mid\!]\, qs \,]\!]^{\mathsf{T}} \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\mathscr{M}\text{-}\textsc{Norm}\text{-}3\,\} \\
&\quad [\![\, (f\, x)[e/x] \mid\!]\, qs \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\text{replacement}\,\} \\
&\quad [\![\, f\, e \mid\!]\, qs \,]\!]^{\mathsf{T}}\,.
\end{aligned}
$$

$\mathscr{M}$-JOIN: we assume $xs : \mathsf{T}'E$ for some E:

$$
\begin{aligned}
&\quad [\![\, e[outl\, v/x][outr\, v/y] \mid\!]\, qs,\ v \leftarrow xs \underset{p|\lambda x.\lambda y.(x,y)}{\bowtie} ys, \\
&\qquad\qquad\qquad\qquad qs'[outl\, v/x][outr\, v/y], \\
&\qquad\qquad\qquad\qquad qs''[outl\, v/x][outr\, v/y] \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\text{unfold } \bowtie \text{ via } \textsc{Join}, \beta\text{-reduction}\,\} \\
&\quad [\![\, e[outl\, v/x][outr\, v/y] \mid\!]\, qs,\ v \leftarrow [\![\, (x,y) \mid\!]\, x \leftarrow xs,\ y \leftarrow ys,\ p\, x\, y \,]\!]^{\mathsf{T}'}, \\
&\qquad\qquad\qquad\qquad qs'[outl\, v/x][outr\, v/y], \\
&\qquad\qquad\qquad\qquad qs''[outl\, v/x][outr\, v/y] \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\mathscr{M}\text{-}\textsc{Norm}\text{-}3\,\} \\
&\quad [\![\, e[outl\, v/x][outr\, v/y][(x,y)/v] \mid\!]\, qs,\ x \leftarrow xs,\ y \leftarrow ys,\ p\, x\, y, \\
&\qquad\qquad\qquad\qquad\qquad\qquad qs'[outl\, v/x][outr\, v/y][(x,y)/v], \\
&\qquad\qquad\qquad\qquad\qquad\qquad qs''[outl\, v/x][outr\, v/y][(x,y)/v] \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\text{replacements, product}\,\} \\
&\quad [\![\, e \mid\!]\, qs,\ x \leftarrow xs,\ y \leftarrow ys,\ p\, x\, y,\ qs',\ qs'' \,]\!]^{\mathsf{T}} \\
&= \quad \{\,\text{shift filter in qualifier list}\,\} \\
&\quad [\![\, e \mid\!]\, qs,\ x \leftarrow xs,\ y \leftarrow ys,\ qs',\ p\, x\, y,\ qs'' \,]\!]^{\mathsf{T}}\,.
\end{aligned}
$$

□

It comes as no surprise that unnesting the rhs of the combinator introduction rules derives their lhs: it is the ultimate point of combinators to capture the structure of a comprehension that has not been fully unnested or, equivalently, rewritten into its nested loops form (*cf.* Paragraph 97).

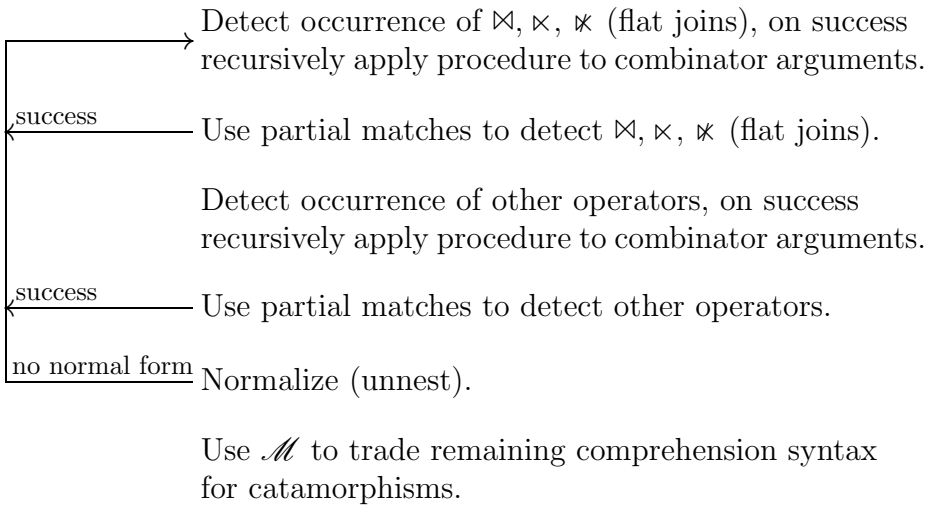
4.2 Combinators From Monad Comprehensions

107 Where is our current position in the greater picture? Prior to the introduction of combinators in this chapter we were dealing with a completely deterministic query translation process, namely $\mathscr{M} \cdot \mathscr{Q}$ followed by

full unnesting. This has changed with our goal to derive combinator queries from monad comprehensions: the set of rewriting rules built from the normalization rules, combinator definitions, and partial combinator patterns (of Paragraph 105) is clearly non-confluent. From the monad comprehension, *e. g.*, $[\![\, f\, x \,]\!]\ x \leftarrow xs, y \leftarrow ys, p\, y\, [\![\, \mathsf{T} \,]\!]$, the rules open a spectrum from purely catamorphic plans (*i. e.*, nested loops), over combinator queries using $\times$ or $\bowtie$ to the most efficient $\ltimes$-based alternative.

- A query rewriter that gives preference to the application of rules stemming from the combinator definitions will perform a *top-down* derivation—with respect to the syntactic structure of comprehensions—of combinator queries. *Benefit:* early detection of, *e. g.*, a semijoin occurrence (or of a flat join in general), can significantly improve the overall quality of the emitted combinator expression.
- The lhs of partial match rules inspect the innards of a comprehension and, once applied, replace parts of the comprehension (qualifier list), thus working their way *bottom-up*. *Benefit:* the lhs are much more likely to match a given query as they require partial matches only and may rewrite a comprehension such that a combinator definition can subsequently be detected.

This suggests the following sketch of a procedure which can guide a query optimizer in its task to assign priorities to rule groups and to organize its rewriting strategy. Probing the rules of the topmost alternative first, the optimizer only as a last resort falls through to the generation of nested loops via the monad comprehension desugaring scheme $\mathscr{M}$ (Definition 69).



108 Let us pick up the query class of Paragraph 95:

$$Q \equiv \begin{array}{l}\texttt{select}\ f\,x \\ \quad\texttt{from}\ xs\ \texttt{as}\ x \\ \texttt{where}\ p\,x\,z\end{array} \qquad \text{with } z = \begin{pmatrix}\texttt{select}\ g\,x\,y \\ \quad\texttt{from}\ ys\ \texttt{as}\ y \\ \texttt{where}\ q\,x\,y\end{pmatrix} .$$

We already know how to deal with this type of query should predicate p be essentially equivalent to $[\![\, p'\,x\,v \mid\!\mid v \leftarrow z \,]\!]^{\mathsf{exists}}$ for some p'. The theorem sketched in Paragraph 95 indicates the need for grouping in all other cases. To illustrate, let p denote a predicate of general form. A combinator query for Q should ideally employ the nestjoin $\triangle$ as this is the combinator which encodes the grouping functionality of the underlying query engine. The calculation goes on as follows (collection monads T, T' are arbitrary):

$$\begin{aligned}
& \mathscr{Q}\,Q \\
= \quad & \{\, \mathscr{Q},\ \text{unfold } z \,\} \\
& [\![\, f\,x \mid\!\mid x \leftarrow xs, p\,x\,[\![\, g\,x\,y \mid\!\mid y \leftarrow ys, q\,x\,y \,]\!]^{\mathsf{T}'} \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \mathscr{M}\text{-Nestjoin} \,\} \\
& [\![\, (f\,x)[outl\,v/x] \mid\!\mid v \leftarrow xs \underset{q|\lambda x.\lambda y.(x,y)}{\triangle} ys,\ (p\,x)[outl\,v/x] \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \text{replacements} \,\} \\
& [\![\, (f \cdot outl)\,v \mid\!\mid v \leftarrow xs \underset{q|\lambda x.\lambda y.(x,y)}{\triangle} ys,\ (p \cdot outl)\,v \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \mathscr{M}\text{-Select} \,\} \\
& [\![\, (f \cdot outl)\,v \mid\!\mid v \leftarrow \sigma\,(p \cdot outl)\,(xs \underset{q|\lambda x.\lambda y.(x,y)}{\triangle} ys) \,]\!]^{\mathsf{T}} \\
= \quad & \{\, \text{Project} \,\} \\
& \pi\,(f \cdot outl)\,(\sigma\,(p \cdot outl)\,(xs \underset{q|\lambda x.\lambda y.(x,y)}{\triangle} ys)) .
\end{aligned}$$

The resulting combinator query is a close variant of the plan that has been proposed in the original work on nestjoin-based query processing [106]. As we do not invoke some sort of *Eureka step*, it is reasonable to assume that the above trail of rewritings will actually be found by the optimizer.

Nestjoin encodes nesting a comprehension's head (see Paragraph 104) so

that its canonical use arises whenever queries of the form

$$Q' \quad \equiv \quad \begin{array}{l} \texttt{select}\ f\,x\,z \\ \quad \texttt{from}\ xs\ \texttt{as}\ x \\ \quad \texttt{where}\ p\,x \end{array} \qquad \text{with } z = \left(\begin{array}{l} \texttt{select}\ g\,x\,y \\ \quad \texttt{from}\ ys\ \texttt{as}\ y \\ \texttt{where}\ q\,x\,y \end{array} \right)$$

are to be processed. For $g\,x\,y = y$, the nestjoin occurrence in Q' is apparent as soon as $\mathscr{Q}$ has desugared the OQL syntax [107]:

$$\begin{array}{ll} & \mathscr{Q}\,Q' \\ = & \quad \{\, \mathscr{Q},\ \text{unfold}\ z \,\} \\ & [\![\, f\,x\,[\![\, y \,|\, y \leftarrow ys,\ q\,x\,y \,]\!]^{\mathsf{T}'} \,|\, x \leftarrow xs,\ p\,x \,]\!]^{\mathsf{T}} \\ = & \quad \{\, \mathscr{M}\text{-}\textsc{Project} \,\} \\ & [\![\, f\,x\,[\![\, y \,|\, y \leftarrow ys,\ q\,x\,y \,]\!]^{\mathsf{T}'} \,|\, x \leftarrow \sigma\ p\,xs \,]\!]^{\mathsf{T}} \\ = & \quad \{\, \textsc{Nestjoin} \,\} \\ & (\sigma\ p\,xs) \underset{q|\lambda x.\lambda y.(x,y)}{\triangle} ys\ . \end{array}$$

For arbitrary projections g, a *tupling* transformation hand in hand with $\textsc{Functor}$-$\mathscr{M}$-$\textsc{Fusion}$ pulls g out of the nested query block, thus paving the way for the use of nestjoin. A generalized nestjoin operator (accepting g as a parameter) that perfectly matches the above query class has been proposed in [106].

Query rewriting in this hybrid framework of combinators and monad comprehensions draws much of its attractiveness on the interleaving of combinator detection and calculus-level rewriting. The uniformity of the monad comprehension calculus—together with its syntactical simplicity—allows to strengthen the rewriting framework by adding only *few* but *generic* (*i. e.*, parametric in the monads) rules. This is the next point we try to make.

109 Example. Both nested existential quantifiers in the following OQL query are not closed (neither p nor q may be evaluated in the scope of one of the quantifiers only):

$$Q'' \quad \equiv \quad \begin{array}{l} \texttt{select distinct}\ x \\ \quad \texttt{from}\ xs\ \texttt{as}\ x \\ \texttt{where exists}\ y\ \texttt{in}\ ys\texttt{:} \\ \quad \texttt{(exists}\ z\ \texttt{in}\ zs\texttt{:}\ q\,x\,z\ \texttt{and}\ p\,y\,z\texttt{)} \end{array} \ .$$

Such variable interdependencies in the source query and its equivalent under mapping $\mathscr{Q}$ may render the introduction of combinators a hard problem. Assuming that the query rewriter is equipped with monad comprehension normalization rules and combinator patterns (we will redo this example shortly), we then calculate as follows:

$$
\begin{aligned}
& \mathscr{Q}\, Q'' \\
= \quad & \{\, \mathscr{Q} \,\} \\
& [\![\, x \mid x \leftarrow xs, [\![\, [\![\, q\, x\, z \wedge p\, y\, z \mid z \leftarrow zs \,]\!]^{\mathsf{exists}} \mid y \leftarrow ys \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-}\textsc{Norm}\text{-}4 \,\} \\
& [\![\, x \mid x \leftarrow xs, y \leftarrow ys, [\![\, q\, x\, z \wedge p\, y\, z \mid z \leftarrow zs \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{partial match for } \curlyvee \,\} \\
& [\![\, outl\, v \mid v \leftarrow xs \curlyvee ys, \\
& \qquad\qquad [\![\, q\, (outl\, v)\, z \wedge p\, (outr\, v)\, z \mid z \leftarrow zs \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-}\textsc{Semijoin} \,\} \\
& [\![\, outl\, v \mid v \leftarrow (xs \curlyvee ys) \underset{\lambda v.\lambda z.q\,(outl\,v)\,z \wedge p\,(outr\,v)\,z}{\ltimes} zs \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \textsc{Project} \,\} \\
& \pi\ outl \left((xs \curlyvee ys) \underset{\lambda v.\lambda z.q\,(outl\,v)\,z \wedge p\,(outr\,v)\,z}{\ltimes} zs \right)
\end{aligned}
$$

The introduction of the cross product performs a tupling of xs and ys so that predicates p and q may be evaluated given this stream of tuples and the bindings of z generated from zs. Only then we obtain a constellation a semijoin can implement. Tupling comes at a rather high cost, not least because of the necessary untupling realized by the π combinator. ◊

110 The rewriting process will clearly benefit from general means to unwind variable interdependencies. Transformations of this type are naturally expressed at the monad comprehension calculus level. The following three lemmata provide such tools. Having these at hand, Paragraph 115 will revisit the above example.

111 **Lemma.** Let T denote a commutative monad. Adjacent qualifiers qs', qs'' in a T-monad comprehension may be exchanged without affecting the comprehension's meaning under $\mathscr{M}$, given that qs' does not refer to variables bound in qs'' and vice versa [114]:

$$\mathscr{M}\, [\![\, e \mid qs,\ qs',\ qs'',\ qs''' \,]\!]^{\mathsf{T}} \quad = \quad \mathscr{M}\, [\![\, e \mid qs,\ qs'',\ qs',\ qs''' \,]\!]^{\mathsf{T}} \quad \textsc{Qual-Ex}$$

Note that repeated application of QUAL-EX gives a more general qualifier exchange theorem.

Proof. We can reuse the proof scheme we have applied to establish the monad comprehension normalization rules in Theorem 92: a proof for the case $qs = \varepsilon$ is found by desugaring the lhs of QUAL-EX and an appeal to the commutativity of $+\!\!+$ in $join^{\mathsf{T}} = (\!|\, [\,]^{\mathsf{T}} \triangledown +\!\!+ \,|\!)$ (*cf.* Lemma 60). Appealing to $\mathscr{M}$ and the base case then readily establishes the statement for arbitrary qs. □

A direct consequence of the above and comprehension unnesting rule $\mathscr{M}$-NORM-5 is the following.

112 Lemma. For Id-monads (monads derived from the algebras of Definition 64) we have that

$$\mathscr{M}\,[\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{Id}} \mid qs \,]\!]^{\mathsf{Id}} \quad = \quad \mathscr{M}\,[\![\, [\![\, e \mid qs \,]\!]^{\mathsf{Id}} \mid qs' \,]\!]^{\mathsf{Id}} \qquad \text{NEST-EX}$$

provided that qs and qs' do not include mutual variable references.

Proof.

$$\begin{aligned}
& \mathscr{M}\,[\![\, [\![\, e \mid qs' \,]\!]^{\mathsf{Id}} \mid qs \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \mathscr{M}\text{-NORM-5} \,\} \\
& \mathscr{M}\,[\![\, e \mid qs,\ qs' \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \text{QUAL-EX} \,\} \\
& \mathscr{M}\,[\![\, e \mid qs',\ qs \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \mathscr{M}\text{-NORM-5} \,\} \\
& \mathscr{M}\,[\![\, [\![\, e \mid qs \,]\!]^{\mathsf{Id}} \mid qs' \,]\!]^{\mathsf{Id}} .
\end{aligned}$$

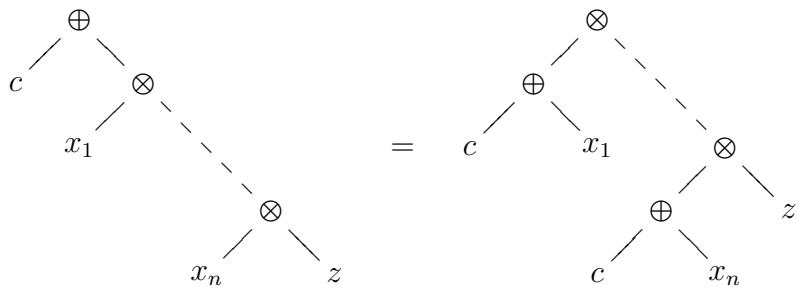
□

The last of the three "unwinding" tools provides a catamorphic generalization of the *descoping* transformation described in [106] and [15]. Based on the notion of *distributivity* defined below, descoping can pull expressions out of a comprehension's head.

113 Definition. Let $\alpha = z \triangledown \otimes : \mathsf{F}A \to A$ denote an F-algebra, with F being the polynomial endofunctor in **Set** describing algebras in insert representation. Function $\oplus : A \times A \to A$ **distributes** over α if for all $c : A$

$$\oplus \cdot (K_c \vartriangle (\!|\alpha|\!)) \quad = \quad (\!|\, z \triangledown \otimes \cdot ((\oplus \cdot (K_c \vartriangle id)) \times id) \,|\!) \,, \qquad \text{DIST}$$

which, in terms of transformations of a spine $[x_1, \ldots, x_n]^{\mathsf{T}}$ (with T denoting the initial type induced by F), can be depicted as



In this sense, $\wedge$ distributes over *false* $\triangledown$ $\vee$, $\vee$ distributes over *true* $\triangledown$ $\wedge$, and $\max_2$ distributes over $\infty \triangledown \min_2$, for example. //

114 Lemma. Let $(\tau = z \triangledown \otimes, \mathsf{Id})$ denote an initial type in insert representation. Given that $\oplus$ distributes over τ we can pull applications of $\oplus : E \times E \to E$ out of the scope of a monad comprehension if no variable bound in qs appears free in c (with $e, c : E$):

$$c \oplus \mathscr{M}\,[\![\, e \mid qs \,]\!]^{\mathsf{Id}} \quad = \quad \mathscr{M}\,[\![\, c \oplus e \mid qs \,]\!]^{\mathsf{Id}}\,. \qquad \textsc{Descope}$$

Proof. Following the proof scheme of Lemma 93, we assume $qs = v \leftarrow e' : \mathsf{T}', qs'$ and first set $qs' = \varepsilon$. If $\mathsf{T}' = \mathsf{Id}$ then Descope reads $c \oplus \mathit{let}\, v = e'\, \mathit{in}\, e = \mathit{let}\, v = e'\, \mathit{in}\, c \oplus e$ which is true because v does not appear free in c. For $\mathsf{T}' \neq \mathsf{Id}$ we calculate:

$$\begin{aligned}
& c \oplus \mathscr{M}\,[\![\, e \mid v \leftarrow e' \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& c \oplus (\!(\tau)\!)\,(\mathscr{M}\,[\![\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \textsc{Dist} \,\} \\
& (\!(z \triangledown \otimes \cdot (\oplus \cdot (K_c \vartriangle id) \times id)\,)\!)\,(\mathscr{M}\,[\![\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \textsc{Acid-Rain} \,\} \\
& ((\!(\tau)\!) \cdot \mathsf{T}'(\oplus \cdot (K_c \vartriangle id)))\,\mathscr{M}\,[\![\, e \mid v \leftarrow e' \,]\!]^{\mathsf{T}'} \\
= \quad & \{\, \textsc{Functor-}\mathscr{M}\textsc{-Fusion} \,\} \\
& (\!(\tau)\!)\,(\mathscr{M}\,[\![\, c \oplus e \mid v \leftarrow e' \,]\!]^{\mathsf{T}'}) \\
= \quad & \{\, \mathscr{M}\text{-3} \,\} \\
& \mathscr{M}\,[\![\, c \oplus e \mid v \leftarrow e' \,]\!]^{\mathsf{Id}}\,.
\end{aligned}$$

Induction step. Assume that DIST holds for $qs = qs'$.

$$
\begin{aligned}
& \mathscr{M}\,[\![\, c \oplus e \,\|\, v \leftarrow e', qs' \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \mathscr{M}\text{-5} \,\} \\
& \mathit{join}^{\mathsf{Id}}\,(\mathscr{M}\,[\![\, [\![\, c \oplus e \,\|\, qs' \,]\!]^{\mathsf{Id}} \,\|\, v \leftarrow e' \,]\!]^{\mathsf{Id}}) \\
= \quad & \{\, \text{induction hypothesis} \,\} \\
& \mathscr{M}\,[\![\, c \oplus [\![\, e \,\|\, qs' \,]\!]^{\mathsf{Id}} \,\|\, v \leftarrow e' \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \text{induction base case} \,\} \\
& c \oplus \mathscr{M}\,[\![\, [\![\, e \,\|\, qs' \,]\!]^{\mathsf{Id}} \,\|\, v \leftarrow e' \,]\!]^{\mathsf{Id}} \\
= \quad & \{\, \mathscr{M}\text{-NORM-5} \,\} \\
& c \oplus \mathscr{M}\,[\![\, e \,\|\, v \leftarrow e', qs' \,]\!]^{\mathsf{Id}} \,.
\end{aligned}
$$

□

115 **Example** (continued from Paragraph 109). Now that we are equipped with these calculus-level rewriting rules, let us revisit query Q'' and the problem of correlated quantifiers. An appeal to NEST-EX and DESCOPE can unwind the nested `exists`-clauses which finally enables the derivation of an efficient semijoin plan for Q'':

$$
\begin{aligned}
& \mathscr{Q}\,Q'' \\
= \quad & \{\, \mathscr{Q} \,\} \\
& [\![\, x \,\|\, x \leftarrow xs, [\![\, [\![\, q\,x\,z \wedge p\,y\,z \,\|\, z \leftarrow zs \,]\!]^{\mathsf{exists}} \,\|\, y \leftarrow ys \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{NEST-EX} \,\} \\
& [\![\, x \,\|\, x \leftarrow xs, [\![\, [\![\, q\,x\,z \wedge p\,y\,z \,\|\, y \leftarrow ys \,]\!]^{\mathsf{exists}} \,\|\, z \leftarrow zs \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{DESCOPE} \,\} \\
& [\![\, x \,\|\, x \leftarrow xs, [\![\, q\,x\,z \wedge [\![\, p\,y\,z \,\|\, y \leftarrow ys \,]\!]^{\mathsf{exists}} \,\|\, z \leftarrow zs \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, [\![\, p \wedge q \,\|\, qs \,]\!]^{\mathsf{exists}} = [\![\, p \,\|\, qs, q \,]\!]^{\mathsf{exists}} \,\} \\
& [\![\, x \,\|\, x \leftarrow xs, [\![\, q\,x\,z \,\|\, z \leftarrow zs, [\![\, p\,y\,z \,\|\, y \leftarrow ys \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \mathscr{M}\text{-SEMIJOIN} \,\} \\
& [\![\, x \,\|\, x \leftarrow xs, [\![\, q\,x\,z \,\|\, z \leftarrow zs \underset{\lambda z.\lambda y.p\,y\,z}{\ltimes} ys \,]\!]^{\mathsf{exists}} \,]\!]^{\mathsf{set}} \\
= \quad & \{\, \text{SEMIJOIN} \,\} \\
& xs \underset{q}{\ltimes} (zs \underset{\lambda z.\lambda y.p\,y\,z}{\ltimes} ys) \,.
\end{aligned}
$$

Similar mechanical calculations derive the algebraic forms that have been proposed by Bry to efficiently process the *open nested quantified* queries listed in [15, Section 3.2]. ◊

To conclude this chapter on hybrid query rewriting, we briefly review the transformation of another class of queries that has been discussed by Claussen, Kemper, Moerkotte, and Peithner in [34].

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

$$
\begin{array}{lll}
Q''' & = & \texttt{select}\ x \\
& & \quad \texttt{from}\ xs\ \texttt{as}\ x \\
& & \quad \texttt{where}\ \texttt{forall}\ y\ \texttt{in}\ z\ :\ q\,x\,y \\
& & \qquad \text{with } z = \begin{pmatrix} \texttt{select}\ y \\ \quad \texttt{from}\ ys\ \texttt{as}\ y \\ \quad \texttt{where}\ p\,y \end{pmatrix}
\end{array}
$$

(Claussen et al. distinguish 16 instances in this query class, each instance corresponding to a subset of $\{x, y\}$ of free variables referenced in p and q. Here, we discuss instances in which (a) the universal quantifier, and (b) the quantifier and z are correlated with the outer query block as these were identified as the most challenging in [34].)

Possible algebraic equivalents involve *set difference*, *relational division*, or a combination of grouping and counting to implement the universal quantifier, *e. g.*,

$$xs \setminus (\pi\ \mathit{outl}\,(xs \underset{\neg q \mid \lambda x.\lambda y.(x,y)}{\bowtie} (\sigma\ p\ ys)))$$

(symbol $\setminus$ denotes set difference).

The derivation of these algebraic forms is tedious, however, and the resulting expressions are judged to be inefficient and too complex to be useful as input to subsequent rewriting phases [34, 88, 105]. The case is even more complex if z is not closed (*e. g.*, if the quantifier predicate $p\,y$ is replaced by $p\,x\,y$). This renders the use of a division combinator impossible.

The universal quantifier poses no particular problem for the monad comprehension calculus. By means of $\mathscr{Q}$-Forall, the universal quantifier is translated into a comprehension to be interpreted in monad all. If z is closed, $\mathscr{Q}\,Q'''$ already emits a perfect match for the rhs of Antijoin:

$$
\begin{array}{ll}
& \mathscr{Q}\,Q''' \\
= & \quad \{\,\mathscr{Q}\,\} \\
& [\![\, x \mid\!]\ x \leftarrow xs, [\![\, q\,x\,y \mid\!]\ y \leftarrow [\![\, y \mid\!]\ y \leftarrow ys, p\,y \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{bag}} \\
= & \quad \{\,\textsc{Antijoin}\,\}
\end{array}
$$

$$
\begin{aligned}
& xs \underset{\lambda x.\lambda y.\neg q\,x\,y}{\ltimes} ([\![\, y \mid y \leftarrow ys, p\,y \,]\!]^{\mathsf{bag}}) \\
= \quad & \{\, \textsc{Select} \,\} \\
& xs \underset{\lambda x.\lambda y.\neg q\,x\,y}{\ltimes} (\sigma\ p\ ys)\ .
\end{aligned}
$$

Should z be correlated with the outer query (replace $p\,y$ by $p\,x\,y$), the derivation of an antijoin-based plan is immediate once p has been moved into the scope of the universal quantifier:

$$
\begin{aligned}
& \mathscr{Q}\,Q''' \\
= \quad & \{\, \mathscr{Q} \,\} \\
& [\![\, x \mid x \leftarrow xs, [\![\, q\,x\,y \mid y \leftarrow [\![\, y \mid y \leftarrow ys, p\,x\,y \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{bag}} \\
= \quad & \{\, \mathscr{M}\text{-}\textsc{Norm-3} \,\} \\
& [\![\, x \mid x \leftarrow xs, [\![\, q\,x\,y \mid y \leftarrow ys, p\,x\,y \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{bag}} \\
= \quad & \{\, [\![\, p \mid qs, q \,]\!]^{\mathsf{all}} = [\![\, p \vee \neg q \mid qs \,]\!]^{\mathsf{all}} \,\} \\
& [\![\, x \mid x \leftarrow xs, [\![\, q\,x\,y \vee \neg p\,x\,y \mid y \leftarrow ys \,]\!]^{\mathsf{all}} \,]\!]^{\mathsf{bag}} \\
= \quad & \{\, \textsc{Antijoin} \,\} \\
& xs \underset{\lambda x.\lambda y.\neg q\,x\,y \wedge p\,x\,y}{\ltimes} ys\ .
\end{aligned}
$$

The derived combinator queries are identical to those that have been identified as the most efficient forms by Claussen et al. [34]. ◊

117 Consider a rewriting rule set consisting of (a) defining equations for the algebraic combinators (Definition 103) as well as their associated partial match rules (Paragraph 105), (b) monad comprehension normalization rules (Definition 91), and (c) the calculus-level equivalences of this section.

Thanks to the poor syntactic variety monad comprehensions exhibit, the patterns occurring in this rule set are *simple*. The generic nature (being parametric in the monads) of the rules renders this rule set *compact*. A quite simple heuristic ordering of the rules, sketched in the introduction to this section, has already led to promising rewriting results. We could significantly improve the quality of the rewriting process with only few rules which is mainly due to their *general applicability in many instantiations*. Should the derivation of a pure combinator query fail, this rule set comes with a guaranteed fallback, namely $\mathscr{M}$, that emits catamorphisms for those query parts that could not be mapped to combinators (these parts are subject to further optimization, see Chapter 6). In this sense, the rule set is *complete*.

We believe that such a comprehension of queries makes a promising foundation upon which to rest a query optimizer for OQL-like declarative languages.

Chapter 5

Comprehending Queries

118 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. This chapter revisits three problems in the advanced query processing domain. In all three cases, it is our 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.

The three problems were previously identified and tackled by others. The first two concentrate on the efficient processing of queries featuring grouping and aggregation (Chatziantoniou and Ross, *Groupwise Processing of Relational Queries* [27] and Cluet and Moerkotte, *Efficient Evaluation of Aggregates on Bulk Types* [36]), while the last is concerned with a pure combinator query representation and its normalization (Cherniack, Malhotra, and Zdonik, *Experiences with Query Translation: Object Queries meet DB2*, [31]). We appreciate the work of these authors and the following is certainly not to lecture them. Rather, we like to argue that the techniques developed and applied in this text can contribute to the comprehension of queries.

5.1 Parallelizing Group-By Queries

119 The database backends of decision support systems frequently face SQL queries of the following general type (termed *group queries* in [26, 27]):

$$\begin{array}{lll} Q\,f\,g\,agg\,xs & \equiv & \texttt{select}\ f\,x\texttt{,}\ agg(g\,x) \\ & & \quad \texttt{from}\ xs\ \texttt{as}\ x \\ & & \texttt{group by}\ f\,x\ . \end{array}$$

Group queries extract a particular dimension or feature—described by function f—from given base data xs and then pair each data point $f\,x$ in this dimension with aggregated data $agg(g\,x)$ associated with that point; agg may be instantiated by one of the SQL aggregate functions, *e. g.*, `sum` or `max`.

There exist a number of execution plans suitable to process the queries in class Q, one of the more efficient but maybe not so obvious variants being a *self-nestjoin* of xs with itself. To find this variant, start with the monad comprehension equivalent for Q (derivable by $\mathscr{Q}$ if its definition is slightly modified to take care of syntactic sugar provided by SQL):

$$Q\,f\,g\,\mathsf{agg}\,xs \quad \equiv \quad [\![\,(f\,x, [\![\,g\,y \mid y \leftarrow xs, f\,y = f\,x\,]\!]^{\mathsf{agg}})\mid x \leftarrow xs\,]\!]^{\mathsf{set}}$$

(let agg denote the Id-monad associated with the non-collection algebra implementing the SQL aggregate agg, *e. g.*, sum or max). Then perform combinator pattern matching as we have proposed in Chapter 4 (we assume $xs : \mathsf{bag}E$ for some E):

$$\begin{array}{ll} & [\![\,(f\,x, [\![\,g\,y \mid y \leftarrow xs, f\,y = f\,x\,]\!]^{\mathsf{agg}})\mid x \leftarrow xs\,]\!]^{\mathsf{set}} \\ = & \quad \{\,\mathscr{M}\text{-Aggregate}\,\} \\ & [\![\,(f\,x, \alpha^{\mathsf{agg}}\,g\,[\![\,y \mid y \leftarrow xs, f\,y = f\,x\,]\!]^{\mathsf{bag}})\mid x \leftarrow xs\,]\!]^{\mathsf{set}} \\ = & \quad \{\,\text{product}\,\} \\ & [\![\,(f \times \alpha^{\mathsf{agg}}\,g)\,(x, [\![\,y \mid y \leftarrow xs, f\,y = f\,x\,]\!]^{\mathsf{bag}})\mid x \leftarrow xs\,]\!]^{\mathsf{set}} \\ = & \quad \{\,\mathscr{M}\text{-Project}\,\} \\ & \pi\,(f \times \alpha^{\mathsf{agg}}\,g)\,[\![\,(x, [\![\,y \mid y \leftarrow xs, f\,y = f\,x\,]\!]^{\mathsf{bag}})\mid x \leftarrow xs\,]\!]^{\mathsf{set}} \\ = & \quad \{\,\text{Nestjoin}\,\} \\ & \pi\,(f \times \alpha^{\mathsf{agg}}\,g)\,(xs \underset{\lambda x.\lambda y.f\,x = f\,y \mid \lambda x.\lambda y.(x,y)}{\triangle} xs)\ . \end{array}$$

120 In [26, 27], Chatziantoniou and Ross propose to take a rather different three-step route to execute queries of type Q:

(a) Separate the data points in dimension f of xs in a preprocessing step, *i. e.*, partition xs with respect to f.

(b) Evaluate a simplified variant Q' of Q on each partition. In particular, Q' does not need to take care of grouping. Let ps denote a partition of xs, then we have

$$Q'\ f\ g\ agg\ ps \quad \equiv \quad \begin{array}{l}\texttt{select}\ f\ x\texttt{,}\ agg(g\ x)\\ \quad\texttt{from}\ ps\ \texttt{as}\ x\ ,\end{array}$$

or, equivalently,

$$Q'\ f\ g\ \mathsf{agg}\ ps \quad \equiv \quad [\![\, (f\ x, [\![\, g\ y \mid y \leftarrow ps \,]\!]^{\mathsf{agg}}) \mid x \leftarrow ps \,]\!]^{\mathsf{set}}\ .$$

(c) Finally, join the results obtained in step (b) to form the query response.

This strategy clearly shows its benefit in stage (b): first, since xs has been split into disjoint partitions during the preprocessing step, we may execute Q' on the different partitions in parallel. Second, there is a chance to process Q' in main memory should the partitions of xs fit. Measurements reproduced in [26, 27] show the performance gains in terms of time and I/O costs to compensate for the effort spent in the partitioning and joining stages.

121 In [26, 27], classical relational algebra is the language at which the translation of group queries is targeted. This choice of query representation introduces subtleties.

Relational algebra lacks canonical forms to express the grouping and aggregation operations found in Q. Chatziantoniou and Ross thus propose to understand Q as a *syntactical* query class: the membership of a specific query in this class and thus the applicability of the partitioning strategy is decided by inspection of the SQL parse tree for that query. (Actually, query graphs are extracted from the parse trees and a query graph level criterion decides the group query property.)

Relational algebra falls short to provide idioms that could express the preprocessing, *i. e.*, partitioning, step of the strategy. To remedy the situation, Chatziantoniou and Ross attribute the nodes of the query graphs to indicate which partition is represented by a specific node.

Finally, the core stage (b) of the partitioning strategy has no equivalent at the target language level as well. Classical relational algebra is unable to express the iteration (or parallel application) inherent to this stage.[1]

[1] Chatziantoniou and Ross implemented this step *on top* of the relational backend and thus outside the relational domain.

Facing this mix of query representations (SQL syntax, query graphs, relational algebra, iteration), it is considerably hard to assess the correctness of Chatziantoniou and Ross' parallel processing strategy for class Q.

122 Reasoning in the monad comprehension calculus and its associated combinator algebra can significantly simplify the matter. Once expressed using monadic notions, we can construct a correctness proof for the strategy which is basically built from the unfolding of definitions and normalization steps alone (and thus could possibly be conducted automatically, although we consider automated proofs outside the scope of this text). Let us proceed by filling the two gaps (partitioning and iteration) that relational algebra has left open.

First, partitioning the base data collection xs with respect to a function f is expressible in the monad comprehension calculus simply as (let T denote a collection monad and E' some equality type, *cf.* Paragraph 78)

$$\begin{array}{lcl} \mathit{partition} & : & (E \to E') \to \mathsf{T}E \to \mathsf{set}\mathsf{T}E \\ \mathit{partition}\ f\ xs & = & [\![\, [\![\, y \,|\, y \leftarrow xs, f\,x = f\,y \,]\!]^{\mathsf{T}} \,|\, x \leftarrow xs \,]\!]^{\mathsf{set}}\,, \end{array} \qquad \textsc{Partition}$$

which builds a set of disjunct partitions so that all elements inside one partition agree on feature f.

Second, recall from Section 2.5 that the type or map functors provide the principle iteration abstractions in our categorical setting. At the combinator algebra level, iteration is at our disposal by means of the combinator $\pi\ f = \mathsf{T}f$ (see Project). Map functor $\mathsf{T}f$ also adequately encodes parallel application of f to the elements of its (collection-typed) argument: the applications of f to the spine elements do not interfere and can be evaluated in parallel. See, for example, the work of Hill in which a complete theory of *data-parallel programming* is developed on top of map functors [62].

With the definition of Q' from Paragraph 120 we can now recast the parallel grouping plan as

$$\mu\left(\pi\,(Q'\ f\ g\,\mathsf{agg})\,(\mathit{partition}\ f\ xs)\right).$$

The following rewrite finally establishes the equivalence of this plan and $Q\ f\ g\,\mathsf{agg}\ xs$ and thus provides a purely calculational proof of the correctness of parallel grouping.

$$\begin{array}{ll} & \mu\left(\pi\,(Q'\ f\ g\,\mathsf{agg})\,(\mathit{partition}\ f\ xs)\right) \\ = & \quad \{\ \textsc{Unnest}\ \} \\ & [\![\, y \,|\, ys \leftarrow \pi\,(Q'\ f\ g\,\mathsf{agg})\,(\mathit{partition}\ f\ xs), y \leftarrow ys \,]\!]^{\mathsf{set}} \end{array}$$

$$
\begin{aligned}
&= \quad \{\,[68]: \quad [\![\, e \mid qs, v \leftarrow \mathsf{T}\, f\, e', v' \leftarrow v, qs' \,]\!]^{\mathsf{T}} \\
&\qquad\qquad = \quad [\![\, e \mid qs, v \leftarrow e', v' \leftarrow f\, v, qs' \,]\!]^{\mathsf{T}} \quad \} \\
&\quad [\![\, y \mid ys \leftarrow \mathit{partition}\, f\, xs, y \leftarrow (Q'\, f\, g\, \mathsf{agg})\, ys \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\text{unfold } Q'\,\} \\
&\quad [\![\, y \mid ys \leftarrow \mathit{partition}\, f\, xs, \\
&\qquad y \leftarrow [\![\, (f\, x, [\![\, g\, x' \mid x' \leftarrow ys \,]\!]^{\mathsf{agg}}) \mid x \leftarrow ys \,]\!]^{\mathsf{set}} \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\mathscr{M}\text{-Norm-3}\,\} \\
&\quad [\![\, (f\, x, [\![\, g\, x' \mid x' \leftarrow ys \,]\!]^{\mathsf{agg}}) \mid ys \leftarrow \mathit{partition}\, f\, xs, x \leftarrow ys \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\text{unfold } \mathit{partition}\,\} \\
&\quad [\![\, (f\, x, [\![\, g\, x' \mid x' \leftarrow ys \,]\!]^{\mathsf{agg}}) \mid \\
&\qquad ys \leftarrow [\![\, [\![\, z' \mid z' \leftarrow xs, f\, z' = f\, z \,]\!]^{\mathsf{bag}} \mid z \leftarrow xs \,]\!]^{\mathsf{set}}, x \leftarrow ys \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\mathscr{M}\text{-Norm-3}\,\} \\
&\quad [\![\, (f\, x, [\![\, g\, x' \mid x' \leftarrow [\![\, z'' \mid z'' \leftarrow xs, f\, z'' = f\, z \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{agg}}) \mid \\
&\qquad z \leftarrow xs, x \leftarrow [\![\, z' \mid z' \leftarrow xs, f\, z' = f\, z \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\mathscr{M}\text{-Norm-3}\,\} \\
&\quad [\![\, (f\, x, [\![\, g\, z'' \mid z'' \leftarrow xs, f\, z'' = f\, z \,]\!]^{\mathsf{agg}}) \mid \\
&\qquad z \leftarrow xs, x \leftarrow [\![\, z' \mid z' \leftarrow xs, f\, z' = f\, z \,]\!]^{\mathsf{bag}} \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\mathscr{M}\text{-Norm-3}\,\} \\
&\quad [\![\, (f\, z', [\![\, g\, z'' \mid z'' \leftarrow xs, f\, z'' = f\, z \,]\!]^{\mathsf{agg}}) \mid z \leftarrow xs, z' \leftarrow xs, f\, z' = f\, z \,]\!]^{\mathsf{set}} \\
&= \quad \{ \quad [\![\, e\, (f\, v) \mid qs, v \leftarrow e', v' \leftarrow e', f\, v = f\, v', qs'\, (f\, v) \,]\!]^{\mathsf{set}} \\
&\qquad = \quad [\![\, e\, (f\, v') \mid qs, v' \leftarrow e', qs'\, (f\, v') \,]\!]^{\mathsf{set}} \quad \} \\
&\quad [\![\, (f\, z, [\![\, g\, z'' \mid z'' \leftarrow xs, f\, z'' = f\, z \,]\!]^{\mathsf{agg}}) \mid z \leftarrow xs \,]\!]^{\mathsf{set}} \\
&= \quad \{\,\text{query class } Q\,\} \\
&\quad Q\, f\, g\, \mathsf{agg}\, xs\ .
\end{aligned}
$$

5.2 Aggregation by Decomposition

123 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 one to assume. The SQL queries in the following class, with some close resemblance to Kim's *type JA* queries [69], provide instances of this phenomenon (once more, let *agg* denote an SQL aggregate function; replace $\otimes$ by a comparison operator, *e. g.*, `=` or `<=`, below

we assume $\ominus$ = `<=`):

$$Q\ f\ \ominus\ agg\ xs\ ys \quad \equiv \quad \begin{array}{l}\texttt{select}\ f\ x\ (agg\ z)\\ \quad\texttt{from}\ xs\ \texttt{as}\ x\\ \qquad\qquad\text{with}\ z = \left(\begin{array}{l}\texttt{select}\ y\\ \quad\texttt{from}\ ys\ \texttt{as}\ y\\ \texttt{where}\ y \ominus x\end{array}\right)\ .\end{array}$$

To be more precise, assume $xs : \mathsf{T}E$, $ys : \mathsf{T}'E$, $f : E \to E \to E'$, $\ominus : E \times E \to Bool$ and let `agg` denote the identity monad realizing *agg* (see Section 5.1). Q is implemented by the combinator form

$$\pi\ (\lambda x.f\ x\ (\alpha^{\texttt{agg}}\ (\sigma\ (\lambda y.y \ominus x)\ ys))\ xs\ .$$

We always have the naive catamorphic interpretation at hand but Kim already described a superior evaluation strategy for Q in [69]: (a) join collections *xs* and *ys* with respect to $\ominus$, then (b) group the join result, and finally (c) aggregate the groups separately to form the overall query result.

Although an improvement, given that the join in step (a) in general will *not* be an equi-join of *xs* and *ys*, we need $O(|xs| \cdot |ys|)$ space to hold the intermediate result. The time complexity of this strategy is clearly the same.

124 Cluet and Moerkotte realized that an optimizer can reduce both the space and time needed to evaluate queries in Q, provided that the aggregate *agg* exhibits a *decomposition* property [36]. Importing the work presented in [36] into our system of categorical type constructors is straightforward.

For $ys : \mathsf{T}'E$, the aggregate function $agg : \mathsf{T}'E \to E$ is *decomposable*, if we can find functions $\alpha : \mathsf{T}'E \to E$ and $\oplus : E \to E \to E$ so that

$$agg\ ys = (\alpha\ ys') \oplus (\alpha\ ys'') \qquad \text{with} \qquad ys = union^{\mathsf{T}'}\ (ys', ys'')\ .$$

Note that this almost characterizes *agg* as a homomorphism of algebras in union representation (*cf.* Section 2.10).

The decomposition property of aggregate *agg* paves the way for an efficient but non-standard evaluation strategy for Q: since *agg* is decomposable we can *incrementally update* the aggregate value for each group every time we encounter a member of that group [52]. Each group's current aggregate state is kept in a support data structure—the $\ominus$-*table*—which forms the core of the algorithm.

The following pseudo-code sketches the implementation of this procedure:

$new(t, \oslash)$;
foreach x **in** xs
    $ins(t, x)$;
$sort(t)$;
foreach y **in** ys
    $update(t, y)$;
$eval(t)$;

$new(t, \oslash)$: Allocate an empty $\oslash$-table t of type $\mathsf{list}(Num \times E \times E)$. For each list member, the first tuple component protocols the multiplicity of element $x{:}E$ in xs; x is stored in the second component; the last component reflects the aggregate state of the group x is associated with.

$ins(t, x)$: $t := (1, x, \alpha\ empty^{\mathsf{T}'}) \overset{\mathsf{list}}{:} t$.

$sort(t)$: Sort list t on second component with respect to $\oslash$, remove duplicates but update multiplicity and aggregate components accordingly.

$update(t, y)$: Incremental update: search for tuple (n, x, a) with smallest x such that $y \oslash x$. Replace this tuple in t with $\big(n, x, a \oplus (\alpha\,(sng^{\mathsf{T}'}\ y))\big)$.

$eval(t)$: $a' := \alpha\ empty^{\mathsf{T}'}$;
    foreach (n, x, a) **in** t **do**
        $a' := a \oplus a'$;
        replace tuple by (n, x, a');
    od;

Note that the actual groups are never constructed. The $\oslash$-table thus has space requirements of $O(|xs|)$ as there are no more groups than elements in xs. Furthermore, should $\oslash$ impose an order on E, we can exploit binary search to implement the *update* operation on the table. This reduces the time complexity to $O(|xs| \cdot \log |xs| + |ys| \cdot \log |xs|)$.

125 There remains the question of the *effective* applicability of the $\oslash$-table approach. Among other concluding remarks in [36], Cluet and Moerkotte marked two open issues concerning effectiveness:

(a) How to enable the optimizer to automatically deduce functions α and $\oplus$ for a given query, specifically for a given aggregate *agg*?

(b) Clearly, there is a larger class of queries which can benefit from $\oslash$-table-driven processing. A $\oslash$-table can support the evaluation of non-equi-joins like $\bowtie(\lambda x.\lambda y.x \oslash y)\ f\ xs\ ys$ and may additionally prove efficient during nestjoin processing (*cf.* query class Q' of Paragraph 108). A syntactical characterization as it is provided by Q seems to be too coarse.

We can provide effective answers to both questions once we have re-expressed Q via our catamorphic (monadic) comprehension of queries.

126 There is an obvious connection between the almost homomorphic nature of a decomposable aggregate function and its catamorphic realization in our world of categorical query semantics. This observation provides an answer to issue (a) raised in the previous paragraph.

Application of mapping $\mathscr{M} \cdot \mathscr{Q}$ makes the catamorphic implementation of Q accessible (let $\tau = e \triangledown \otimes : 1 + E \times E \to E$ denote the algebra in insert representation that is associated with the identity monad agg, see Definition 64):

$$\begin{array}{ll} & (\mathscr{M} \cdot \mathscr{Q})\,(Q\ f\ \oslash\ agg\ xs\ ys) \\ = & \quad \{\, \mathscr{Q} \,\} \\ & \mathscr{M}\,[\![\, f\ x\ [\![\, y \mid y \leftarrow ys, y \oslash x \,]\!]^{\mathsf{agg}} \mid x \leftarrow xs \,]\!]^{\mathsf{T}} \\ = & \quad \{\, \mathscr{M}\text{-}3 \,\} \\ & [\![\, f\ x\ (\!|\tau|\!)\ [\![\, y \mid y \leftarrow ys, y \oslash x \,]\!]^{\mathsf{T}'}) \mid x \leftarrow xs \,]\!]^{\mathsf{T}}\ , \end{array}$$

i. e., we have $agg = (\!|\tau|\!) : \mathsf{T}'E \to E$.

Now note that the $\oslash$-table algorithm exclusively performs element-wise insertions of elements into the $\oslash$-table (by means of function *update*) which, in turn, lead to element-wise updates of the aggregate state. This suggests to take an insert representation view of the decomposition property: aggregate function $agg : \mathsf{T}'E \to E$ is *decomposable*, if we can find functions $\alpha : \mathsf{T}'E \to E$ and $\oplus : E \to E \to E$ so that

$$agg\ ys = y \oplus (\alpha\ ys') \qquad \text{with} \qquad ys = y \overset{\mathsf{T}'}{:} ys'\ .$$

Being so far, we can almost immediately read off suitable definitions for α and $\oplus$:

$$\begin{array}{ll} & agg\ ys \\ = & \quad \{\, ys = y \overset{\mathsf{T}'}{:} ys', agg = (\!|\tau|\!) \,\} \\ & (\!|\tau|\!)\ (y \overset{\mathsf{T}'}{:} ys') \\ = & \quad \{\, \tau = e \triangledown \otimes, \textsc{Cata-Ins-Rep} \,\} \end{array}$$

$$
\begin{array}{ll}
 & y \otimes (\!|e \triangledown \otimes|\!)\ ys') \\
\Rightarrow & \quad \{\,\text{aggregate decomposition}\,\} \\
 & \quad \alpha \;=\; (\!|e \triangledown \otimes|\!) \;=\; (\!|\tau|\!) \\
 & \wedge\ \oplus \;=\; \otimes\,.
\end{array}
$$

127 To tackle the effective detection of (sub)queries that are subject to $\lessdot$-table-based evaluation, observe that the $\lessdot$-table algorithm is obviously capable of processing queries of the general form

$$[\![\, f\ x\ [\![\, y\ [\![\, y \leftarrow ys, y \lessdot x \,]\!]^{\mathsf{T}'}\ [\![\, x \leftarrow xs \,]\!]^{\mathsf{T}}\,.$$

The monad comprehension form provides a *structural* characterization of the $\lessdot$-table processible queries. This structural encoding of the query class is more effective and *useful*[2] than the simple syntactical query pattern Q. The monad comprehension form covers all application scenarios envisioned in [36], including those that were understood as possible future extensions:

(a) (Sub)queries in class Q are detected since $\mathscr{Q}\,Q$ is an instance of the above monad comprehension form (with $\mathsf{T}' = \mathsf{agg}$, see the previous paragraph).

(b) For the particular queries in which monad T' is instantiated by a collection monad (as opposed to an Id-monad), we obtain a perfect match with the defining pattern for $\triangle$ (NESTJOIN), specifically $\triangle \lessdot f\ xs\ ys$. In fact, this is a rediscovery of a remark in [36] in which the possible applicability of $\lessdot$-tables to the efficient implementation of nestjoin-like operators has been outlined.

(c) Finally, the cases in which T' denotes a collection monad are not special with respect to an automatic derivation of α and $\oplus$. To see this, simply repeat the calculation of the previous paragraph (with $\tau = [\,]^{\mathsf{T}'} \triangledown \vdots^{\mathsf{T}'}$):

$$
\begin{array}{lll}
 & \alpha \;=\; (\!|\,[\,]^{\mathsf{T}'} \triangledown \vdots^{\mathsf{T}'}|\!) \;=\; id & (\textsc{Cata-Reflect}) \\
\wedge & \oplus \;=\; \vdots^{\mathsf{T}'}\,. &
\end{array}
$$

These choices for α and $\oplus$ exactly configure the $\lessdot$-table algorithm for the processing of non-equi-joins and nestjoins as it has been proposed in [36] .

[2] Here, "*useful*" is used in the sense of Cluet and Moerkotte in [36]: "*[Further research should] come up with a useful characterization of the corresponding queries. By useful we mean that an optimization can easily detect the applicability of θ-tables.*" (In [36], Cluet and Moerkotte use symbol θ in place of $\lessdot$.)

Note that the above insights are completely based on rather simple pattern matches and unifications with the monad comprehension-based encoding of the query class. It is this encoding which enables a query optimizer to effectively exploit the ⋉-table idea.

5.3 Normalization of KOLA Combinator Queries

In a series of articles, Cherniack and Zdonik developed the KOLA query algebra as a target algebra for the translation and optimization of OQL queries [28, 29, 30, 31, 32, 33]. Being a successor to the AQUA algebra [108, 109], KOLA follows a design that emphasizes the use of few but generic higher-order query primitives. All KOLA primitives are combinators: KOLA has no notion of variables, scope, and binding, whatsoever. In this sense, KOLA exhibits similarities with the combinator algebra of Chapter 4.

Mainly because the meaning of variables is strongly tied to their scope, Cherniack and Zdonik argue that programs expressed in variable-based query languages are hard to optimize: the analysis of subqueries is inherently tightly coupled with the inspection of the global variable environment inherited from the enclosing query. There is no true locality of query analysis. This, in turn, complicates the formalization of query transformations and rewriting rules. The applicability of a rule is not only dependent on a structural (syntactical) match with its lhs but, in general, additionally on the current variable environment.

KOLA queries, lacking variables, enable pure *syntactical* rule matching and rewriting. The detection of a query pattern requires local analysis only. In *any* expression of the form (`join` denotes KOLA's θ-join combinator; the predicate is evaluated against pairs built from the elements from collections xs and ys)

$$\texttt{join}\,((p \cdot outl) \wedge q)\; xs\; ys$$

p may be pushed down the expression tree for xs since p is applied after the projection $outl$ and thus cannot depend on the second join argument ys (an assurance justified by p's nature of being a combinator). Here, predicate p's independence of ys is expressed structurally rather than by (the absence of) variable references. Cherniack and Zdonik exploited this feature of KOLA to prove a large library of rewriting rules correct with the help of an automated theorem prover [29, 30].

128 Let us emphasize one consequence of the combinator approach. KOLA uses nested pairs to represent tuples and structured objects much like we did

in the preparations to the discussion of mapping $\mathscr{Q}$ in Paragraph 78. A typical KOLA query is swamped with primitives that build and destruct pairs as there are no means to reference a specific tuple or object component besides an explicit sequence of projections on pairs. Similar remarks apply to KOLA rewrite rules. The rule sets that implement a specific global query rewrite—like the KOLA normalization we will examine in detail below—tend to be large: a rewrite is not only concerned with the global transformation goal but also needs to appeal to a variety of auxiliary rules which are primarily designed to, *e. g.*, manipulate nested pairs.

The large number of rules calls for sophisticated rewriting strategies to prevent a KOLA optimizer from getting lost in the vast space of equivalent expressions it can generate from these rule sets. This observation led Cherniack and Zdonik to the design and implementation of a control language for rule application, KOKO [32, 33].

While there are clear benefits of combinator query languages (*cf.* Paragraph 99), we believe that the combinator paradigm considerably complicates the implementation of certain query translation stages. To exemplify, we will try to comprehend a complex KOLA query rewrite [31] in terms of the monad comprehension calculus. A change in the point of view on query representation can have significant effects on the complexity of the specification and implementation of optimization goals. This is the point we are trying to make before we conclude the current chapter.

129 In [31], Cherniack et al. describe an OQL frontend which is designed to run on top of the relational DBMS DB2. OQL queries are translated into KOLA, then optimized. A subsequent stage generates DB2 SQL statements from KOLA queries. For the OQL `select-from-where` block

$$\begin{array}{l}\texttt{select}\ f\,(x_1,\ldots,x_n)\\ \quad\texttt{from}\ e_1\ \texttt{as}\ x_1\texttt{,}\ \ldots\texttt{,}e_n\ \texttt{as}\ x_n\\ \quad\texttt{where}\ p\,(x_1,\ldots,x_n)\ ,\end{array}$$

the frontend emits the KOLA form

$$\big((\texttt{iterate}\,p\,f)\cdot(\texttt{unnest}\,id\,(\texttt{K}\,f_n))\cdot\cdots\cdot(\texttt{unnest}\,id\,(\texttt{K}\,f_1))\cdot\texttt{singleton}\big)\,()$$

in which the combinators and symbols are defined to mean (here, we recast the combinators as monadic functions to give their precise semantics as well as to prepare the things to come)

$$\texttt{unnest}\,f\,g\,xs \quad = \quad [\![\, f\,(x,y)\,[\!]\, x \leftarrow xs, y \leftarrow g\,x\,]\!]^{\mathsf{bag}}$$

$$\begin{aligned}
\texttt{iterate}\, p\, f\, xs &= [\![\, f\, x \mid x \leftarrow xs, p\, x \,]\!]^{\mathsf{bag}} \\
\texttt{singleton} &= unit^{\mathsf{bag}} \\
\texttt{K} &= K \\
() &= \text{an arbitrary dummy value} \\
f_i &= \text{KOLA form to represent } e_i.
\end{aligned}$$

This translation scheme is applicable to arbitrary e_i (*e. g.*, path expression, subqueries) and does not restrict the e_i to be constant collections (in the KOLA query above, the f_i may be arbitrarily complex KOLA functions).

The goal of the subsequent optimization stage is to derive a join query from this initial KOLA form.[3] Cherniack et al. proceed in two steps to achieve this: (a) *normalize* the initial form, *i. e.*, rewrite it into an equivalent query in which a chain of `unnest` combinators is invoked on the f_i directly, thus rendering the auxiliary construction involving the dummy value () void, and (b) derive a join query from the normalized expression which is more amenable to join detection as the initial form.

The KOLA implementation of this rewrite is too complex to reproduce here as it falls back on 46 rewrite rules and about 100 lines of rule firing control code [31].

130 A monad comprehension semantics for KOLA, in contrast, provides a straightforward, almost mechanical, derivation of a join query from the initial KOLA form. There is nothing more involved than the unfolding of combinator definitions given in the previous paragraph and the application of monad comprehension normalization rules from Definition 91. The simplicity of this process makes external guidance through specific rule firing control code obsolete.

The appeal to comprehension normalization comes at no surprise once we analyze the involved KOLA rewriting rules a bit deeper. A significant ratio of these rules actually implement specific calculus normalization steps. The KOLA rule (with $\texttt{shr}\,((x,y),z) = (x,(y,z))$; note that `shr` is one of the two arrows that establish the isomorphism between nested products, *cf.* Paragraph 78)

$$\begin{aligned}
&(\texttt{unnest}\,(outr \cdot \texttt{shr})\,(f \cdot outr)) \cdot (\texttt{unnest}\, id\, (\texttt{K}\, e)) \cdot \texttt{singleton} \quad \rightsquigarrow \\
&\quad \texttt{K}\,(\texttt{unnest}\, id\, f\, e)\ ,
\end{aligned}$$

[3] This join query forms the input to the last stage which finally emits code to implement the OQL `select-from-where` block using DB2 SQL.

for example, bundles the application of normalization rules $\mathscr{M}$-Norm-2 and $\mathscr{M}$-Norm-3:

$$
\begin{aligned}
& \big((\texttt{unnest}\,(outr \cdot \texttt{shr})\,(f \cdot outr)) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,e)) \cdot \texttt{singleton}\big)\,e' \\
= \quad & \{\,\text{unfold KOLA combinators}\,\} \\
& [\![\,(outr \cdot \texttt{shr})\,(x,y) \mid x \leftarrow [\![\,(x',y') \mid x' \leftarrow unit^{\mathsf{bag}}\,e', y' \leftarrow K_e\,x' \,]\!]^{\mathsf{bag}}, \\
& \qquad\qquad y \leftarrow (f \cdot outr)\,x \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\mathscr{M}\text{-Norm-2}\,\} \\
& [\![\,(outr \cdot \texttt{shr})\,(x,y) \mid x \leftarrow [\![\,(e',y') \mid y' \leftarrow e \,]\!]^{\mathsf{bag}}, y \leftarrow (f \cdot outr)\,x \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\mathscr{M}\text{-Norm-3}\,\} \\
& [\![\,(outr \cdot \texttt{shr})\,((e',y'),y) \mid y' \leftarrow e, y \leftarrow (f \cdot outr)\,(e',y') \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,outr, \texttt{shr}\,\} \\
& [\![\,(y',y) \mid y' \leftarrow e, y \leftarrow f\,y' \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\texttt{unnest}, \texttt{K}\,\} \\
& \texttt{K}\,(\texttt{unnest}\,id\,f\,e)\,e'
\end{aligned}
$$

(from which the rule follows immediately by the principle of extensionality).

The normalization of the KOLA form goes off as a canonical monad comprehension normalization once we have unfolded the monadic KOLA combinator definitions:

$$
\begin{aligned}
& \big((\texttt{iterate}\,p\,f) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_n)) \cdot \cdots \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_1)) \cdot \\
& \quad \texttt{singleton}\big)\,() \\
= \quad & \{\,\text{unfold } \texttt{singleton}, \texttt{unnest}\,\} \\
& \big((\texttt{iterate}\,p\,f) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_n)) \cdot \cdots \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_2))\big) \\
& \quad [\![\,(x_1,y_1) \mid x_1 \leftarrow unit^{\mathsf{bag}}\,(), y_1 \leftarrow K_{f_1}\,x_1 \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\mathscr{M}\text{-Norm-2}\,\} \\
& \big((\texttt{iterate}\,p\,f) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_n)) \cdot \cdots \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_2))\big) \\
& \quad [\![\,((),y_1) \mid y_1 \leftarrow f_1 \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\text{unfold } \texttt{unnest}\,\} \\
& \big((\texttt{iterate}\,p\,f) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_n)) \cdot \cdots \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_3))\big) \\
& \quad [\![\,(x_2,y_2) \mid x_2 \leftarrow [\![\,((),y_1) \mid y_1 \leftarrow f_1 \,]\!]^{\mathsf{bag}}, y_2 \leftarrow K_{f_2}\,x_2 \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\mathscr{M}\text{-Norm-3}\,\} \\
& \big((\texttt{iterate}\,p\,f) \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_n)) \cdot \cdots \cdot (\texttt{unnest}\,id\,(\texttt{K}\,f_3))\big) \\
& \quad [\![\,(((),y_1),y_2) \mid y_1 \leftarrow f_1, y_2 \leftarrow f_2 \,]\!]^{\mathsf{bag}} \\
= \quad & \{\,\text{repeat last two steps } n-3 \text{ times}\,\} \\
& (\texttt{iterate}\,p\,f) \\
& \quad [\![\,((\cdots((),y_1)\cdots),y_n) \mid y_1 \leftarrow f_1, \ldots, y_n \leftarrow f_n \,]\!]^{\mathsf{bag}}
\end{aligned}
$$

$$= \quad \{\,\text{isomorphism of nested products, } \texttt{iterate}\,\}$$
$$[\![\, f\,(y_1, \ldots, y_n) \mid y_1 \leftarrow f_1, \ldots, y_n \leftarrow f_n, p\,(y_1, \ldots, y_n)\,]\!]^{\mathsf{bag}}\,.$$

The generation of an equivalent KOLA `join` query is now immediate by means of the combinator patterns JOIN and $\mathscr{M}$-JOIN.

Note, in addition, that the normalized query is exactly what mapping $\mathscr{Q}$ would directly emit for the OQL `select-from-where` block shown in the previous paragraph. We believe that a pure combinator representation renders the early stages of query processing—namely translation and normalization—an unnecessarily hard task. An employment of the monad comprehension calculus clearly shows its elegance here.

Chapter 6

Query Deforestation

131 Recall that a combinator query exhibits a tree-like structure in which the nodes represent abstractions of the spine transformers (physical operators) supplied by the query engine backend (*cf.* the introduction to Chapter 4). It is a demanding task to efficiently schedule the flow of data from the leaves of this processing tree—*i. e.*, the complex values which are actually materialized on persistent storage—to its root which represents the query result.

Query engine models that assign combinators to separate operating system processes or threads connected by inter-process communication (IPC) facilities (to control the flow of execution) as well as disk files (to communicate temporary results) have been found to be inefficient: the operating system scheduler, necessary process context switches, IPC, and I/O of temporary results incur an overhead that dominates the overall query cost by far [53, 54].

Different lines of research [46, 47, 78] thus led to the development of single-process query engines. A combinator query is compiled into a single *iterative* or *recursive procedure* which is then executed inside the monolithic query engine process. Whenever possible, these approaches strive for a *stream-based* or *pipelined* query execution to avoid the I/O of temporary data. The query engine benefits from streaming since data is addressed and brought in from persistent storage only once. Further processing is in-memory and no intermediate writes to the persistent store and subsequent reads of materialized intermediate results occur.

132 The derivation of a stream-based query program from a combinator expression is a problem that has been primarily tackled on the implementation level only. In a sense, the forthcoming material drags this query processing phase out of the hands of the database implementors and instead aims at

making the matter accessible to our categorical tools—and thus comprehensible by an optimizer.

6.1 Streaming

133 Given that the query compiler arranges the query plan as a tree structure of independent combinators, it is not immediate how an efficient streaming procedure may be automatically derived. Combinators consume their input spine as a whole, transform it, and then emit the transformed spine as an intermediate result. We are not given any reference to "reach inside" and collect partial results early in order to immediately pass them to the parent combinator. The possibility to do so would be the key to realize streaming but violates the combinator encapsulation.

These observations led to a spectrum of solutions to this dilemma, ranging from implementation level modifications to the query engine to algebraic transformation of execution plans.

134 In his seminal work on the engineering of query engines for large databases, Graefe proposed an iterator *implementation discipline* for combinators that facilitates streaming [53, 54]. Any combinator implements an interface specifying a function *next* and a constant *empty!*. Application of this iterator discipline does not alter the initial combinator tree but leads the combinators to schedule each other inside the tree structure.

Given a combinator tree, query evaluation is driven by the root combinator *on demand*: repeated calls to *next* request the production of the next stream element (*e. g.*, tuple, complex value, or object) by the combinator's child node(s). The children recursively forward the *next* call until a leaf combinator can satisfy the request through read access to the data source it is associated with. Each combinator passes stream elements upwards the processing tree, possibly after application of a combinator-specific action to the elements. Combinators iterate this procedure until the special stream element *empty!* signals that the child nodes are exhausted.

Following this discipline, a sketch of the implementation of function *next* for the $\pi\, f$ combinator (see PROJECT) would thus read (with $\mathit{empty?}\, x \Leftrightarrow x = \mathit{empty!}$):

$$
\begin{array}{l}
x := \mathit{next};\\
\texttt{if}\ (\texttt{not}\ (\mathit{empty?}\, x))\ \texttt{then}\\
\quad\ \texttt{return}\ (f\ x);\\
\texttt{return}\ \mathit{empty!};
\end{array}
$$

135 The on-demand scheduling of combinators to thread the next stream element (the head of the stream's tail) through the combinator tree bears close resemblance with the *lazy semantics of evaluation* in functional programming systems [92]. Buneman, Frankel, and Rishiyur actually devised a combinator query language with lazy semantics, FQL, in [18].

A lazy stream xs is represented in its *weak head normal form* [92] $x /\!\!/ s$ in which x denotes the stream's head element and s represents a *suspension*, *i. e.*, a function that—if ever evaluated—returns the stream $x' /\!\!/ s'$, the tail of xs. A suspension, or *closure*, $\ulcorner f, x \urcorner$ bundles a function f and its argument x. Function f is not applied until the suspension is actually evaluated (*forced*):

$$eval \ulcorner f, x \urcorner \quad = \quad f\, x\ .$$

Iterating over a stream then means to repeatedly evaluate its suspended tail.

In continuation from the previous paragraph, we could implement the π combinator in the suspension model as

$$\begin{aligned} \pi\, f\, xs \quad &= \quad \ulcorner f, x \urcorner /\!\!/ \ulcorner \pi\, f, eval\, s \urcorner \\ &\qquad \text{with } xs = x /\!\!/ s\ . \end{aligned}$$

Note that π does not yet apply f to the stream's head x but rather creates a suspension $\ulcorner f, x \urcorner$. This saves the cost for the application of f to x should the stream element be discarded later on.

Suspensions are, once again, a device that postpones the solution of the streaming problem until query engine implementation. It is the point of this chapter to show that we can benefit from pulling the issue up to the query representation level. At this level, streaming programs are derived by rewriting much like in a transformation-based query optimizer. Let us proceed with a short review of a transformational approach due to Freytag and Goodman [44, 45, 46, 47] before the next section discusses a remarkably simple method to achieve streaming in the catamorphic query model.

136 Guided by observations about the advantages of single-process query engines, Freytag and Goodman set out with the principal goal to transform combinator queries into monolithic *iterative* procedures [44, 45, 46, 47]. The emitted code was designed to be *compiled* and then linked against a relational database backend.

During the derivation of an iterative program form, Freytag and Goodman break the encapsulation inherent to the combinator notion (*cf.* Paragraph 99): given a combinator query Q, combinators in Q are *unfolded*, *i. e.*, replaced by their defining expressions—recursive programs expressed in a subset of LISP—with the principle aim to fuse these with the definitions of

neighboring combinators. To continue our treatment of the π combinator in different streaming models, unfolding $\pi\, f\, xs$ would reveal its *control structure* and *action* as the LISP form on the rhs

$$\begin{array}{lcl} \pi\, f\, xs & = & (\mathit{if}\ (\mathit{empty?}\, xs) \\ & & \quad \mathit{empty!} \\ & & \quad (\mathit{cons}\, (f\, (\mathit{hd}\, xs))\, (\pi\, f\, (\mathit{tl}\, xs)))) \ . \end{array}$$

Actual fusion is then performed through simplification of the unfolded LISP forms (all occurring forms are restricted to use a referentially transparent subset of LISP thus facilitating the specification of equivalence-preserving simplification rules). The rules aim to transform the program for Q so that occurrences of the defining expression for Q can be detected. On detection, these subexpressions are replaced by recursive calls to Q itself (*folding*). In a final step, recursion is traded for iteration.

To illustrate, consider the following sketch of the *unfold-simplify-fold* steps for the combinator query $Q\, f\, g\, xs = \pi\, g\, (\pi\, f\, xs)$:

$$\begin{array}{ll} & Q\, f\, g\, xs \\ = & \quad \{\, \text{unfold defining query for } Q \,\} \\ & (\pi\, g\, (\pi\, f\, xs)) \\ = & \quad \{\, \text{unfold } \pi\, g \,\} \\ & (\mathit{if}\ (\mathit{empty?}\, xs) \\ & \quad \mathit{empty!} \\ & \quad (\mathit{cons}\, (g\, (\mathit{hd}\, (\pi\, f\, xs))) \\ & \qquad (\pi\, g\, (\mathit{tl}\, (\pi\, f\, xs))))) \\ = & \quad \{\, \text{unfold } \pi\, f \text{ once} \,\} \\ & (\mathit{if}\ (\mathit{empty?}\, xs) \\ & \quad \mathit{empty!} \\ & \quad (\mathit{cons}\, (g\, (\mathit{hd}\, (\mathit{if}\ (\mathit{empty?}\, xs) \\ & \qquad\qquad\qquad \mathit{empty!} \\ & \qquad\qquad\qquad (\mathit{cons}\, (f\, (\mathit{hd}\, xs))\, (\pi\, f\, (\mathit{tl}\, xs)))))) \\ & \qquad (\pi\, g\, (\mathit{tl}\, (\pi\, f\, xs))))) \\ = & \quad \{\, \text{further unfolding, simplification steps} \,\} \\ & (\mathit{if}\ (\mathit{empty?}\, xs) \\ & \quad \mathit{empty!} \\ & \quad (\mathit{cons}\, (g\, (f\, (\mathit{hd}\, xs))) \\ & \qquad \underline{(\pi\, g\, (\pi\, f\, (\mathit{tl}\, xs)))}))) \\ = & \quad \{\, \text{fold marked occurrence of } Q\, f\, g\, (\mathit{tl}\, xs) \,\} \end{array}$$

$$
\begin{array}{l}
(\mathit{if}\ (\mathit{empty?}\ \mathit{xs}) \\
\quad \mathit{empty!} \\
\quad (\mathit{cons}\ (g\ (f\ (\mathit{hd}\ \mathit{xs}))) \\
\qquad\quad (Q\ f\ g\ (\mathit{tl}\ \mathit{xs})))) \ .
\end{array}
$$

Note that the simplified program for Q is equivalent to the combinator query $\pi\,(g{\cdot}f)\,\mathit{xs}$ which, unlike the original query, does not generate any intermediate result.

137 This approach comes with apparent difficulties.

(a) Combinators are encoded by recursive LISP forms that intertwine control structure *and* action. Consequently, simplification rules are not only concerned with the optimization of stream operations like, *e. g.*,

$$(\mathit{tl}\ (\mathit{cons}\ e\ e')) \quad \rightsquigarrow \quad e'$$

but also modify control structure as in

$$(\mathit{if}\ p\ e\ (\mathit{if}\ p\ e'\ e'')) \quad \rightsquigarrow \quad (\mathit{if}\ p\ e\ e'') \ .$$

The resulting derivations tend to be long, leading to significant transformation effort [44]. We could clearly benefit from a separation of control and action.

(b) As an instance of Burstall and Darlington's general *unfold-transform-fold* program transformation strategy [21], the approach suffers from the need for a *fold* step. The transformation algorithm needs a "memory" to spot previously seen expressions in order to be able to successfully "*tie the knot*". We also have to find an upper bound on the number of unfolding steps to perform. Otherwise, unfolding might never stop for the cases in which we fail to spot expressions to recur. The resulting complexity of the approach has prevented its inclusion into actual query optimizers.

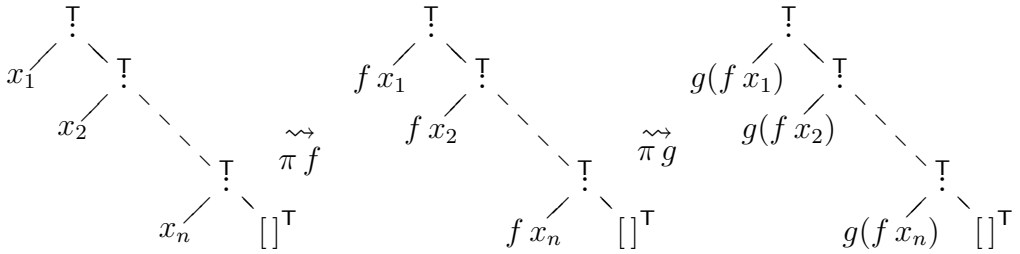
6.2 Cheap Deforestation

138 The catamorphic query discipline provides us with handles to rather elegantly approach these difficulties. First, note that the catamorphism $(\!|e\,\triangledown\otimes|\!)$ separates action (the target algebra $e\,\triangledown\otimes$) from control in that $(\!|\cdot|\!)$ abstracts from the primitive recursion scheme Cata-Ins-Rep. Second, as $(\!|\cdot|\!)$ is the *only* recursive form occurring, we do not need the full power of a general *unfold-fold* transformation strategy.

A general theme of this text recurs once again: a restrictive discipline of expression can lead to deeper insights.

139 The following continues the train of thought we have started in Section 2.6 on spine transformer fusion. To see that fusion is tightly connected with the derivation of streaming programs let us take up the π combinator as an example one last time.

An illustration of the application of query $Q = (\pi\, g) \cdot (\pi\, f)$ to $xs = [x_1, \ldots, x_n]^{\mathsf{T}}$ gives the obvious sequence of spine transformations (recall from PROJECT that $\pi\, f = \mathsf{T} f$):



A streaming program for Q would skip the intermediate step but rather directly emit the spine on the right-hand side. Reducing the length of a spine transformer chain reduces the number of necessary spine walks and thus the number of intermediate results to handle. Complete fusion ultimately leads to real stream-based execution.

In the example above, we experience no particular difficulty to derive the streaming program as we only have to appeal to the functoriality hidden in π:

$$
\begin{aligned}
& (\pi\, g) \cdot (\pi\, f) \\
= \quad & \{\, \textsc{Project} \,\} \\
& \mathsf{T} g \cdot \mathsf{T} f \\
= \quad & \{\, \textsc{Functor} \,\} \\
& \mathsf{T}(g \cdot f) \\
= \quad & \{\, \textsc{Project} \,\} \\
& \pi\, (g \cdot f)\ .
\end{aligned}
$$

For general catamorphic queries, the following theorem establishes an equivalence which serves as the only but effective fusion tool we have to fall back upon.

140 **Acid Rain Theorem** [112]. The underlying category is **Set**. In the category of F-algebras $\mathbf{Alg}(\mathsf{F})$, once more let $\tau : \mathsf{F}T \to T$ and $\alpha = \mathsf{F}A \to A$ denote the initial and an arbitrary algebra, respectively. Fix a type D. Then it holds for any function $f : \forall C.(\mathsf{F}C \to C) \to D \to C$, that

$$(\!|\alpha|\!)_{\mathsf{F}} \cdot (f\, \tau) \quad = \quad f\, \alpha\ . \qquad \textsc{Acid-Rain}$$

Proof. It is remarkable how *parametricity* [116], *i. e.*, the polymorphic type of a function, provides the proof of Acid-Rain for free: the parametricity theorem for the type of f reads (with $\alpha : \mathsf{F}A \to A$, $\beta : \mathsf{F}B \to B$ and $h : B \to A$ chosen at will)

$$h \cdot \beta = \alpha \cdot \mathsf{F}h \quad \Rightarrow \quad h \cdot (f\,\beta) = f\,\alpha\ .$$

Choosing $\beta = \tau$ and $h = (\!|\alpha|\!)_{\mathsf{F}} : T \to A$ turns the premise of the statement of the parametricity theorem into the characterizing property for catamorphisms Cata which, of course, is met by $(\!|\alpha|\!)_{\mathsf{F}}$. As a consequence, Acid-Rain holds. □

The polymorphic type of $f : \forall C.(\mathsf{F}C \to C) \to D \to C$ gives insight into the workings of f: as f is indifferent to the actual choice of C, we can be sure that f builds its result of type C solely by application of the constructors for C (which f receives as its first argument).

Law Acid-Rain then validates the intuition that if we manufacture a value using the constructors of τ (this is what $f\,\tau$ does) and subsequently replace τ's constructors by those of α (this is the action of $(\!|\alpha|\!)_{\mathsf{F}}$), we may rather directly construct the result by using the constructors of α from the start. This way we avoid the computation of the intermediate result of type T—which is what fusion is all about.

141 The functional programming community has devised a family of program transformation techniques, referred to as *deforestation* transformations, which aim to eliminate the production of intermediate (algebraic, and thus tree-shaped) data structures. Wadler characterized a syntactic program class, the *treeless programs*, for which a proof of the guaranteed termination of a *unfold-fold*-based deforestation algorithm could be found [118]. Since then, the criteria which identify the programs that are subject to deforestation have been relaxed [82], but the inherent problems of controlling the *unfold-fold* steps remain.

On the contrary, Takano and Meijer's Acid-Rain theorem exploits a rather fixed producer ($f\,\tau$) and consumer ($(\!|\alpha|\!)_{\mathsf{F}}$) scenario to establish a *cheap* one-step transformation that fuses adjacent consumers and producers. Acid-Rain constitutes a categorical generalization of earlier work on the cheap (or shortcut) deforestation of *listful* programs (in which *foldr* takes the role of the list consumer, see Paragraph 28) [50, 51] and the deforestation of programs over mutually recursive sum-of-product types [37, 74].

142 There are three major driving forces behind this chapter.

(a) In continuation from Paragraph 98, pick up the intriguing idea of a purely spine transformer-based query engine (which means to essentially ignore the material on combinator algebras of Chapter 4). Query deforestation, applied after the syntactic mapping $\mathscr{M} \cdot \mathscr{Q}$ can thus lead to an (at least moderately) efficient, purely transformation-based, implementation of a complex value database engine [59, 60].

(b) We have seen that the combinator pattern matching process of Chapter 4 can fail in that monad comprehension-based parts of a query survive combinator introduction. Comprehension desugarer $\mathscr{M}$ then acts as a fallback which emits catamorphisms, *i. e.*, nested loops, for these query parts. Subsequent deforestation can further optimize these subqueries through reduction of the loop nesting depth. In this sense, deforestation may be perceived as an *a posteriori execution plan optimization*, to be performed after algebraic query rewriting.

(c) Unlike the query engine level techniques (iterator models, suspended evaluation of a stream's tail) which realize streaming *dynamically* at query runtime, deforestation derives streaming plans *statically*, *i. e.*, at query compile time. A deforestation step via law Acid-Rain cancels a spine transformer pair so that formerly separate query parts become adjacent. This, in turn, can reveal previously hidden opportunities for further optimizations. In this sense, contrary to point (b), we can regard deforestation also as a combinator level and thus algebraic optimization tool.

To enable deforestation as a combinator level optimization, as suggested in (c), means to allow the optimizer to desugar (selected) combinator definitions (Definition 103) via mapping $\mathscr{M}$ in order to unbox their catamorphic implementations. Deforestation will especially pay off for those combinators the query engine is likely to implement through simple looping anyway, *e. g.*, π and α. Let us close this chapter with two related examples.

143 Example. Under the proviso $xs : \mathsf{T}\mathit{Num}$ with $\mathsf{T} \neq \mathsf{set}$, naive evaluation of the following combinator query obviously wastes work:

$$((\alpha^{\mathsf{sum}}\, K_1) \cdot (\pi\ g) \cdot (\pi\ f))\, xs$$

(note that $\mathscr{Q}$ yields the aggregate $\alpha^{\mathsf{sum}}\, K_1 : \mathsf{T}\mathit{Num} \to \mathit{Num}$ as a translation of OQL's `count` clause). Since the query effectively counts the nodes of xs' spine only, there is no point in mapping g or f over xs at all. The π combinators could be discarded.

Note that Graefe's iterator discipline yields a streaming program for the above query but the resulting plan will nevertheless apply f and g to all elements that pass the π combinators.

At query runtime, the suspension model will create the closure $\ulcorner g, \ulcorner f, x \urcorner \urcorner$ but will not force it provided that combinator K_x has a non-strict implementation.

Deforestation, however, realizes this optimization statically:

$$
\begin{aligned}
& (\alpha^{\mathsf{sum}}\, K_1) \cdot (\pi\ g) \cdot (\pi\ f) \\
= \quad & \{\, \textsc{Aggregate}, \textsc{Project}, \mathscr{M} \,\} \\
& (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!) \cdot \mathsf{T}g \cdot \mathsf{T}f \\
= \quad & \{\, \textsc{Type-Functor}, \textsc{Acid-Rain} \,\} \\
& (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!) \cdot (\!|([\,]^{\mathsf{T}} \triangledown \overset{\mathsf{T}}{!}) \cdot \mathsf{F}(g \cdot f, id)|\!) \\
= \quad & \{\, \textsc{Acid-Rain} \,\} \\
& (\!|(0 \triangledown (+ \cdot (K_1 \times id))) \cdot \mathsf{F}(g \cdot f, id)|\!) \\
= \quad & \{\, \text{sum} \,\} \\
& (\!|0 \triangledown (+ \cdot ((K_1 \cdot (g \cdot f)) \times id))|\!) \\
= \quad & \{\, K_x \cdot f = K_x \,\} \\
& (\!|0 \triangledown (+ \cdot (K_1 \times id))|\!) \\
= \quad & \{\, \textsc{Aggregate} \,\} \\
& \alpha^{\mathsf{sum}}\, (K_1)\ .
\end{aligned}
$$

The resulting plan does not rely on lazy evaluation semantics at runtime and may be executed by a strict query engine (which is the common case, as the typical imperative implementation languages have strict semantics). ◊

144 Example. Recall that we have adopted a catamorphic representation of the sort operator ς (see SORT and $\mathscr{Q}$-SORT). In interaction with deforestation this sets the scene to reveal and remove superfluous sorting stages in a query plan by means of query rewriting. Traditionally, optimizations of this kind are detected through the inspection of the *physical sort order(s)* of a query expression. As sort orders have no algebraic denotation, their inspection requires extra code outside the core query rewriting system [53, 55].

The catamorphic encoding, however, provides us with enough clue to comprehend ς's action at the algebraic level. Let agg denote the Id-monad associated with the algebra $e \triangledown \otimes$ over carrier E (*i. e.*, $\otimes$ is left-commutative, *cf.* Definition 64). With $\circledcirc : E \times E \to \mathit{Bool}$, we then have that

$$
\begin{aligned}
& (\alpha^{\mathsf{agg}}\, f) \cdot (\varsigma\ \circledcirc) \\
= \quad & \{\, \textsc{Aggregate}, \textsc{Sort} \,\} \\
& (\!|e \triangledown (\otimes \cdot (f \times id))|\!) \cdot (\!|\mathit{nil}^{\mathsf{list}} \triangledown \mathit{ins}^{\mathsf{list}}\, \circledcirc|\!)
\end{aligned}
$$

$$
\begin{array}{ll}
= & \quad \{\, \textsc{Acid-Rain} \,\} \\
& (\!|\, e \triangledown (\otimes \cdot (f \times id)) \,|\!) \\
= & \quad \{\, \textsc{Aggregate} \,\} \\
& \alpha^{\mathsf{agg}}\, f
\end{array}
$$

(a similar calculation asserts the idempotence of ς: $(\varsigma \circledcirc) \cdot (\varsigma \circledcirc) = \varsigma \circledcirc$). $\Diamond$

Chapter 7

Finalization

145 Datatypes. The consistent comprehension of queries as mappings between datatypes, *i. e.*, their algebras of value constructors, has been our guide rail throughout the entire text.

Once this was all set, almost everything else was a consequence of the datatype centric approach: if a query f is a mapping between types A and B, how can we go about and represent f inside a computer or, more specifically, a query engine? Under the proviso that A is finite we could exploit a lookup table to encode f. 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. This is just what the endofunctors encoding the insert representation of objects in category **Set** did for us. The type of f will thus actually be $f : \mathsf{F}A \to B$. Second, for the encoding of f 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 f is a homomorphism (or, for initial A, a catamorphism). This describes the basic understanding of queries in this text already fairly well.

146 Catamorphisms. Much of the conciseness and elegance of this treatment of queries was drawn on the uniform encoding F, the insert representation, of bulk values as well as aggregates and quantifiers. This made level to represent a diversity of query constructs, from parallel application via type functors to sorting (ς), using a single program form, namely the catamorphism combinator $(\!|\cdot|\!)$.

The sorting combinator ς provides a canonical example of our intention behind this approach: at the time of its introduction as a catamorphic insertion sort this may have appeared intriguing or interesting at best, but a weird thing to do in practice (sorting in the context of very large databases calls for external sorting techniques that operate on secondary storage). This,

however, was not the point at that time. Rather, we could benefit from this encoding as it characterized the act of sorting on the algebraic level. Sorting, being just another spine transformer, thus became accessible to algebraic program transformation which is a big win for a query optimizer since it facilitates purely algebraic—*i. e.*, calculational—query rewriting (maintenance of an extra sortedness attribute is superfluous in such a setup).

Last but not least, catamorphisms are a deeply understood program form in the theory of *Constructive Algorithmics*. With the three fusion laws, notably law ACID-RAIN, the work of this community provided input that proved to be most useful in many places.

147 Monads and the Monad Comprehension Calculus. 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 T-monad comprehension $[\![\, f\, x \,|\, x \leftarrow xs \,]\!]^{\mathsf{T}}$ can describe parallel application of f, duplicate elimination, aggregation, or a quantifier ranging over xs, dependent on the actual choice of monad T. This uniformity enabled 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) in Section 3.2.

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.

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. 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.

148 We have no doubt that one can and should further travel down this road.

(a) The catamorphic spine transformers have turned out to be expressive enough to grasp modern declarative user query languages. We could, however, push the limits of expressiveness of this approach through substitution of catamorphisms by *paramorphisms* [83]. Paramorphisms provide a significant generalization of the catamorphism notion: *any* function definable on an initial datatype has a paramorphic equivalent. Let $\tau : \mathsf{F}T \to T$ denote the initial algebra for functor F. Any function $f : T \to A$ can be computed by the paramorphism $outl \cdot (\!| \alpha \vartriangle (\tau \cdot \mathsf{F}\, outr) |\!)$ for a suitably chosen F-algebra $\alpha : \mathsf{F}(A \times T) \to A$. As paramorphisms are particular tupling catamorphisms they enjoy calculational properties similar to those of catamorphisms, *e. g.*, a variant of law Cata-Fusion can be established for paramorphisms, too. Paramorphisms thus provide full expressiveness *and* effective control over program form. This could lay a foundation of the comprehension of full-fledged database programming languages and their optimization.

(b) The monads we have encountered in this text have been induced by the type system and the constructs of the user query language. Of course, this is not principle to the method. We envision a query engine in which further aspects, especially implementation level issues, of query processing are encoded in the monad framework and thus become subject to analysis and transformation just like queries. This could spawn research heading into several directions. As already pointed out in the closing paragraph of Chapter 2, the referentially transparent treatment of stateful computation in the state monad can lead to an optimizable model of database updates and transactions. Certain tree-shaped datatypes are open to a monadic interpretation as well. It is conceivable to develop a deeper understanding of lookups in tree-shaped index structures (think of *B-trees*) and how these lookups could be efficiently interleaved with query execution. Monadic tree datatypes also make level for a comprehension of queries over semi-structured data in the spirit of [17].

(c) Different levels of sophistication are equally expressible in this framework because the query translation process itself is completely compositional. A query compiler can be rapidly prototyped and look as simple as the composition $\mathscr{M} \cdot \mathscr{Q}$. Throughout this text, however, we have repeatedly observed our model to assimilate other advanced approaches to query optimization with relative ease. Its abstract foundation, catamorphisms and monads, make this understanding of queries a lowest common denominator of these approaches, in a sense. This provides an ideal playground in which the interaction of a diversity of optimization techniques may be studied, much like we did in Chapter 5. There have been other optimization techniques knocking on the

door in the course of Chapter 5 and we are positive that we can further improve our comprehension of queries if we let them in.

References

[1] Serge Abiteboul and Paris C. Kanellakis. Query Languages for Complex Object Databases. *ACM SIGACT News*, 21(3):9–18, 1990.

[2] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures*. Foundations of Computing Series. The MIT Press, 1991.

[3] Roland Backhouse. An Exploration of the Bird-Meertens Formalism. Technical Report CS8810, University of Groningen, Department of Mathematics and Computer Science, 1988.

[4] François Bancilhon, Ted Briggs, Setrag Khoshafian, and Patrick Valduriez. FAD, a Powerful and Simple Database Language. In *Proc. of the 13th Int'l Conference on Very Large Data Bases (VLDB)*, pages 97–105, Brighton, England, September 1987.

[5] Colin R. Banger and David B. Skillicorn. Flat Arrays as a Categorical Data Type. Department of Computing and Information Science, Queen's University, Kingston, Canada, November 1992.

[6] Catriel Beeri and Yoram Kornatzky. Algebraic Optimization of Object-Oriented Languages. In Paris C. Kanellakis and Serge Abiteboul, editors, *Proc. of the 3rd Int'l Conference on Database Theory (ICDT)*, number 470 in Lecture Notes in Computer Science (LNCS), pages 72–88, Paris, France, December 1990. Springer Verlag.

[7] 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.

[8] Richard S. Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122–126, 1989.

[9] Richard S. Bird. Functional Algorithm Design. In *Mathematics of Program Construction (MPC)*, number 947 in Lecture Notes in Computer Science (LNCS), pages 2–17. Springer Verlag, 1995.

[10] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall International Series in Computer Science. Prentice Hall Europe, 1997.

[11] Richard S. Bird and Lambert Meertens. Two Excersises Found in a Book on Algorithmics. In Lambert Meertens, editor, *Proc. of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation*, pages 451–457, Bad Tölz, Germany, April 1986. Elsevier Science Publishers B.V. (North Holland).

[12] Peter A. Boncz, Fred Kwakkel, and Martin L. Kersten. High Performance Support for OO Traversals in Monet. In *Proc. of the 14th British National Conference on Databases (BNCOD)*, pages 152–169, Edinburgh, UK, 1996.

[13] 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.

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

[15] François Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 193–204, Portland, Oregon, USA, June 1989.

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

[17] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[18] Peter Buneman, Robert E. Frankel, and Nikhil Rishiyur. An Implementation Technique for Database Query Languages. *ACM Transactions on Database Systems*, 7(2):164–186, June 1982.

[19] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23:87–96, March 1994.

[20] 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.

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

[22] Luca Cardelli. Type Systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[23] 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.

[24] Daniel Chan. *Object-oriented Language Design and Processing*. PhD thesis, Computer Science Department, University of Glasgow, 1994.

[25] Daniel Chan and Philip Trinder. Object Comprehensions: A Query Notation for Object-Oriented Databases. In *Proc. of the 12th British National Conference on Databases (BNCOD)*, pages 55–72, Guildford, UK, 1994.

[26] Damianos Chatziantoniou. *Optimization of Complex Aggregate Queries in Relational Databases*. PhD thesis, Columbia University, June 1997.

[27] 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.

[28] Mitch Cherniack. Form(ers) Over Function(s): The KOLA Reference Manual. Technical report, Department of Computer Science, Brown University, 1996.

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

[30] Mitch Cherniack. *Building Query Optimizers with Combinators*. PhD thesis, Brown University, May 1999.

[31] Mitch Cherniack, Ashok Malhotra, and Stanley B. Zdonik. Experiences with Query Translation: Object Queries meet DB2. Technical Report A149, Department of Computer Science, Brown University Database Group, 1998.

[32] 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.

[33] Mitch Cherniack and Stanley B. 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.

[34] 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)*, pages 286–295, Athens, Greece, August 1997.

[35] 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.

[36] 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.

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

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

[39] Leonidas Fegaras. Optimizing Queries with Object Updates. *Journal of Intelligent Information Systems*, 12(2/3):219–242, 1999. Special Issue on Functional Approach to Intelligent Information Systems.

[40] 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.

[41] Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. Unpublished work, 1998.

[42] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Enschede, 1992.

[43] Maarten M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[44] Johann Christoph Freytag. *Translating Relational Queries into Iterative Programs*. PhD thesis, IBM Almaden Research Center, San Jose, USA, April 1987.

[45] 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.

[46] 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.

[47] 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.

[48] Richard A. Ganski and Harry K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 23–33, San Francisco, USA, 1987.

[49] Jeremy Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming*, 6(4):657–665, 1996.

[50] Andrew J. Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, January 1996.

[51] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, pages 223–232, Copenhagen, Denmark, April 1993.

[52] 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.

[53] 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.

[54] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[55] Goetz Graefe and William J. McKenna. The VOLCANO Optimizer Generator: Extensibility and Efficient Search. In *Proc. IEEE Conference on Data Engineering*, page 209 ff., April 1993.

[56] Malcolm Grant. Homomorphisms and Promotability. In Jan L. A. 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.

[57] Malcolm Grant. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[58] 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.

[59] 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.

[60] 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.

[61] Seymour Hayden and John F. Kennison. *Zermelo-Fraenkel Set Theory*. Merrill, Columbus, Ohio, 1968.

[62] Jonathan M.D. Hill. *Data-Parallel Lazy Functional Programming*. PhD thesis, University of London, Queen Mary and Westfield College, September 1994.

[63] Jonathan M.D. Hill and Keith Clarke. An Introduction to Category Theory Monads and their Relationship to Functional Programming. Technical Report QMW-DCS-681, Department of Computer Science, Queen Mary & Westfield College, August 1994.

[64] C.A.R. Hoare. Notes on an Approach to Category Theory for Computer Scientists. In Manfred Broy, editor, *Constructive Methods in Computing Science*, volume F 55 of *NATO ASI Series*, pages 245–305. Springer Verlag, 1989.

[65] John Hughes and Simon L. Peyton Jones (editors). Haskell 98: A Non-strict, Purely Functional Language. `http://haskell.org/definition/`, February 1999.

[66] Matthias Jarke and Jürgen Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[67] Johan Jeuring. *Theories for Algorithm Calculation*. PhD thesis, University of Utrecht, 1993.

[68] Mark P. Jones and Luc Duponcheel. Composing Monads. Technical report, Yale University, Department of Computer Science, New Haven, Conneticut, USA, December 1993.

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

[70] Heinrich Kleisli. Every Standard Construction is Induced by a Pair of Adjoint Functors. *Proc. of the American Mathematical Society (AMS)*, 16:544–546, 1965.

[71] Joachim Lambek. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift*, 103:151–161, 1968.

[72] Joachim Lambek. Least Fixpoints of Endofunctors of Cartesian Closed Categories. *Mathematical Structures in Computer Science*, 3(2):229–257, 1993.

[73] John Launchbury and Simon L. Peyton Jones. Lazy Functional State Threads. In *Proc. of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, Orlando, Florida, June 1994. ACM Press.

[74] 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.

[75] Kazem Lellahi and Val Tannen. A Calculus for Collections and Aggregates. In *Proc. of the 7th Conference on Category Theory and Computer Science (CTCS)*, number 1290 in Lecture Notes in Computer Science (LNCS), pages 261–280, S. Margherita Ligure, Italy, September 1997. Springer Verlag.

[76] 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.

[77] Leonid Libkin, Rona Machlin, and Limsoon Wong. A Query Language for Multidimensional Arrays: Design, Implementation, and Optimization Techniques. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 228–239, June 1996.

[78] 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.

[79] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Verlag, 1971.

[80] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[81] Ernest G. Manes and Michael A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer Verlag, 1986.

[82] Simon David Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, Department of Computing Science, University of Glasgow, September 1995.

[83] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4: 413–424, 1992.

[84] Erik Meijer, Marten M. 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.

[85] Robert Milner, Mads Tofte, Robert Harper, and David Mac Queen. *The Definition of Standard ML – Revised*. The MIT Press, 1997.

[86] Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proc. of the IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, USA, June 1989.

[87] Eugenio Moggi. Notions of Computations and Monads. *Information and Computation*, 93(1):55–92, 1991.

[88] 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.

[89] Piergiorgio Odifreddi. *Classical Recursion Theory*, volume 125 of *Studies in Logic and the Foundation of Mathematics*. Elsevier Science Publishers B.V. (North Holland), 1989.

[90] Atsushi Ohori. Representing Object Identity in a Pure Functional Language. In Paris C. Kanellakis and Serge Abiteboul, editors, *Proc. of the 3rd Int'l Conference on Database Theory (ICDT)*, number 470 in Lecture Notes in Computer Science (LNCS), pages 41–55, Paris, France, December 1990. Springer Verlag.

[91] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell. `http://haskell.org/definition/haskell-report-1.4.ps.gz`, April 1997.

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

[93] Simon L. Peyton Jones and John Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, December 1995.

[94] Simon L. Peyton Jones and Philip Wadler. Imperative Functional Programming. In *20th ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, Charlotte, North Carolina, January 1993.

[95] György E. Révész. *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

[96] 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.

[97] Holger Riedel and Marc H. Scholl. A Formalization of ODMG Queries. In *Proc. of the 7th Int'l Conference on Data Semantics (DS-7)*, Leysin, Switzerland, October 1997.

[98] David E. Rydeheard and Rod. M. Burstall. *Computational Category Theory*. Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., 1988.

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

[100] 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.

[101] Marc H. Scholl and Torsten Grust. Hybrid Strategies for Query Translation and Optimisation. Technical Report 72/1998, Faculty of Mathematics and Computer Science, Database Research Group, University of Konstanz, October 1998. Also appeared as ESPRIT *Pastel* Research Report RT2R1.

[102] Patricia G. Selinger, Morton M. Astrahan, Donald M. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 23–34, Boston, Massachusetts, USA, 1979.

[103] Mike Spivey. A Categorical Approach to the Theory of Lists. In Jan L. A. 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 399–408, Groningen, The Netherlands, June 1989. Springer Verlag.

[104] Mike Spivey. Category Theory for Functional Programming. Technical Report PRG-TR-7-93, Computing Laboratory, Oxford University, 1993.

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

[106] 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.

[107] 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.

[108] Bharathi Subramanian, Stanley B. Zdonik, Theodore W. Leung, and Scott L. Vandenberg. Ordered Types in the AQUA Data Model. In *Proc. of the 4th Int'l Workshop on Database Programming Languages (DBPL)*, September 1993.

[109] Bharati Subramanian, Theodore W. Leung, Scott L. Vandenberg, and Stanley B. Zdonik. The AQUA Approach to Querying Lists and Trees in Object–Oriented Databases. In *Proc. of the 11th Intl. Conference on Data Engineering*, 1995.

[110] Dan Suciu and Val Breazu-Tannen. A Query Language for NC. In *Proc. of the 13th ACM Symposium on Principles of Databases (PODS)*, pages 167–178, Minneapolis, Minnesota, May 1994.

[111] 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.

[112] Akihiko Takano and Erik Meijer. Shortcut Deforestation in Calculational Form. In *Proc. of the ACM Conference on Functional Programming and Computer Architecture (FPCA)*, pages 306–313, La Jolla, USA, June 1995. ACM Press.

[113] Philip 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.

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

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

[116] Philip Wadler. Theorems for Free! In *Proc. of the 4th Int'l Conference on Functional Programming and Computer Architecture (FPCA)*, London, England, September 1989.

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

[118] Philip Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.

[119] Philip Wadler. The Essence of Functional Programming. In *19th ACM Symposium on Principles of Programming Languages (POPL)*, Albuquerque, New Mexico, January 1992.

[120] Philip Wadler. Monads for Functional Programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science (LNCS). Springer Verlag, 1995.

[121] Philip Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

[122] 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.

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

[124] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, Philadelphia, August 1994.

[125] Limsoon Wong. *The Kleisli/CPL Extensible Query Optimizer—Programmer Guide*. Institute of Systems Science, Singapore, April 1996.

Index