Relational Algebra by Way of Adjunctions

JEREMY GIBBONS, University of Oxford, United Kingdom
FRITZ HENGLEIN, University of Copenhagen, Denmark
RALF HINZE, Universität Kaiserslautern, Germany
NICOLAS WU, University of Bristol, United Kingdom

Bulk types such as sets, bags, and lists are monads, and therefore support a notation for database queries based on comprehensions. This fact is the basis of much work on database query languages. The monadic structure easily explains most of standard relational algebra—specifically, selections and projections—allowing for an elegant mathematical foundation for those aspects of database query language design. Most, but not all: monads do not immediately offer an explanation of relational join or grouping, and hence important foundations for those crucial aspects of relational algebra are missing. The best they can offer is cartesian product followed by selection. Adjunctions come to the rescue: like any monad, bulk types also arise from certain adjunctions; we show that by paying due attention to other important adjunctions, we can elegantly explain the rest of standard relational algebra. In particular, graded monads provide a mathematical foundation for indexing and grouping, which leads directly to an efficient implementation, even of joins.

CCS Concepts: • **Information systems → Relational database query languages**; • **Theory of computation → Categorical semantics**; • **Software and its engineering → Semantics**;

Additional Key Words and Phrases: SQL, comprehension, adjunction, monad, graded monad.

ACM Reference Format:
Jeremy Gibbons, Fritz Henglein, Ralf Hinze, and Nicolas Wu. 2018. Relational Algebra by Way of Adjunctions. *Proc. ACM Program. Lang.* 2, ICFP, Article 86 (September 2018), 35 pages. https://doi.org/10.1145/3236781

1 INTRODUCTION

Consider the following SQL query, which computes the names and amounts outstanding of customers with overdue invoices, by joining separate customer and invoice tables:

```
SELECT name, amount
FROM customers, invoices
WHERE cid = cust AND due < today
```

Standard database techniques [Date 2004] translate this query into applications of relational algebra operations such as projection π, selection σ, and equijoin $\bowtie$:

$$\pi_{name,\, amount}\ (\sigma_{due<today}\ (customers\ {}_{cid}{\bowtie}_{cust}\ invoices))$$

About 25 years ago, several people observed [Buneman et al. 1994; Trinder 1991; Wadler 1992] that bulk types such as sets and bags (sometimes called 'multisets') form monads, and so support a comprehension syntax that provides a convenient notation and clear and well-understood mathematical

Authors' addresses: Jeremy Gibbons, Department of Computer Science, University of Oxford, Oxford, United Kingdom, jeremy.gibbons@ox.ac.uk; Fritz Henglein, DIKU, University of Copenhagen, Copenhagen, Denmark, henglein@diku.dk; Ralf Hinze, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, ralf@cs.uni-kl.de; Nicolas Wu, Department of Computer Science, University of Bristol, Bristol, United Kingdom, nicolas.wu@bristol.ac.uk.

2475-1421/2018/9-ART86
https://doi.org/10.1145/3236781

foundations for database queries. Many modern programming languages provide built-in syntax for such comprehensions, yielding for free an embedded domain-specific language for queries—for example, in Haskell (writing a lowered dot for record field selection):

$$[\,(c.name, i.amount) \mid c \leftarrow customers, i \leftarrow invoices, c.cid == i.cust, i.due < today\,]$$

Comprehensions have been used for many years as a programming construct for collections, dating back at least to Schwartz's SETL from the late 1960s [Schwartz et al. 1986] and Darlington's NPL from the 1970s [Darlington 1975]. They are the basis of queries in the Kleisli [Wong 2000], LINQ [Meijer 2007; Syme 2006], and Links [Cheney et al. 2013; Cooper et al. 2006] database languages, and also feature in proposals for query notations in OQL [Grust and Scholl 1999], XQuery [Fernandez et al. 2001], and MapReduce-style data analysis [Alexandrov et al. 2015].

However, there is a catch. Monads and monad comprehensions provide no direct way to express an equijoin; this can only be expressed indirectly, as a cartesian product followed by a selection. This becomes clear if we desugar the Haskell comprehension notation, yielding something equivalent to the following:

$$\begin{array}{l}
fmap\ (\lambda(c, i) \rightarrow (c.name, i.amount))\ (\\
\quad filter\ (\lambda(c, i) \rightarrow i.due < today)\ (\\
\quad\quad filter\ (\lambda(c, i) \rightarrow c.cid == i.cust)\ (\\
\quad\quad\quad cp\ (customers, invoices))))
\end{array}$$

This amounts to a very inefficient query plan, computing the full cartesian product (via *cp*) of the *customers* and *invoices* tables before immediately discarding the unmatching pairs—typically the great majority—via the innermost *filter*. Of course, there is a much better plan, computing the same subset of customer/invoice pairs in a more circumspect manner:

$$\begin{array}{l}
fmap\ (fmap\ name \times fmap\ amount)\ (cod\ (\\
\quad fmap\ (id \times filter\ (\lambda i \rightarrow i.due < today))\ (\\
\quad\quad merge\ (customers\ `indexBy`\ cid, invoices\ `indexBy`\ cust))))
\end{array}$$

Here, *indexBy* groups a collection into an indexed table, and is written as an infix operator; *merge* combines two indexed tables with matching indices; and *cod* retrieves the elements—the 'codomain'—from an indexed table. The types are something like the following:

$$\begin{array}{ll}
indexBy & :: Key\ k \Rightarrow \mathsf{Bag}\ v \rightarrow (v \rightarrow k) \rightarrow (k \rightarrow \mathsf{Bag}\ v) \\
merge & :: Key\ k \Rightarrow (k \rightarrow v, k \rightarrow v') \rightarrow (k \rightarrow (v, v')) \\
cod & :: Key\ k \Rightarrow (k \rightarrow \mathsf{Bag}\ v) \rightarrow \mathsf{Bag}\ v
\end{array}$$

parametric in the type v of values, but with a qualification *Key* on the type k of keys—only certain types are suitable as keys—so that k-indexed tables of v-values are represented by the type $k \rightarrow \mathsf{Bag}\ v$. (We will explain the details later in the paper.)

This alternative program yields a result of type $\mathsf{Bag}\ (\mathsf{Bag}\ Name \times \mathsf{Bag}\ Amount)$, and admits a straightforward linear-time implementation. (Going one step further and flattening the result to a $\mathsf{Bag}\ (Name \times Amount)$ may blow up the size of the result from linear in the input size to quadratic, so does not admit a linear-time implementation.) However, this program does not directly correspond to anything that can be written solely in the more accessible comprehension syntax.

Do we have to sacrifice the mathematical elegance of comprehension notation on the altar of efficient execution? In this paper, we show how to sidestep such an unfortunate dilemma. Specifically, we reconsider the mathematics that gives rise to the monads of bulk types like sets and bags—*category theory*, the "mathematics of mathematics" [Cheng 2015]. The central notion is that of *adjunction*, the categorical generalisation of Galois connections and "a concept of fundamental logical and mathematical importance that is not captured elsewhere in mathematics" [Awodey

2006, p179]. Adjunctions capture *universal properties*, which embody essentially all the important equivalences that justify query transformations—in particular, query optimizations.

For example, the fact that bags are the *free commutative monoid* on a type of elements is embodied in an adjunction between two categories: the category of commutative monoids and their homomorphisms, and the category of sets and total functions. The equational properties of bag homomorphisms such as maps and filters that underlie many query optimizations all arise from this adjunction.

Of course, we are not the first to argue that adjunctions are a crucial ingredient of programming language design, even in the more specific domain of query languages; for example, Libkin and Wong [1997] present an approach "based on turning universal properties of collections into programming syntax". The novelty in this paper is to show that adjunctions induce not only monads and comprehensions, which explain relational database selections and projections, but also all the other abstract gadgets needed to complete the explanation of the efficient query plan—in particular, of the *grouping and merging* required for linear-time equijoins. One might say that monads give rise to the accessible point-wise comprehension notation that exemplifies the relational calculus, whereas the underlying adjunctions give rise to the more flexible point-free combinators of the relational algebra.

In summary, we revisit the story of how a comprehension-style notation for collection processing falls out of certain adjunctions, and extend that story to explain how equijoins can also fit into the picture without sacrificing efficiency. Specifically, our contributions are:

- elaborating how *adjunctions* underlie collection processing, including the convenient notation based on comprehensions for expressing queries over collections;
- deriving the *equational properties* of collection-processing operations from the adjunction identities;
- extending the operations to support *grouping* into and *merging* of indexed collections, and identifying the relevant adjunctions;
- adapting the *comprehension notation* to accommodate also equijoins, by translation into grouping and merging;
- exploiting *graded monads* as an appropriate abstraction for capturing the *finiteness* required for aggregations;
- justifying a body of *query transformations* that underlie query optimization, by calculation from the equational properties of collection-processing operations.

We do not claim to present any new query transformations; rather, we see it as a strength that we explain existing well-known transformations from a new perspective.

The rest of this paper is structured as follows. Section 2 recalls some fundamental concepts from category theory on which we build, mainly for the purposes of fixing notation. Section 3 builds a simplified version of the relational algebra in which the containers are all finite bags. We then develop a series of richer models of containers, starting with finite maps in Section 4, then moving to indexed tables in Section 5; in Section 6 we prove the correctness of a linear-time implementation of equijoin on indexed tables. We then introduce graded monads in Section 7 as a means of modelling indexed tables with finite domains; this serves as our final model of databases. With the machinery in place, we show in Section 8 how the transformations used for query optimization arise naturally from the theory. We conclude in Section 9. Appendix A (in the online supplementary materials) gives more details of the adjunctions underlying graded monads; Appendix B provides a prototype implementation in Haskell—intended more as an alternative specification of behaviour than something to be executed.

2 CATEGORY THEORY

The mathematical structures we present as a foundation for relational algebra are rooted in category theory, so we present a very quick overview of the relevant definitions. A full tutorial is beyond the scope of this paper, but this section serves to fix notation for those who have some familiarity. Nevertheless, we hope this provides at least some intuition for readers otherwise new to the material. For more details, see any standard textbook [Awodey 2006; Pierce 1991]. Conversely, the categorically adept reader might simply skip this section.

2.1 Categories

A *category* $\mathscr{C}$ consists of a collection of *objects* and a collection of *arrows*; each arrow $f : A \to B$ goes from a *source* object A to a *target* object B. For each object A there is an identity arrow $id\ A : A \to A$. Adjacent arrows $f : A \to B$ and $g : B \to C$ compose to form $g \cdot f : A \to C$. Composition is associative, with appropriate identity arrows as neutral elements. The *hom-set* $\mathscr{C}(A, B)$ consists of the arrows in $\mathscr{C}$ from A to B. An *initial* object 0 has a unique arrow from it to any other object. Dually, a *final* object 1 has a unique arrow to it from any other.

Think of the objects as 'types' and the arrows as 'functions' or 'programs' from one type to another. A fundamental instance is the category **Set**, in which the objects are sets and the arrows total functions; the initial object $0 = \{\,\}$ is the empty set, and any singleton set $1 = \{()\}$ is final. But we will also consider instances with more structured objects (such as ordered sets, or monoids); then the arrows are typically structure-preserving mappings (in these cases, monotonic functions and monoid homomorphisms, respectively). For instance, the category **CMon** in Section 3 has commutative monoids M as objects, represented by triples $(M, \epsilon, \otimes)$ where M is a set and the operation $\otimes : M \times M \to M$ is associative and commutative with neutral element $\epsilon : M$. Arrows $h : (M_1, \epsilon_1, \otimes_1) \to (M_2, \epsilon_2, \otimes_2)$ in this category are homomorphisms: functions $h : M_1 \to M_2$ between the underlying sets satisfying $h\ \epsilon_1 = \epsilon_2$ and $h\ (x \otimes_1 y) = h\ x \otimes_2 h\ y$. The underlying set of a commutative monoid is called its *carrier*, and when the context is clear is often used to refer to the whole structure, leaving the operation and neutral element implicit.

2.2 Functors

Categories are themselves structured objects; *functors* are the structure-preserving mappings between categories. Functor $\mathsf{F} : \mathscr{C} \to \mathscr{D}$ from category $\mathscr{C}$ to category $\mathscr{D}$ maps the objects and arrows of $\mathscr{C}$ to objects and arrows of $\mathscr{D}$, respecting the structure by preserving sources and targets ($\mathsf{F}\ f : \mathsf{F}\ A \to \mathsf{F}\ B$ when $f : A \to B$) and identities and composition ($\mathsf{F}\ (id\ A) = id\ (\mathsf{F}\ A)$ and $\mathsf{F}\ (g \cdot f) = \mathsf{F}\ g \cdot \mathsf{F}\ f$). There is an identity functor Id, and functors compose $\mathsf{G} \circ \mathsf{F}$; so categories and functors themselves form a category **Cat**. For example, functor $\mathsf{U} : \mathbf{CMon} \to \mathbf{Set}$ maps a monoid to its carrier, and a monoid homomorphism to the underlying function between carriers. When $\mathscr{C}$ and $\mathscr{D}$ coincide, we call F an *endofunctor*. In **Set**, endofunctors can be seen as parametric type constructors such as Bag, which maps an object A (a set of elements) to the object Bag A (the set of finite bags of those elements), and an arrow $f : A \to B$ to the arrow Bag f : Bag $A \to$ Bag B that applies f to each element of a bag. It is an instructive exercise to verify that Bag indeed respects structure.

2.3 Natural Transformations

Given two functors $\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ between the same two categories, a *natural transformation* $\phi : \mathsf{F} \dot{\to} \mathsf{G}$ is a family of arrows $\phi\ A : \mathsf{F}\ A \to \mathsf{G}\ A$ in $\mathscr{D}$, one for each object A in $\mathscr{C}$, respecting the arrow structure of $\mathscr{C}$, in the sense that $\mathsf{G}\ f \cdot \phi\ A = \phi\ B \cdot \mathsf{F}\ f$ for each $f : A \to B$ (see Figure 1(a)). A natural transformation between endofunctors in **Set** is a polymorphic function; for example,

$$\begin{tikzcd}
\mathsf{F}\ A \arrow[r, "\phi\ A"] \arrow[d, "\mathsf{F}\ f"'] & \mathsf{G}\ A \arrow[d, "\mathsf{G}\ f"] \\
\mathsf{F}\ B \arrow[r, "\phi\ B"'] & \mathsf{G}\ B
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\mathsf{F}\ A \arrow[r, "\mathsf{F}\ h"] \arrow[d, "f"'] & \mathsf{F}\ B \arrow[d, "g"] \\
A \arrow[r, "h"'] & B
\end{tikzcd}$$

(b)

Fig. 1. (a) natural transformation, (b) algebra homomorphism

$\{\text{-}\} : \mathsf{Bag} \mathrel{\dot{\to}} \mathsf{Set}$ is the polymorphic family of functions, one per element type, flattening a bag to its underlying set by forgetting multiplicity of elements.

Natural transformations can be composed. Given the functors $\mathsf{E}, \mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$, and natural transformations $\psi : \mathsf{E} \mathrel{\dot{\to}} \mathsf{F}$ and $\phi : \mathsf{F} \mathrel{\dot{\to}} \mathsf{G}$, the *vertical composition* $\phi \cdot \psi : \mathsf{E} \mathrel{\dot{\to}} \mathsf{G}$ has components $(\phi \cdot \psi)\ A = \phi\ A \cdot \psi\ A$. Given the functors $\mathsf{F}, \mathsf{G} : \mathscr{C} \to \mathscr{D}$ and $\mathsf{H}, \mathsf{K} : \mathscr{D} \to \mathscr{E}$, and the natural transformations $\phi : \mathsf{F} \mathrel{\dot{\to}} \mathsf{G}$ and $\gamma : \mathsf{H} \mathrel{\dot{\to}} \mathsf{K}$, the *horizontal composition* $\gamma\ \phi : \mathsf{H} \circ \mathsf{F} \mathrel{\dot{\to}} \mathsf{K} \circ \mathsf{G}$ has components $(\gamma\ \phi)\ A = \gamma\ (\mathsf{G}\ A) \cdot \mathsf{H}\ (\phi\ A) = \mathsf{K}\ (\phi\ A) \cdot \gamma\ (\mathsf{F}\ A)$. It is convenient to mix these natural transformations and functors: the natural transformation $\mathsf{H}\ \phi$ is given by $(\mathsf{H}\ \phi)\ A = \mathsf{H}\ (\phi\ A)$, and $\gamma\ \mathsf{F}$ is given by $(\gamma\ \mathsf{F})\ A = \gamma\ (\mathsf{F}\ A)$.

2.4 Adjunctions

Functors $\mathsf{L} : \mathscr{D} \to \mathscr{C}$ and $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ form an *adjunction* $\mathsf{L} \dashv \mathsf{R} : \mathscr{D} \to \mathscr{C}$ if the arrows $\mathsf{L}\ A \to B$ in $\mathscr{C}$ are in natural 1-to-1 correspondence with the arrows $A \to \mathsf{R}\ B$:

$$\lfloor\text{-}\rfloor : \mathscr{C}(\mathsf{L}\ A, B) \cong \mathscr{D}(A, \mathsf{R}\ B) : \lceil\text{-}\rceil$$

This is 'natural' in the sense that there are natural transformations $\eta : \mathsf{Id} \mathrel{\dot{\to}} \mathsf{R} \circ \mathsf{L}$, defined so that $\eta\ A = \lfloor id\ (\mathsf{L}\ A)\rfloor$ is the correspondent in $\mathscr{D}$ of $id\ (\mathsf{L}\ A)$ in $\mathscr{C}$, and symmetrically $\epsilon : \mathsf{L} \circ \mathsf{R} \mathrel{\dot{\to}} \mathsf{Id}$ with $\epsilon\ B = \lceil id\ (\mathsf{R}\ B)\rceil$ the correspondent in $\mathscr{C}$ of $id\ (\mathsf{R}\ B)$ in $\mathscr{D}$, subject to the so-called triangle identities $\mathsf{R}\ \epsilon \cdot \eta\ \mathsf{R} = id$ and $\epsilon\ \mathsf{L} \cdot \mathsf{L}\ \eta = id$. We call L the 'left adjoint', R the 'right adjoint', η the 'unit', and ϵ the 'counit'. In diagrams (such as in Figure 2), it is often convenient to draw L and R as parallel horizontal arrows in opposite directions; then it is conventional to rotate the $\dashv$ symbol by 90 degrees.

Adjunctions really do model natural correspondences between categories, generalizing Galois connections from lattice theory; they are generally embedding–retraction pairs. For example, the product category $\mathbf{Set}^2$ has pairs (A, B) of $\mathbf{Set}$-objects A and B as objects and pairs $(f, g) : (A, B) \to (C, D)$ of $\mathbf{Set}$-arrows $f : A \to C$ and $g : B \to D$ as arrows, and the diagonal functor $\Delta : \mathbf{Set} \to \mathbf{Set}^2$ takes object A and arrow f in $\mathbf{Set}$ to (A, A) and (f, f) in $\mathbf{Set}^2$; then the adjunction $\Delta \dashv \times$ between $\mathbf{Set}^2$ and $\mathbf{Set}$ in Section 3 captures the equivalence between arrows $\Delta A \to (B, C)$ in $\mathbf{Set}^2$ (that is, pairs of functions with a common domain) and arrows $A \to B \times C$ in $\mathbf{Set}$ (that is, functions constructing pairs)—an embedding of the pair-forming functions within the function pairs. A more important example for us is the adjunction $\mathsf{Free} \dashv \mathsf{U}$ between $\mathbf{CMon}$ and $\mathbf{Set}$ in Section 3, capturing the embedding of a set A as the 'free' commutative monoid $\mathsf{Free}\ A = (\mathsf{Bag}\ A, \emptyset, \uplus)$, and the retraction $\mathsf{U}\ (M, \epsilon, \otimes) = M$ of a commutative monoid to its carrier. The 1-to-1 correspondence in this case is between bag aggregations $\mathsf{Free}\ A \to (M, \epsilon, \otimes)$ which are bag homomorphisms to some commutative monoid, and plain functions $A \to \mathsf{U}\ M$ from bag elements into the commutative monoid. The triangle identities specialize to give all the important properties of aggregations.

2.5 Monads

A *monad* (T, η, μ) consists of an endofunctor T equipped with two natural transformations, unit $\eta : \mathsf{Id} \dot{\to} \mathsf{T}$ and multiplication $\mu : \mathsf{T} \circ \mathsf{T} \dot{\to} \mathsf{T}$, satisfying three identities: $\mu \cdot \mathsf{T}\ \eta = id$, $\mu \cdot \eta\ \mathsf{T} = id$, and $\mu \cdot \mathsf{T}\ \mu = \mu \cdot \mu\ \mathsf{T}$. The composition of adjoint functors $\mathsf{R} \circ \mathsf{L}$ always forms a monad; the unit η of the monad is the unit of the adjunction, and the multiplication is constructed from the counit as $\mu = \mathsf{R}\ \epsilon\ \mathsf{L}$.

Bulk types such as Bag are monads; η constructs a singleton bag, and μ flattens a bag of bags to a bag. The Bag monad arises from the adjunction $\mathsf{Free} \dashv \mathsf{U}$ discussed above.

2.6 Algebras

An *algebra* for endofunctor F is a pair $(A, f : \mathsf{F}\ A \to A)$, with A called the *carrier* of the algebra and f the *action*. A *homomorphism* h between F-algebras (A, f) and (B, g) is an arrow $h : A \to B$ in the underlying category that is coherent with the actions: $h \cdot f = g \cdot \mathsf{F}\ h$ (see Figure 1(b)). The identity arrow is a homomorphism, and homomorphisms compose; so F-algebras and their homomorphisms form a category $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$.

There is a *forgetful* functor $\mathsf{U}^{\mathsf{F}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ discarding the algebraic structure: $\mathsf{U}^{\mathsf{F}}\ (A, f) = A$ and $\mathsf{U}^{\mathsf{F}}\ (h : (A, f) \to (B, g)) = h : A \to B$. Under mild conditions on F and $\mathscr{C}$ (for example, for polynomial functors F on **Set**, that is, sums of products), the forgetful functor U^{F} has a left adjoint $\mathsf{Free}^{\mathsf{F}} \dashv \mathsf{U}^{\mathsf{F}}$ (see Figure 2(h)). The functor $\mathsf{Free}^{\mathsf{F}}$ maps an object A to the *free* F-algebra $\mathsf{Free}^{\mathsf{F}}\ A$, which can be seen as the terms built from signature F on variables A. Similar to the closely related adjunction $\mathsf{Free} \dashv \mathsf{U}$ between **CMon** and **Set** above, the adjunction $\mathsf{Free}^{\mathsf{F}} \dashv \mathsf{U}^{\mathsf{F}}$ states the 1-to-1 correspondence between the homomorphisms $\mathsf{Free}^{\mathsf{F}}\ A \to (B, g)$ on the free algebra and the mappings $A \to \mathsf{U}^{\mathsf{F}}\ (B, g) = A \to B$ on their variables. In the special case that $A = 0$, the initial object in the underlying category, $\mathsf{Free}^{\mathsf{F}}$ yields the initial F-algebra $(\mu\ \mathsf{F}, In)$, and the unique homomorphism to a target algebra is called a *fold*. The adjunction is crucial for programming with algebraic datatypes: these are interpreted as initial algebras, so there is a unique (the empty) mapping on the set of variables, and therefore a homomorphism is completely determined by specifying a target algebra.

An *algebra for a monad* (T, η, μ) (or 'Eilenberg–Moore algebra') is an algebra $(A, f : \mathsf{T}\ A \to A)$ for T considered as an endofunctor, which additionally respects the monadic structure: $f \cdot \eta\ A = id\ A$ and $f \cdot \mu\ A = f \cdot \mathsf{T}\ f$. Monad algebras model well-behaved mappings from structured objects; in particular, algebras for a bulk type such as Bag are the aggregation operations, such as *sum* and *maximum*.

3 RELATIONAL ALGEBRA

By exploiting a small number of fundamental adjunctions between familiar categories of structured objects, we can build up a collection of mathematical operators that form a basis for relational algebra. We start in this section with *finite bags*; in Section 4, we introduce the *pointed sets* we need for *finite maps*; in Section 5, we combine these to yield *indexed tables*; in Section 6, we define grouping to indexed tables, and prove that equijoin implemented using grouping agrees with the naive nested-loop version; finally, in Section 7, we present a modest generalisation of monads called *grading* in order to accommodate finiteness. It turns out that a little category theory goes a surprisingly long way: everything we need arises from the handful of adjunctions in Figure 2. This is intellectually satisfying. But more than that: the adjunctions not only give us the appropriate structures, they also provide precisely the equational laws needed for typical query optimization, as we discuss in Section 8.

$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift right=1.5ex, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbf{Set}^0 \arrow[l, shift right=1.5ex, "0"'] \arrow[r, shift right=1.5ex, "1"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\Delta"']\end{tikzcd}$$

$$\begin{aligned} \Delta &: A \mapsto * \\ 0 &: * \mapsto \{\,\} \\ 1 &: * \mapsto \{*\} \end{aligned}$$

(a)

$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift right=1.5ex, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbf{Set}^2 \arrow[l, shift right=1.5ex, "+"'] \arrow[r, shift right=1.5ex, "\times"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\Delta"']\end{tikzcd}$$

$$\begin{aligned} \Delta &: A &&\mapsto (A, A) \\ + &: (A, B) &&\mapsto A + B \\ \times &: (A, B) &&\mapsto A \times B \end{aligned}$$

(b)

$$\begin{tikzcd}\mathbf{Set}_* \arrow[r, shift right=1.5ex, "\mathsf{U}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\mathsf{L}"']\end{tikzcd}$$

$$\begin{aligned} \mathsf{L} &: A &&\mapsto (A \cup \{*\}, *) \\ \mathsf{U} &: (A, a) &&\mapsto A \end{aligned}$$

(c)

$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift right=1.5ex, "(\text{-})^P"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\text{-}\times P"']\end{tikzcd}$$

$$\begin{aligned} (\text{-} \times P) &: (f : A \to B) \mapsto (f \times P : A \times P \to B \times P) \\ (\text{-})^P &: (f : A \to B) \mapsto ((f\ \cdot) : A^P \to B^P) \end{aligned}$$

(d)

$$\begin{tikzcd}\mathbf{Rel} \arrow[r, shift right=1.5ex, "P\times\text{-}"'] \arrow[r, phantom, "\perp"] & \mathbf{Rel} \arrow[l, shift right=1.5ex, "\text{-}\times P"']\end{tikzcd}$$

$$\begin{aligned} (\text{-} \times P) &: (R : A \leftrightarrow B) \mapsto (R \times P : A \times P \leftrightarrow B \times P) \\ (P \times \text{-}) &: (R : A \leftrightarrow B) \mapsto (P \times R : P \times A \leftrightarrow P \times B) \end{aligned}$$

(e)

$$\begin{tikzcd}\mathbf{CMon} \arrow[r, shift right=1.5ex, "\mathsf{U}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\mathsf{Free}"']\end{tikzcd}$$

$$\begin{aligned} \mathsf{Free} &: A \mapsto (\mathsf{Bag}\ A, \emptyset, \uplus) \\ \mathsf{U} &: (M, \epsilon, \otimes) \mapsto M \end{aligned}$$

(f)

$$\begin{tikzcd}\mathbf{Rel} \arrow[r, shift right=1.5ex, "\mathsf{E}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\mathsf{J}"']\end{tikzcd}$$

$$\begin{aligned} \mathsf{J} &: (f : A \to B) \mapsto \{(a, f\ a) \mid a \in A\} \\ \mathsf{E} &: (\mathsf{R} : A \leftrightarrow B) \mapsto \\ &\quad \lambda X \subseteq A. \{b \in B \mid \exists a \in X\ .\ a\ R\ b\} \end{aligned}$$

(g)

$$\begin{tikzcd}\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \arrow[r, shift right=1.5ex, "\mathsf{U}^{\mathsf{F}}"'] \arrow[r, phantom, "\perp"] & \mathscr{C} \arrow[l, shift right=1.5ex, "\mathsf{Free}^{\mathsf{F}}"']\end{tikzcd}$$

$$\begin{aligned} \mathsf{Free}^{\mathsf{F}} &: A \mapsto \text{“terms of sign. F} \\ &\qquad\quad \text{with vars } A\text{”} \\ \mathsf{U}^{\mathsf{F}} &: (A, f) \mapsto A \end{aligned}$$

(h)

Fig. 2. The adjunctions underlying relational algebra

As a first approximation, a database table is a finite bag of values—a bag rather than a set or a list, as we wish to preserve the multiplicity of elements but ignore any ordering. Preserving multiplicity is important for non-idempotent aggregations, such as summing a collection of numbers:

$$sum\ (fmap\ amount\ (filter\ (\lambda i \to i.due < today)\ invoices))$$

With sets rather than bags as the bulk datatype, the query simply doesn't work: coincidentally duplicate amounts are incorrectly discarded.

The values contained in a relational database are typically records, but we do not wish to limit attention to this particular class of types. The reason is simple: intermediate tables often contain non-record values; for instance, in the example above, the argument to *sum* is a bag of monetary values; and the running example in the introduction yielded a bag of pairs of bags as the final result. The operations of the relational algebra enjoy straightforward implementations in terms of bags—Figure 3 summarizes the notation, with full definitions to follow.

The function *single* takes a single element to a singleton bag, and $\uplus$ is bag union. We use bag brackets as a shorthand to define bags by listing their elements, instead of using *single* and $\uplus$; so $\Lbag a, b, b \Rbag = single\ a \uplus single\ b \uplus single\ b$ is the bag that contains a once and b twice. Even though the bag model of database tables is rather naive, exploring the theory of bags in more depth is still useful: it will pay dividends later when we discuss more sophisticated models.

Most of the operations and their accompanying properties arise out of the adjunction

$$\begin{tikzcd}\mathbf{CMon} \arrow[r, shift right=1.5ex, "\mathsf{U}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift right=1.5ex, "\mathsf{Free}"']\end{tikzcd}$$

table of V values	Bag V	projection π_f	Bag f
empty table	$\emptyset$	selection σ_p	*filter* p
singleton table	*single*	aggregation in monoid M	*reduce* M
union of tables	$\uplus$		
cartesian product of tables	$\times$		
neutral element	$\{\!\{()\}\!\}$		

Fig. 3. Relational algebra operators, implemented in terms of bags

between the category **CMon** of commutative monoids and their homomorphisms and the category **Set** of sets and total functions (Figure 2(f)). Very briefly, the forgetful functor U maps a commutative monoid $\mathsf{M} = (M, \epsilon, \otimes)$ to its carrier set M, and a homomorphism $h : (M_1, \epsilon_1, \otimes_1) \to (M_2, \epsilon_2, \otimes_2)$ to its underlying function $h : M_1 \to M_2$ on the carriers; the algebraic structure is forgotten. Its left adjoint Free takes a set A to Free A = (Bag $A, \emptyset, \uplus$), the so-called *free commutative monoid* on A, and lifts a function f on elements to homomorphism Bag f on bags. Adjointness means that homomorphisms of type Free $A \to \mathsf{M}$ are in 1-to-1 correspondence with functions of type $A \to$ U M, where A is a set and M is a commutative monoid. (To be precise, there is a 1-to-1 correspondence classically; in a constructive setting, one would have to take the usual care over what is meant by 'equals', perhaps assuming decidable equality on A [Carette 2018].) This bijection is natural in A and M. Let us now investigate what this tells us about the operations of relational algebra.

Monoid operations. First of all, we record that (Bag $A, \emptyset, \uplus$) is a commutative monoid. The empty bag and bag union

$$\begin{aligned} \emptyset &: \text{Bag } A \\ (\uplus) &: \text{Bag } A \times \text{Bag } A \to \text{Bag } A \end{aligned}$$

satisfy the required laws:

$$\emptyset \uplus s = s = s \uplus \emptyset \tag{1a}$$

$$(s \uplus t) \uplus u = s \uplus (t \uplus u) \tag{1b}$$

$$s \uplus t = t \uplus s \ . \tag{1c}$$

Projections. The type of finite bags is functorial in the elements, as it can be written as a composition of two functors, Bag = U $\circ$ Free; so mapping over bags satisfies the two functor laws.

$$\text{Bag } \mathit{id} = \mathit{id} \tag{2a}$$

$$\text{Bag } (g \cdot f) = \text{Bag } g \cdot \text{Bag } f \tag{2b}$$

The functor Free maps a function to a homomorphism; Bag inherits this property (since U acts as the identity on arrows).

$$\text{Bag } f\ \emptyset = \emptyset \tag{3a}$$

$$\text{Bag } f\ \{\!\{a\}\!\} = \{\!\{f\ a\}\!\} \tag{3b}$$

$$\text{Bag } f\ (s \uplus t) = \text{Bag } f\ s \uplus \text{Bag } f\ t \tag{3c}$$

These three equations fully determine Bag f. In the special case that the elements of the bag are records, and function f is a record projection (preserving some fields and discarding others), then Bag f can be seen as projecting some columns out of a database table. But the functorial action is more general than this: the bag elements need not be records, and the function f may perform some non-trivial computation on each element rather than merely a projection.

Aggregations. The adjunction $\mathsf{Free} \dashv \mathsf{U}$ captures a 1-to-1 correspondence between homomorphisms of type $\mathsf{Free}\ A \to \mathsf{M}$ in **CMon** and functions of type $A \to \mathsf{U}\ \mathsf{M}$ in **Set**. Setting $\mathsf{M} := \mathsf{Free}\ A$, the correspondent in **Set** of the identity homomorphism $id\ (\mathsf{Free}\ A)$ is

$$single\ A : A \to \mathsf{U}\ (\mathsf{Free}\ A)\ \ ,$$

which maps an element to a singleton bag; *single* is the unit of the adjunction. We also write $single\ A\ a$ as $\wr a \int$. Conversely, setting $A := \mathsf{U}\ \mathsf{M}$, the correspondent in **CMon** of the identity function $id\ (\mathsf{U}\ \mathsf{M})$ is

$$reduce\ \mathsf{M} : \mathsf{Free}\ (\mathsf{U}\ \mathsf{M}) \to \mathsf{M}\ \ ,$$

which is the unique homomorphism from the free commutative monoid on the carrier of M to the commutative monoid M itself that takes the singleton $\wr a \int$ to a; this is the counit of the adjunction. Using the units, we can capture the 1-to-1 correspondence as an equivalence:

$$h = reduce\ \mathsf{M} \cdot \mathsf{Free}\ f \iff \mathsf{U}\ h \cdot single\ A = f\ \ , \tag{4}$$

for all homomorphisms $h : \mathsf{Free}\ A \to \mathsf{M}$ and functions $f : A \to \mathsf{U}\ \mathsf{M}$. In other words, a homomorphism from the free commutative monoid is uniquely determined by its behaviour on singleton bags. The homomorphism h is sometimes called the extension of f.

Turning to the properties, $reduce\ \mathsf{M}$ for monoid $\mathsf{M} = (M, \epsilon, \otimes)$ is a homomorphism:

$$reduce\ \mathsf{M}\ \emptyset = \epsilon \tag{5a}$$

$$reduce\ \mathsf{M}\ \wr a \int = a \tag{5b}$$

$$reduce\ \mathsf{M}\ (s \uplus t) = reduce\ \mathsf{M}\ s \otimes reduce\ \mathsf{M}\ t\ \ . \tag{5c}$$

Furthermore, *single* and *reduce* are natural transformations:

$$\mathsf{U}\ (\mathsf{Free}\ f) \cdot single\ A = single\ B \cdot f \tag{6a}$$

$$h \cdot reduce\ \mathsf{M} = reduce\ \mathsf{N} \cdot \mathsf{Free}\ (\mathsf{U}\ h)\ \ , \tag{6b}$$

for all functions $f : A \to B$ and homomorphisms $h : \mathsf{M} \to \mathsf{N}$. Equation (6a) is the point-free presentation of (3b).

Simple examples of homomorphisms from the free commutative monoid are aggregations such as counting and conjoining.

aggregation	monoid	on singletons
count	$(\mathbb{N}, 0, +)$	$\wr a \int \mapsto 1$
sum	$(\mathbb{Z}, 0, +)$	$\wr a \int \mapsto a$
max	$(\mathbb{Z}, minBound, max)$	$\wr a \int \mapsto a$
min	$(\mathbb{Z}, maxBound, min)$	$\wr a \int \mapsto a$
all	$(\mathbb{B}, True, \wedge)$	$\wr a \int \mapsto a$
any	$(\mathbb{B}, False, \vee)$	$\wr a \int \mapsto a$

The last four operations target monoids that are also idempotent, so these would continue to work if bags were replaced by sets—unlike the first two operations.

Strictly speaking, a homomorphism such as $h = reduce\ (\mathbb{N}, 0, +)$ is an arrow in **CMon**, and we ought to write '$\mathsf{U}\ h$' for the corresponding function in **Set**; but in the remainder of the paper we will often abuse notation by omitting the U.

Selection. A restriction or selection, here called *filter*, is a further example of a homomorphism from the free commutative monoid. A filter can be seen as the intersection of a finite bag with an arbitrary set represented by its $\mathbb{B}$-valued characteristic function.

$$\begin{aligned}&\mathit{filter} : (A \to \mathbb{B}) \to (\mathsf{Bag}\ A \to \mathsf{Bag}\ A)\\ &\mathit{filter}\ p = \mathit{reduce}\ (\mathsf{Free}\ A) \cdot \mathsf{Bag}\ (\mathit{guard}\ p)\\ &\mathit{guard} : (A \to \mathbb{B}) \to (A \to \mathsf{Bag}\ A)\\ &\mathit{guard}\ p\ a = \mathbf{if}\ p\ a\ \mathbf{then}\ \wr a \int\ \mathbf{else}\ \emptyset\end{aligned}$$

In other words, *filter* p is the extension of *guard* p. The latter function can be neatly written using a guarded bag comprehension: $\mathit{guard}\ p = \lambda a \to \wr\, a \mid p\ a \int$, which motivates the choice of name. Turning to the properties, *filter* p is a homomorphism and a dinatural transformation (7d).

$$\mathit{filter}\ p\ \emptyset = \emptyset \tag{7a}$$
$$\mathit{filter}\ p\ \wr a \int = \mathit{guard}\ p\ a \tag{7b}$$
$$\mathit{filter}\ p\ (s \uplus t) = \mathit{filter}\ p\ s \uplus \mathit{filter}\ p\ t \tag{7c}$$
$$\mathit{filter}\ p \cdot \mathsf{Bag}\ f = \mathsf{Bag}\ f \cdot \mathit{filter}\ (p \cdot f) \tag{7d}$$

Monad operations. The functor $\mathsf{Bag} = \mathsf{U} \circ \mathsf{Free}$ is furthermore a monad: the composition of a right adjoint with its left adjoint always yields a monad. The unit η of the monad is *single*; the multiplication μ, which takes a bag of bags to its union, is constructed from the counit by

$$\begin{aligned}&\mathit{union}\ A : \mathsf{Bag}\ (\mathsf{Bag}\ A) \to \mathsf{Bag}\ A\\ &\mathit{union}\ A = \mathsf{U}\ (\mathit{reduce}\ (\mathsf{Free}\ A))\end{aligned}$$

By construction, *single* and *union* satisfy the monad laws:

$$\mathit{union}\ A \cdot \mathsf{Bag}\ (\mathit{single}\ A) = \mathit{id}\ (\mathsf{Bag}\ A) \tag{8a}$$
$$\mathit{union}\ A \cdot \mathit{single}\ (\mathsf{Bag}\ A) = \mathit{id}\ (\mathsf{Bag}\ A) \tag{8b}$$
$$\mathit{union}\ A \cdot \mathsf{Bag}\ (\mathit{union}\ A) = \mathit{union}\ A \cdot \mathit{union}\ (\mathsf{Bag}\ A) \tag{8c}$$

The fact that Bag is a monad justifies the use of bag comprehensions, as these can be desugared into applications of the monad operations (together with a 'zero') [Wadler 1992]:

$$\wr\, e \mid\ \int = \eta\ e \tag{9a}$$
$$\wr\, e \mid b \int = \mathbf{if}\ b\ \mathbf{then}\ \eta\ e\ \mathbf{else}\ \emptyset \tag{9b}$$
$$\wr\, e \mid p \leftarrow m \int = \mathsf{Bag}\ (\lambda p \to e)\ m \tag{9c}$$
$$\wr\, e \mid q, q' \int = \mu\ \wr\, \wr\, e \mid q' \int \mid q \int \tag{9d}$$

In particular, one can show by induction that the functorial action of Bag distributes over the comprehension syntax:

$$\mathsf{Bag}\ f\ \wr\, e \mid q \int = \wr\, f\ e \mid q \int \tag{10}$$

Cartesian product. The unit type 1 and the product bifunctor $\times$ turn the category of sets into a so-called symmetric monoidal category: 1 is the neutral element of $\times$ up to a natural isomorphism, and $\times$ is associative and commutative up to natural isomorphisms. These two structures arise from the two adjunctions (Figure 2(a)(b)).

$$\begin{tikzcd}\mathbf{Set}^0 \arrow[r, bend right, "1"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, bend right, "\Delta"']\end{tikzcd} \qquad \begin{tikzcd}\mathbf{Set}^2 \arrow[r, bend right, "\times"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, bend right, "\Delta"']\end{tikzcd}$$

Very briefly, Δ is overloaded to denote the *diagonal* functor $\mathbf{Set} \to \mathbf{Set}^k$ for any k, taking each object A and arrow f to k copies of the same; arrows in the one-object, one-arrow category $\mathbf{Set}^0$

are in 1-to-1 correspondence with functions to a singleton set (there is precisely one of each), and pairs of functions with a common domain are in 1-to-1 correspondence with functions constructing pairs.

The functor Bag has *strength*, a broadcast operation of type $A \times \mathsf{Bag}\ B \to \mathsf{Bag}\ (A \times B)$. Moreover, Bag can be extended to a *lax monoidal* functor ('lax' because *unit* and $\times$ are one-way mappings rather than isomorphisms):

$$\begin{array}{l} \mathit{unit} : \mathsf{Bag}\ 1 \quad \text{-- really } 1 \to \mathsf{Bag}\ 1 \\ \mathit{unit} = \wr ()\int \\ (\times) : \mathsf{Bag}\ A \times \mathsf{Bag}\ B \to \mathsf{Bag}\ (A \times B) \\ s \times t = \wr (a, b) \mid a \leftarrow s, b \leftarrow t \int \end{array}$$

The unit bag and cartesian product satisfy:

$$s \times \mathit{unit} \cong s \cong \mathit{unit} \times s \tag{11a}$$

$$s \times (t \times u) \cong (s \times t) \times u \tag{11b}$$

$$s \times t \cong t \times s \tag{11c}$$

(the notation is a bit loose: $a \cong b$ denotes equality modulo the respective canonical isomorphism α, that is, $\mathsf{Bag}\ \alpha\ a = b$), so we also say that Bag is a *commutative* or *symmetric* monoidal monad; in particular,

$$\wr (a, b) \mid a \leftarrow s, b \leftarrow t \int = \wr (a, b) \mid b \leftarrow t, a \leftarrow s \int \tag{12}$$

A lax-monoidal functor with strength is also known as an *applicative functor* or *idiom* [McBride and Paterson 2008].

We have introduced the cartesian product of bags mainly for completeness. It is expensive to compute, so should generally be avoided in favour of joins. To do so is difficult with mere bags of values, which are a rather naive model of database tables; to implement joins efficiently, we require *indexed* tables. These are modelled by finite maps, which are based on the notion of a pointed set, to which we turn next.

4 POINTED SETS

A *pointed set* (A, a) is a set A with a distinguished point or base element $a \in A$. We may write null_A or simply *null* for the point of A. Let (A, null_A) and (B, null_B) be pointed sets; a total function $f : A \to B$ is *point-preserving* if $f\ \mathit{null}_A = \mathit{null}_B$. Pointed sets and point-preserving functions form the category $\mathbf{Set}_*$, which is equivalent to the category of sets and partial functions.

Let A be a set, and let (B, null_B) be a pointed set. The *domain* of the function $f : A \to (B, \mathit{null}_B)$ is the set of all arguments that are mapped to non-null_B values:

$$\mathit{dom}\ f = \{ a \in A \mid f\ a \neq \mathit{null}_B \} \tag{13}$$

(In general, a function between pointed sets will have an infinite domain; in Section 4 we focus on finite maps, which by definition have finite domain.)

To add a base point to an arbitrary set, we can use lifting (called Maybe in Haskell, and `option` in ML). Indeed, the Maybe monad (1 + -) arises as the composition $\mathsf{U} \circ \mathsf{L}$, where

$$\begin{tikzcd} \mathbf{Set}_* \arrow[r, bend right, "\mathsf{U}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, bend right, "\mathsf{L}"'] \end{tikzcd}$$

is the adjunction between the category of pointed sets and point-preserving functions and the category of sets and total functions (Figure 2(c)). Very briefly, the forgetful functor U maps (A, a) to A and a point-preserving function to its underlying function. Its left adjoint L lifts a set A

to $(\mathsf{Maybe}\ A, \mathit{Nothing})$ and lifts a function on A to the obvious point-preserving function. The adjunction $\mathsf{L} \dashv \mathsf{U}$ tells us a lot about the structure of $\mathbf{Set}_*$: since left adjoints preserve initial objects, $0 \cong \mathsf{L}\ 0 = (\mathsf{Maybe}\ 0, \mathit{Nothing}) \cong (1, ())$, the pointed set $(1, ())$ is initial in $\mathbf{Set}_*$; it is is also final, as right adjoints preserve final objects. Right adjoints furthermore preserve products, so

$$\begin{aligned} 0 &\cong (1, ()) \\ 1 &\cong (1, ()) \\ (A, \mathit{null}_A) \times (B, \mathit{null}_B) &\cong (A \times B, (\mathit{null}_A, \mathit{null}_B)) \end{aligned}$$

Some types have an obvious point. For example, the empty bag is a canonical choice for bags, and more generally the neutral element for any monoid. Thus we can turn Bag into a functor $\mathsf{Bag}_* : \mathbf{Set} \to \mathbf{Set}_*$:

$$\begin{aligned} \mathsf{Bag}_*\ A &= (\mathsf{Bag}\ A, \emptyset) \\ \mathsf{Bag}_*\ f &= \mathsf{Bag}\ f \end{aligned}$$

Property (3a) shows that $\mathsf{Bag}\ f$ is point-preserving. Indeed, almost all the bag operations preserve points: union $\uplus$ (1a); projection $\mathsf{Bag}\ f$ (3a); selection $\mathit{filter}\ p$ (7a); aggregation $\mathit{reduce}\ \mathsf{M}$ (5a), assuming that the neutral element of a monoid is its point; and cartesian product $\times$, since $\emptyset \times \emptyset = \emptyset$. The only notable exceptions are *single* and hence *unit*. This implies that Bag_* is not a monoidal functor, only a so-called *semi-monoidal* functor between the semi-monoidal categories $(\mathbf{Set}, \times)$ and $(\mathbf{Set}_*, \times)$ ('semi-monoidal' meaning lacking the unit)—this will be important later.

Finite Maps. Let K be a set, and let V be a pointed set. A *map* of type $\mathsf{Map}\ \mathsf{K}\ V$ is a total function from K to V; a *finite map* of type $\mathsf{Map}_*\ \mathsf{K}\ V$ is a map that yields null_V for all but finitely many arguments. Finite maps are also known as key-value maps, association lists, and dictionaries; they are more appropriate for modelling databases than possibly infinite maps are, because the latter cannot in general be aggregated.

$$\mathsf{Map}_*\ \mathsf{K}\ V = \{ s : \mathsf{K} \to V \mid \mathit{dom}\ s \text{ is finite} \}$$

We can turn $\mathsf{Map}_*\ \mathsf{K}$ into an endofunctor over the category $\mathbf{Set}_*$ of pointed sets and point-preserving functions.

$$\begin{aligned} \mathsf{Map}_*\ \mathsf{K}\ V &= (\mathsf{Map}\ \mathsf{K}\ V, \lambda k \to \mathit{null}_V) \\ \mathsf{Map}_*\ \mathsf{K}\ f &= \lambda s \to f \cdot s \end{aligned}$$

The point is the empty map, which sends all its arguments to null_V. The functorial action on arrows is just post-composition, which is point-preserving when f is, as it sends $\lambda k \to \mathit{null}_V$ to $\lambda k \to \mathit{null}_W$ for $f : V \to W$. We record that $\mathsf{Map}_*\ \mathsf{K}$ satisfies the functor laws:

$$\mathsf{Map}_*\ \mathsf{K}\ \mathit{id} = \mathit{id} \tag{14a}$$

$$\mathsf{Map}_*\ \mathsf{K}\ (g \cdot f) = \mathsf{Map}_*\ \mathsf{K}\ g \cdot \mathsf{Map}_*\ \mathsf{K}\ f \tag{14b}$$

It is important to note that $\mathsf{Map}_*\ \mathsf{K}\ f$ does not necessarily preserve the domain of its argument map: it acts as a *filter* if f sends any non-*null* values to *null*. For example,

$$\mathsf{Map}_*\ \mathsf{K}\ (\lambda v \to \textbf{if}\ \mathit{bad}\ v\ \textbf{then}\ \mathit{null}\ \textbf{else}\ v)\ s$$

filters out the 'bad' values in s. The codomain of a finite map is defined

$$\begin{aligned} &\mathit{cod} : \mathsf{Map}_*\ \mathsf{K}\ V \to \mathsf{Bag}_*\ V \\ &\mathit{cod}\ s = \lbag\, s\ k \mid k \leftarrow \mathit{dom}\ s \,\rbag \ . \end{aligned}$$

(Note that *cod* yields a finite bag, whereas *dom* returns a set.)

Laws of exponents. The finite map functor satisfies a number of properties which are commonly known as the 'laws of exponents' (write $\mathsf{Map}_* \; \mathsf{K} \; V$ as V^{K} to see why).

$$\mathsf{Map}_* \; 0 \; V \cong 1 \tag{15a}$$

$$\mathsf{Map}_* \; 1 \; V \cong V \tag{15b}$$

$$\mathsf{Map}_* \; (\mathsf{K}_1 + \mathsf{K}_2) \; V \cong \mathsf{Map}_* \; \mathsf{K}_1 \; V \times \mathsf{Map}_* \; \mathsf{K}_2 \; V : (\triangledown) \tag{15c}$$

$$\mathit{curry} : \mathsf{Map}_* \; (\mathsf{K}_1 \times \mathsf{K}_2) \; V \cong \mathsf{Map}_* \; \mathsf{K}_1 \; (\mathsf{Map}_* \; \mathsf{K}_2 \; V) : \mathit{curry}^\circ \tag{15d}$$

The isomorphisms are natural in the type V of values. For unrestricted functions, they are consequences of the currying adjunction $(\text{-} \times P) \dashv (\text{-})^P$ (Figure 2(d)). Property (15d) follows directly from this adjunction, and Property (15b) follows from currying and $A \times 1 \cong A$, the fact that product is monoidal.

Properties (15a) and (15c) are also a consequence of currying, but indirectly: this adjunction can be used to form yet another one, namely that $(X^{(\text{-})})^{\mathrm{op}} \dashv X^{(\text{-})}$, which swaps the arguments of a binary function. Since $X^{(\text{-})}$ is right adjoint and contravariant, it takes 0 to 1 and + to ×, and the properties follow.

All of these properties inform the way we can use maps in practice. For example, the penultimate isomorphism (15c) expresses that two functions with the same target type can be represented by a single function from the sum (coproduct) of their source types ('case analysis').

$$s \triangledown t = \lambda k \to \mathbf{case}\; k \;\mathbf{of}\; \{\mathit{Inl}\; a \to s\; a; \mathit{Inr}\; b \to t\; b\}$$

The last isomorphism (15d) captures 'currying': a function of two arguments can be treated as a function of the first argument whose values are functions of the second argument.

$$\mathit{curry}^\circ \; s = \lambda(k_1, k_2) \to (s\; k_1)\; k_2$$

We have to show that the isomorphisms are still valid when restricted to finite maps. For (15a) and (15b) this is trivial. The domain of $s \triangledown t$ and $\mathit{curry}^\circ \; s$ are given by

$$\mathit{dom}\; (s \triangledown t) = \{\mathit{Inl}\; a \mid a \leftarrow \mathit{dom}\; s\} \cup \{\mathit{Inr}\; b \mid b \leftarrow \mathit{dom}\; t\}$$

$$\mathit{dom}\; (\mathit{curry}^\circ \; s) = \{(k_1, k_2) \mid k_1 \leftarrow \mathit{dom}\; s, k_2 \leftarrow \mathit{dom}\; (s\; k_1)\}$$

Clearly, $\mathit{dom}\; (s \triangledown t)$ is finite if and only if both $\mathit{dom}\; s$ and $\mathit{dom}\; t$ are [Connelly and Morris 1995, Proposition 1.1]. Likewise, $\mathit{curry}^\circ \; s$ is a finite map if and only if s and each partial application of s is finite [Connelly and Morris 1995, Proposition 1.3]. It remains to verify that the isomorphisms are also point-preserving. For (15a) and (15b) this is again immediate. For (15c) this follows from $\mathit{null} \cdot \mathit{Inl} = \mathit{null}$ and $\mathit{null} \cdot \mathit{Inr} = \mathit{null}$. Finally, for (15d) we calculate

$$\begin{aligned}
& \mathit{curry}^\circ \; \mathit{null} \\
= \quad & \{ \text{ definition of } \mathit{curry}^\circ \; \} \\
& \lambda(k_1, k_2) \to (\mathit{null}\; k_1)\; k_2 \\
= \quad & \{ \text{ definition of } \mathit{null} \text{ for finite maps } \} \\
& \lambda(k_1, k_2) \to \mathit{null}\; k_2 \\
= \quad & \{ \text{ definition of } \mathit{null} \text{ for finite maps } \} \\
& \lambda(k_1, k_2) \to \mathit{null} \\
= \quad & \{ \text{ definition of } \mathit{null} \text{ for finite maps } \} \\
& \mathit{null}
\end{aligned}$$

These isomorphisms are important, as they can be used to implement finite maps generically for finite sums and finite products [Hinze 2000]. Since the isomorphisms are effectively turned into equalities, the mappings back and forth are constant-time operations.

Union. The 'laws of exponents' come in two flavours: the laws above vary the key type and keep the value type fixed; the laws below fix the key type and vary the value type—the unit type 1 is clearly pointed.

$$empty : 1 \cong \mathsf{Map}_* \; \mathsf{K} \; 1 \tag{15e}$$

$$merge : \mathsf{Map}_* \; \mathsf{K} \; V_1 \times \mathsf{Map}_* \; \mathsf{K} \; V_2 \cong \mathsf{Map}_* \; \mathsf{K} \; (V_1 \times V_2) \tag{15f}$$

The last isomorphism is natural in V_1 and V_2. It implements the merge or union of two finite maps.

```
$empty : \mathsf{Map}_* \; \mathsf{K} \; 1$    -- really $1 \to \mathsf{Map}_* \; \mathsf{K} \; 1$
$empty = \lambda k \to ()$
$merge : \mathsf{Map}_* \; \mathsf{K} \; V_1 \times \mathsf{Map}_* \; \mathsf{K} \; V_2 \to \mathsf{Map}_* \; \mathsf{K} \; (V_1 \times V_2)$
$merge \; (s, t) = \lambda k \to (s \; k, t \; k)$
```

Again we have to check that the isomorphisms are still valid when restricted to finite maps.

$$\begin{aligned} dom \; empty &= \{\,\} \\ dom \; (merge \; (s, t)) &= dom \; s \cup dom \; t \end{aligned}$$

The unique arrow to 1 is certainly finite ($empty = null$). Clearly, $merge \; (s, t)$ is a finite map if (and indeed, only if) both s and t are. Furthermore, the isomorphisms are point-preserving. For (15e) this is trivial. For (15f) this follows from $fst \cdot null = null$ and $snd \cdot null = null$.

These two definitions turn $\mathsf{Map}_* \; \mathsf{K}$ into a strong monoidal functor over $\mathbf{Set}_*$. (We use the 'strong' in 'strong monoidal functor' to indicate that the maps $1 \to \mathsf{F} \; 1$ for unit and $\mathsf{F} \; A \times \mathsf{F} \; B \to \mathsf{F} \; (A \times B)$ for product are isomorphisms. To minimize confusion, we use 'lax monoidal functor with strength' to indicate the existence of tensorial strength, a broadcast operation of type $A \times \mathsf{F} \; B \to \mathsf{F} \; (A \times B)$, and avoid the ambiguous term 'strong lax monoidal functor'.) The empty finite map and the merge operation satisfy:

$$merge \; (s, empty) \cong s \cong merge \; (empty, s) \tag{16a}$$

$$merge \; (s, merge \; (t, u)) \cong merge \; (merge \; (s, t), u) \tag{16b}$$

$$merge \; (s, t) \cong merge \; (t, s) \tag{16c}$$

(again, $\cong$ means equality modulo the respective canonical isomorphism). However, $\mathsf{Map}_* \; \mathsf{K}$ is *not* an applicative functor, because it does not support a strength operation of type $A \times \mathsf{Map}_* \; \mathsf{K} \; B \to \mathsf{Map}_* \; \mathsf{K} \; (A \times B)$: the resulting map would in general not be finite. In particular, we cannot define the *pure* operation of applicative functors in terms of the *unit*, ie $pure \; a = \mathsf{Map}_* \; \mathsf{K} \; (\lambda () \to a) \; unit$, because $\lambda () \to a$ is not point-preserving.

5 INDEXED TABLES

An indexed table is a finite map from keys (indices) to finite bags of values.

$$\mathsf{Table} \; \mathsf{K} \; V = \mathsf{Map}_* \; \mathsf{K} \; (\mathsf{Bag}_* \; V)$$

Note that $\mathsf{Table} \; \mathsf{K}$ is just the composition of $\mathsf{Map}_* \; \mathsf{K}$ and Bag_*. Since these are both semi-monoidal functors, and since semi-monoidal functors compose, $\mathsf{Table} \; \mathsf{K} : \mathbf{Set}_* \to \mathbf{Set}_*$ is a semi-monoidal functor too.

Figure 4 summarizes the operations on tables; full definitions follow. We use the monoidal structure of $\mathsf{Map}_* \; \mathsf{K}$ to lift operations on bags to operations on tables: the lifted version of the unary bag function f is $\mathsf{Map}_* \; \mathsf{K} \; f$; the lifted version of the binary bag function g is $\mathsf{Map}_* \; \mathsf{K} \; g \cdot merge$.

K-indexed table of V values	$\mathsf{Table}\ \mathsf{K}\ V$
empty table	empty
singleton table (k, v)	$k \mapsto \wr v \int$
union of tables	$\mathsf{Map}_*\ \mathsf{K}\ (\uplus) \cdot \mathit{merge}$
projection π_f	$\mathsf{Map}_*\ \mathsf{K}\ (\mathsf{Bag}_*\ f)$
selection σ_p	$\mathsf{Map}_*\ \mathsf{K}\ (\mathit{filter}\ p)$
aggregation in monoid M	$\mathsf{Map}_*\ \mathsf{K}\ (\mathit{reduce}\ M)$
natural join	$\mathsf{Map}_*\ \mathsf{K}\ (\times) \cdot \mathit{merge}$

Fig. 4. Relational algebra operators, implemented in terms of indexed tables

Indexing. Relations are in 1-to-1 correspondence with set-valued functions. This correspondence is captured by the adjunction

$$\begin{tikzcd} \mathbf{Rel} \arrow[r, bend right, "\mathsf{E}"'] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, bend right, "\mathsf{J}"'] \end{tikzcd}$$

between the category of sets and relations and the category of sets and total functions (Figure 2(g)). Very briefly, the functor J embeds **Set** into **Rel**, viewing functions as relations. Its right adjoint is the existential image functor E, which sends a set to its powerset and a relation R to the function $\lambda X \to \{ b \in B \mid \exists a \in X\ .\ a\ R\ b \}$. Now, the 1-to-1 correspondence $\mathsf{Set}\ (\mathsf{K} \times V) \cong \mathsf{Map}\ \mathsf{K}\ (\mathsf{Set}\ V)$ remains valid if restricted to finite sets and finite maps, and furthermore if finite sets are replaced by finite bags:

$$\mathit{ix} : \mathsf{Bag}_*\ (\mathsf{K} \times V) \cong \mathsf{Map}_*\ \mathsf{K}\ (\mathsf{Bag}_*\ V) : \mathit{ix}^{\circ} \tag{17}$$

The isomorphism is natural in V. The operation ix can be seen as providing a view or an indexing structure on a table (given as a bag of key-value pairs). It is defined

$$\mathit{ix}\ s = \lambda k \to \wr v \mid (k', v) \leftarrow s, k == k' \int$$

The domain of $\mathit{ix}\ s$ is given by

$$\mathit{dom}\ (\mathit{ix}\ s) = \{ k \mid (k, v) \leftarrow s \}$$

Clearly, ix sends a finite bag to a finite map, which in turn maps each key to a finite bag. The isomorphism is point-preserving, as the empty bag is sent to the empty map. With care and an appropriate choice of representation, ix can be made to run in linear time [Henglein and Hinze 2013].

A useful derived isomorphism is relational currying:

$$\mathsf{Table}\ (\mathsf{K}_1 \times \mathsf{K}_2)\ V \cong \mathsf{Table}\ \mathsf{K}_1\ (\mathsf{K}_2 \times V) \tag{18}$$

which allows us to shift a key into and out of an index. The proof is straightforward:

$$\begin{array}{ll} & \mathsf{Table}\ (\mathsf{K}_1 \times \mathsf{K}_2)\ V \\ = & \quad \{ \text{definition of Table} \} \\ & \mathsf{Map}_*\ (\mathsf{K}_1 \times \mathsf{K}_2)\ (\mathsf{Bag}_*\ V) \\ \cong & \quad \{ \text{currying (15d)} \} \\ & \mathsf{Map}_*\ \mathsf{K}_1\ (\mathsf{Map}_*\ \mathsf{K}_2\ (\mathsf{Bag}_*\ V)) \\ \cong & \quad \{ \text{indexing (17)} \} \\ & \mathsf{Map}_*\ \mathsf{K}_1\ (\mathsf{Bag}_*\ (\mathsf{K}_2 \times V)) \\ = & \quad \{ \text{definition of Table} \} \\ & \mathsf{Table}\ \mathsf{K}_1\ (\mathsf{K}_2 \times V) \end{array}$$

Consequently, the isomorphism is given by the composition $\mathsf{Map}_* \ \mathsf{K}_1 \ ix^\circ \cdot curry$. As an aside, equation (18) is related to the adjunction

$$\begin{tikzcd} \mathbf{Rel} \arrow[r, phantom, "\perp"] \arrow[r, bend right, "P\times\text{-}"'] & \mathbf{Rel} \arrow[l, bend right, "\text{-}\times P"'] \end{tikzcd}$$

(Figure 2(e)), capturing the fact that the relation types $(A \times P) \leftrightarrow B$ and $A \leftrightarrow (P \times B)$ are both essentially the powerset of $A \times P \times B$; this shows that **Rel** is a closed monoidal category—but note that $\times$ is not the categorical product in **Rel**, and that **Rel** is not cartesian closed.

6 EQUIJOIN BY INDEXING

The 'index by' or 'group by' operation

$$\begin{aligned} &indexBy : \mathsf{Bag}_* \ V \times (V \to \mathsf{K}) \to \mathsf{Map}_* \ \mathsf{K} \ (\mathsf{Bag}_* \ V) \\ &s \ \text{`}indexBy\text{`} \ f = ix \ (\mathsf{Bag}_* \ (f \vartriangle id) \ s) \end{aligned}$$

is the crux to an efficient implementation of joins: the equijoin

$$\begin{aligned} &(\bowtie) :: Eq \ k \Rightarrow (a \to k) \to (b \to k) \to \mathsf{Bag} \ a \to \mathsf{Bag} \ b \to \mathsf{Bag} \ (a, b) \\ &x \ {}_f{\bowtie}_g \ y = \wr (a, b) \mid a \leftarrow x, b \leftarrow y, f \ a == g \ b \int \end{aligned}$$

can be computed by indexing x and y by their keys, merging these two indexed tables, performing small cartesian products at each key—thus far, a full outer-join—then extracting the elements from the table:

$$x \ {}_f{\bowtie}_g \ y = elems \ (\mathsf{Map}_* \ (\times) \ (merge \ (x \ \text{`}indexBy\text{`} \ f, y \ \text{`}indexBy\text{`} \ g)))$$

where

$$\begin{aligned} &elems \ : \ \mathsf{Table} \ \mathsf{K} \ V \to \mathsf{Bag}_* \ V \\ &elems = \mu \cdot cod = \mathsf{Bag}_* \ snd \cdot ix^\circ \end{aligned}$$

Of course, a properly engineered database engine should not recompute an index each time a table is accessed: it should memoise the index, and maintain the cached index as the table changes.

In this section, we prove the equality

$$\begin{aligned} &\wr \ (a, b) \mid a \leftarrow x, b \leftarrow y, f \ a == g \ b \int \\ &\quad = elems \ (\mathsf{Map}_* \ \mathsf{K} \ (\times) \ (merge \ (x \ \text{`}indexBy\text{`} \ f, y \ \text{`}indexBy\text{`} \ g))) \end{aligned}$$

demonstrating the correctness of the grouping implementation of the equijoin $x \ {}_f{\bowtie}_g \ y$. For convenience, we recall the types

$$\begin{aligned} &\mathsf{Map}_* \ \mathsf{K} \ V = \mathsf{K} \to V \\ &\mathsf{Table} \ \mathsf{K} \ V \ = \mathsf{Map}_* \ \mathsf{K} \ (\mathsf{Bag}_* \ V) \end{aligned}$$

(Bag_* and Map_* specifically denote *finite* bags and maps, but finiteness is not relevant for this proof), and functions

$$\begin{aligned} &(\times) &&: \mathsf{Bag}_* \ A \times \mathsf{Bag}_* \ B \to \mathsf{Bag}_* \ (A \times B) \\ &filter &&: (A \to \mathbb{B}) \to (\mathsf{Bag}_* \ A \to \mathsf{Bag}_* \ A) \\ &indexBy &&: \mathsf{Bag}_* \ V \times (V \to \mathsf{K}) \to \mathsf{Table} \ \mathsf{K} \ V \\ &merge &&: \mathsf{Map}_* \ \mathsf{K} \ V_1 \times \mathsf{Map}_* \ \mathsf{K} \ V_2 \to \mathsf{Map}_* \ \mathsf{K} \ (V_1 \times V_2) \\ &elems &&: \mathsf{Table} \ \mathsf{K} \ V \to \mathsf{Bag}_* \ V \end{aligned}$$

We use the following lemmas, each of which is easy to prove:

(1) *elems* is a post-inverse of *indexBy*:

$$elems \ (x \ \text{`}indexBy\text{`} \ f) = x$$

(2) lookup in an indexed table is a filter:

$$(x \,\text{`}\mathit{indexBy}\text{`}\, f)\ k = \mathit{filter}\ (\lambda a \to k == f\ a)\ x$$

(3) independent filters promote through cartesian product:

$$\mathit{filter}\ (\lambda(a,b) \to p\ a \wedge q\ b)\ (x \times y) = \mathit{filter}\ p\ x \times \mathit{filter}\ q\ y$$

(4) lookup in merged indices is a pair of lookups:

$$\mathit{merge}\ (x, y)\ k = (x\ k, y\ k)$$

(5) functoriality of $\mathsf{Map}_*\ \mathsf{K}$:

$$\mathsf{Map}_*\ \mathsf{K}\ g\ f = g \cdot f$$

We have

$$\begin{array}{ll}
& \{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\} \\
& \quad = \mathit{elems}\ (\mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g))) \\
\Longleftrightarrow & \quad \{\ \mathit{elems} \text{ and } \mathit{indexBy}\ (1)\ \} \\
& \mathit{elems}\ (\{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\}\,\text{`}\mathit{indexBy}\text{`}(f \cdot \mathit{fst})) \\
& \quad = \mathit{elems}\ (\mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g))) \\
\Longleftarrow & \quad \{\ \text{Leibniz}\ \} \\
& \{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\}\,\text{`}\mathit{indexBy}\text{`}(f \cdot \mathit{fst}) \\
& \quad = \mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g)) \\
\Longleftrightarrow & \quad \{\ \text{function equality: } (f = g) \Leftrightarrow (\forall a\,.\ f\ a = g\ a)\ \} \\
& \forall k\,.\ (\{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\}\,\text{`}\mathit{indexBy}\text{`}(f \cdot \mathit{fst}))\ k \\
& \quad = (\mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g)))\ k
\end{array}$$

so it suffices to prove the latter equation. We have, for arbitrary k:

$$\begin{array}{ll}
& (\{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\}\,\text{`}\mathit{indexBy}\text{`}(f \cdot \mathit{fst}))\ k \\
= & \quad \{\ \text{lookup in table } (2)\ \} \\
& \mathit{filter}\ (\lambda(a,b) \to f\ a == k)\ \{\!\!\{ (a,b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \}\!\!\} \\
= & \quad \{\ \text{comprehensions}\ \} \\
& \mathit{filter}\ (\lambda(a,b) \to f\ a == k)\ (\mathit{filter}\ (\lambda(a,b) \to f\ a == g\ b)\ (x \times y)) \\
= & \quad \{\ \text{filter composition}\ \} \\
& \mathit{filter}\ (\lambda(a,b) \to (f\ a == k) \wedge (f\ a == g\ b))\ (x \times y) \\
= & \quad \{\ \text{equality an equivalence relation}\ \} \\
& \mathit{filter}\ (\lambda(a,b) \to (f\ a == k) \wedge (g\ b == k))\ (x \times y) \\
= & \quad \{\ \text{independent filters } (3)\ \} \\
& \mathit{filter}\ (\lambda a \to f\ a == k)\ x \times \mathit{filter}\ (\lambda b \to g\ b == k)\ y \\
= & \quad \{\ \text{lookup in table } (2)\ \} \\
& (x\,\text{`}\mathit{indexBy}\text{`}\,f)\ k \times (y\,\text{`}\mathit{indexBy}\text{`}\,g)\ k \\
= & \quad \{\ \text{lookup in merged indices } (4)\ \} \\
& (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g)\ k) \\
= & \quad \{\ \text{functoriality of maps } (5)\ \} \\
& (\mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\,\text{`}\mathit{indexBy}\text{`}\,f, y\,\text{`}\mathit{indexBy}\text{`}\,g)))\ k
\end{array}$$

As it stands, this implementation is efficient up to a point. Provided that the predicates used for selection, the functions used for projection, and the equality comparisons used for grouping and joining all take constant time, it takes time linear in the number of tuples in the input to compute a query involving selection, projection, and join. The result is a bag of pairs of bags, as in Section 1; it is important to note that one cannot in general multiply out the pairs of bags to make one big bag of pairs in linear time, because the resulting bag may have more than linearly many tuples.

When the predicates and comparisons are more expensive, such as on unbounded strings, then one needs something like discrimination-based techniques [Henglein and Larsen 2010] in order to retain the linear time guarantee.

7 GRADED MONADS

Let us take stock of where we have got to. We model database tables by some collection type. It is important for that collection type to form a monad, because only monads support the convenient comprehension syntax for queries; we started off simply with Bag V. We elaborated this to tables Map K (Bag_* V), in order to support grouping and indexing for efficient joins. We had to restrict these to finite tables Table K V = Map_* K (Bag_* V), in order also to accommodate aggregation. But sadly, although maps and tables form monads, *finite* maps and tables do not, for the same reason that they are only semi-monoidal and not fully monoidal: the units for the map and table monads, of type $V \rightarrow$ Map K V and $V \rightarrow$ Map K (Bag_* V) respectively, yield constant functions that represent *infinite* tables (at least, when the key type K is unbounded).

Fortunately, there is a way to enjoy both finiteness and the monadic structure. This is to use *graded monads* [Fujii et al. 2016], also sometimes called 'indexed monads' [Orchard et al. 2014] or 'parametric monads' [Katsumata 2014]—that is, to consider Table K as a family of endofunctors for various K that collectively act like a monad, rather than actually being a single monad for all or for a fixed K.

Formally, an M-graded monad (T, η, μ) over a monoid $(M, \epsilon, \otimes)$ is an indexed family of endofunctors T_m for $m : M$, with natural transformations $\eta : \mathsf{Id} \dot{\rightarrow} \mathsf{T}_\epsilon$ and $\mu_{m,n} : \mathsf{T}_m \circ \mathsf{T}_n \dot{\rightarrow} \mathsf{T}_{m \otimes n}$ satisfying indexed versions of the three coherence equations of ordinary monads:

$$\mu_{m,\epsilon} \cdot \mathsf{T}_m\ \eta = id \qquad (19a)$$
$$\mu_{\epsilon,n} \cdot \eta\ \mathsf{T}_n = id \qquad (19b)$$
$$\mu_{m,n\otimes p} \cdot \mathsf{T}_m\ \mu_{n,p} = \mu_{m\otimes n,p} \cdot \mu_{m,n}\ \mathsf{T}_p \qquad (19c)$$

A familiar example involves the monoid $(\mathbb{N}, 1, \times)$ of naturals under multiplication, and the family Vector_n of functors representing vectors of length n; the unit $\eta\ A : A \rightarrow \mathsf{Vector}_1\ A$ constructs a singleton vector, and the multiplication $\mu_{m,n}\ A : \mathsf{Vector}_m\ (\mathsf{Vector}_n\ A) \rightarrow \mathsf{Vector}_{m\times n}\ A$ flattens a rectangular vector of vectors into one long vector of elements.

We want the related monoid $(\mathbb{K}*, \langle\,\rangle, ;)$ of finite sequences of finite key types $\mathbb{K}$, under the empty sequence $\langle\,\rangle$ and sequence concatenation $(;)$. (We could instead take finite bags of key types, if we wanted to consider the column order of a table not to be significant.) To be concrete, let $\Pi : \mathbb{K}* \rightarrow \mathbb{K}$ denote the cartesian product $\Pi\ Ks$ of a finite sequence $Ks : \mathbb{K}*$ of finite key types. Then (T, η, μ) is a monad graded over $(\mathbb{K}*, \langle\,\rangle, ;)$, where the carrier $\mathsf{T}_{Ks}\ V$ = Table $(\Pi\ Ks)\ V$ is finite maps from Ks to V, the unit $\eta\ V : V \rightarrow \mathsf{T}_{\langle\rangle}\ V$ constructs a singleton finite map on the empty sequence of key types, and $\mu_{Ks,Ks'}\ V : \mathsf{T}_{Ks}\ (\mathsf{T}_{Ks'}\ V) \rightarrow \mathsf{T}_{Ks;Ks'}\ V$ is relational uncurrying (18).

How do graded monads fit into the adjunction story? At first sight, it does not look promising. An ordinary monad is necessarily an endofunctor on a category $\mathscr{C}$, arising as it does from an adjunction between two categories. An M-graded monad, on the other hand, is a functor $\mathsf{T} : \mathscr{M} \times \mathscr{C} \rightarrow \mathscr{C}$ (viewing monoid M as a discrete category $\mathscr{M}$), or equivalently $\mathsf{T} : \mathscr{M} \rightarrow \mathscr{C}^{\mathscr{C}}$; neither of these is an endofunctor, so neither can arise directly from an adjunction.

Nevertheless, there is an adjunction story to graded monads, albeit under a layer of disguise. A *strict action* of monoid $\mathsf{M} = (M, \epsilon, \otimes)$ on category $\mathscr{D}$ is a family A_m of endofunctors on $\mathscr{D}$ for $m : M$, such that $\mathsf{A}_\epsilon = \mathsf{Id}$ and $\mathsf{A}_{m\otimes n} = \mathsf{A}_m \circ \mathsf{A}_n$; it is a special case of an M-graded monad in which the η and μ are identities. Then an M-graded monad in category $\mathscr{C}$ factorises into an adjunction $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \rightarrow \mathscr{D}$ and a strict M-action on $\mathscr{D}$ [Fujii et al. 2016; Street 1972]. Indeed, each graded

$$
\begin{array}{ll}
[e \mid \;] & = \mathit{return}\ e \\
[e \mid b] & = \mathbf{if}\ b\ \mathbf{then}\ \mathit{return}\ e\ \mathbf{else}\ \mathit{mzero} \\
[e \mid p \leftarrow m] & = \mathit{liftM}\ (\lambda p \rightarrow e)\ m \\
[e \mid q, r] & = \mathit{mult}\ [[e \mid r] \mid q] \\
[e \mid (q \mid r), s] & = \mathit{liftM}\ (\lambda(\mathsf{v}^q, \mathsf{v}^r) \rightarrow [e \mid s])\ (\mathit{mzip}\ [\mathsf{v}^q \mid q]\ [\mathsf{v}^r \mid r]) \\
\multicolumn{2}{l}{[e \mid q, \mathbf{then\ group\ by}\ b\ \mathbf{using}\ f, r]} \\
 & = \mathit{liftM}\ (\lambda ys \rightarrow \mathbf{case}\ (\mathit{fmap}\ \mathsf{v}^q_1\ ys, \ldots, \mathit{fmap}\ \mathsf{v}^q_n\ ys)\ \mathbf{of}\ \mathsf{v}^q \rightarrow [e \mid r]) \\
 & \qquad\qquad (f\ (\lambda \mathsf{v}^q \rightarrow b)\ [\mathsf{v}^q \mid q])
\end{array}
$$

Fig. 5. Definition by translation of Haskell's generalised comprehensions

monad gives rise to a whole family of such factorisations, with two canonical such—a 'least' one and a 'greatest' one. Conversely, any strict M-action on a category $\mathscr{D}$ can be transferred over an adjunction $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \rightarrow \mathscr{D}$ to yield an M-graded monad on $\mathscr{C}$. An ordinary monad T arising from adjunction $\mathsf{L} \dashv \mathsf{R}$ is a graded monad over the trivial singleton monoid; in that case, the canonical factorisations reduce to the Kleisli and Eilenberg–Moore adjunctions for the monad, and conversely, T coincides with the transfer of the identity monad (the trivial strict action of the trivial singleton monoid) over $\mathsf{L} \dashv \mathsf{R}$. The constructions are given in full by Fujii et al. [2016]; the details do not add any more insight beyond the above, so we skip them here, although for completeness we summarize them in Appendix A.

In Section 6 we had the characterisation

$$
\begin{aligned}
x\ {}_f\!\bowtie_g\ y &= \lbag (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \rbag \\
&= \mathit{elems}\ (\mathsf{Map}_*\ (\times)\ (\mathit{merge}\ (x\ \text{`}\mathit{indexBy}\text{`}\ f, y\ \text{`}\mathit{indexBy}\text{`}\ g)))
\end{aligned}
$$

of equijoin in terms of indexed tables. The linear execution time of the latter combinator version is appealing; but it is a shame to have had to give up on the comprehension notation of the former in order to obtain it. On the face of it, this sacrifice seems inevitable: the combinator version is heterogeneous, involving different monads Bag_* and $\mathsf{Table}\ \mathsf{K}$ as the input and output of *indexBy*, whereas traditional comprehension notation is homogeneous.

Nevertheless, it is possible to generalise the comprehension notation to support heterogeneity with grouping [Peyton Jones and Wadler 2007] and merging [Plasmeijer and van Eekelen 1995], as has been done in the Glasgow Haskell Compiler [Giorgidze et al. 2011b; Glasgow 2015]. Haskell's generalised comprehension for a monad M is defined by translation, as shown in Figure 5. In a nutshell, the translation is as follows. The first four clauses precisely match our properties of bag comprehensions (9a)–(9d) in Section 3: *mult* and *return* are the multiplication μ and unit η of the monad M, *liftM* its functorial action, and *mzero* the neutral element of the monoid—in particular, $\mathit{mzero} = \emptyset$ for $M = \mathsf{Bag}_*$.

In the fifth and sixth clauses, v^q denotes the tuple of variables bound by qualifiers q, and v^q_i projects out the ith component from this tuple; and similarly for v^r and r. In the fifth clause, the monad M is assumed to support an appropriate 'zip' function

$$\mathit{mzip} :: M\ a \rightarrow M\ b \rightarrow M\ (a, b)$$

For example, with $M = \mathsf{List}$, a suitable instantiation is $\mathit{mzip} = \mathit{zip}$; so the comprehension

$$[a + b + c \mid (a, b) \leftarrow [(1, 3), (2, 4)] \mid c \leftarrow [5, 6]]$$

desugars to

$$\mathit{liftM}\ (\lambda((a, b), c) \rightarrow a + b + c)\ (\mathit{zip}\ [(a, b) \mid (a, b) \leftarrow [(1, 3), (2, 4)]]\ [c \mid c \leftarrow [5, 6]])$$

and hence evaluates to $[9, 12]$. More interestingly, with $M = \mathsf{Table\ K}$, an appropriate instantiation is $mzip = merge$.

In the sixth clause, the argument f is required to have type $(a \to \mathsf{K}) \to M\ a \to N\ (F\ a)$, where N is a (possibly different) monad, and F some functor; note that this introduces heterogeneity into the comprehension. This translation [Peyton Jones and Wadler 2007] is particularly ingenious, in that the variables v^q bound by qualifiers q and used in key expression b are *rebound* by the **case** expression for the scope containing e and r. For example, GHC provides a function

$$groupWith :: Ord\ b \Rightarrow (a \to b) \to [a] \to [[a]]$$

such that *groupWith k xs* which categorizes the elements of a list *xs* according to *k*, groups them by category, and sorts the groups; then the comprehension

$$\begin{aligned}&[(head\ c, sum\ n) \mid (c, n) \leftarrow [(\texttt{'a'}, 3), (\texttt{'b'}, 4), (\texttt{'a'}, 5)],\\ &\qquad\qquad\qquad \mathbf{then\ group\ by}\ c\ \mathbf{using}\ groupWith]\end{aligned}$$

desugars to

$$\begin{aligned}&liftM\ (\lambda ys \to \mathbf{case}\ (fmap\ fst\ ys, fmap\ snd\ ys)\ \mathbf{of}\ (c, n) \to [(head\ c, sum\ n) \mid])\\ &\qquad (groupWith\ (\lambda(c, n) \to c)\ [(c, n) \mid (c, n) \leftarrow [(\texttt{'a'}, 3), (\texttt{'b'}, 4), (\texttt{'a'}, 5)]\ ])\end{aligned}$$

and hence evaluates to $[(\texttt{'a'}, 8), (\texttt{'b'}, 4)]$ (note that c and n are bound initially to characters and integers, and then rebound to lists of such). For the purposes of this paper, we will use *indexBy* for f, thereby instantiating both M and F to Bag_* and N to $\mathsf{Table\ K}$.

(In GHC [Glasgow 2015], the choice of *mzip* function is specified indirectly by way of a subclass *MonadZip* of *Monad*, whereas the grouping function f is specified directly.) The full translation of the extended comprehension notation is discussed in more detail by its designers [Giorgidze et al. 2011a,b; Peyton Jones and Wadler 2007] and, in this particular context, in a related paper [Gibbons 2016]. Others have designed embedded query DSLs with grouping and aggregation [Suzuki et al. 2016], but not as far as we know in terms of comprehensions.

With these extensions, we can write the equijoin $x\ {}_f\!\bowtie_g\ y$ in Haskell as follows:

$$\begin{aligned}&elems\ [\ cp\ a\ b \mid a \leftarrow x, \mathbf{then\ group\ by}\ f\ \mathbf{using}\ indexBy\\ &\qquad\qquad\quad \mid b \leftarrow y, \mathbf{then\ group\ by}\ g\ \mathbf{using}\ indexBy]\end{aligned}$$

—note how the variable a is bound first to an element of x by the qualifier $a \leftarrow x$, and used as such in the key expression f; but rebound to a finite bag of such elements by the grouping function *indexBy*, and used as such in the term $cp\ a\ b$. The example query from Section 1 turns out as follows:

$$\begin{aligned}&reduce\\ &\quad [\ \ cp\ name\ amount\\ &\quad \mid\ (cid, name) \leftarrow customers, \mathbf{then\ group\ by}\ cid\ \mathbf{using}\ indexBy\\ &\quad \mid\ (iid, cust, due, amount) \leftarrow invoices, due < today, \mathbf{then\ group\ by}\ cust\ \mathbf{using}\ indexBy\\ &\quad]\end{aligned}$$

So graded monads are really a small generalisation of ordinary monads, and enjoy generalisations of the same properties. Crucially, they still support the same comprehension notation, satisfying the same equational properties. In particular, the translation of grouping and zipping comprehensions in Figure 5 works as well for graded monads as for ungraded ones, and for heterogeneous comprehensions (where grouping takes an $M\ A$-collection to a nested $N\ (\mathsf{F}\ A)$ collection) as for homogeneous ones (where M, N, and F coincide). We do not repeat and renumber all the previous comprehension properties here; but it is important to note that subsequent appeals to properties about 'comprehensions' (for example, in Section 8 below) depend implicitly on their definition for graded monads.

8 QUERY TRANSFORMATION

The classical System R approach to database query optimization [Selinger et al. 1979] is based on enumerating a number of extensionally equivalent expressions for an input query, estimating the execution cost of each of them, and then generating code for the best one. Many of the equivalences between query expressions amount to algebraic properties of the various operators, which in turn arise from the adjunctions we have explored above. We discuss some of the most important such equivalences in what follows. As a reminder, the relational algebra operators have the following specifications:

$$\begin{aligned} \pi_f &= \mathsf{Bag}\ f \\ \sigma_p &= \mathit{filter}\ p \\ x\ {}_f\!\bowtie_g\ y &= \wr (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \int \end{aligned}$$

(although equijoin has the more efficient implementation discussed in Section 6).

We do not cover equivalences that depend on data invariants, such as functional dependencies, nor how to use statistics on the actual data in the tables in order to choose among equivalent queries; both aspects depend on information outside our model, and "whereof one cannot speak, thereof one must be silent".

8.1 High-School Algebra

Tarski's *high-school algebra identities* [Burris and Lee 1993; Fiore et al. 2006] are the eleven basic laws of the positive naturals that one typically learns at school: addition is associative and commutative; multiplication is associative and commutative with neutral element 1; multiplication distributes over addition; exponentiation promotes through addition ($x \uparrow (y + z) = (x \uparrow y) \times (x \uparrow z)$) and distributes over multiplication ($(x \times y) \uparrow z = (x \uparrow z) \times (y \uparrow z)$); exponentiation is a right action of the monoid of multiplication ($x \uparrow (y \times z) = (x \uparrow y) \uparrow z$ and $x \uparrow 1 = x$); and 1 is a left zero of exponentiation.

But the positive naturals are not the only model of this equational theory; another model is given by $(\mathsf{Set}, \uplus, \times, \rightarrow, \{*\})$, which arises from sets under disjoint union, cartesian product, function spaces, and singletons. So too are finite non-empty sets under the same operations; and set cardinality is a theory morphism from finite non-empty sets to the positive naturals (that is, it preserves all the laws). These properties of sets fall out of the adjunctions in Figure 2(a),(b) yielding sums, products, and singletons.

To show that multiplication distributes through addition, we use the *Yoneda Embedding*, which states that

$$B \cong A \iff \mathscr{C}(A, \text{-}) \cong \mathscr{C}(B, \text{-})$$

One can see the Yoneda Embedding as a high-level reasoning principle of *indirect proof*; in the special case of ordered sets, it becomes

$$(b = a) \iff (\forall c\ .\ (a \leqslant c) \Leftrightarrow (b \leqslant c))$$

We can use the Yoneda Embedding to show that $\mathsf{L}\ (A_1 + A_2) \cong \mathsf{L}\ A_1 + \mathsf{L}\ A_2$ whenever we have an adjunction $\mathsf{L} \dashv \mathsf{R}$—it suffices to show $\mathbf{Set}\ (\mathsf{L}\ (A_1 + A_2), X) \cong \mathbf{Set}\ (\mathsf{L}\ A_1 + \mathsf{L}\ A_2, X)$ for arbitrary X, and so we calculate:

$$
\begin{array}{ll}
& \mathbf{Set}\ (\mathsf{L}\ (A_1 + A_2), X) \\
\cong & \quad \{\ \mathsf{L} \dashv \mathsf{R}\ \} \\
& \mathbf{Set}\ (A_1 + A_2, \mathsf{R}\ X) \\
\cong & \quad \{\ + \dashv \Delta\ \} \\
& \mathbf{Set}\ (A_1, \mathsf{R}\ X) \times \mathbf{Set}\ (A_2, \mathsf{R}\ X) \\
\cong & \quad \{\ \mathsf{L} \dashv \mathsf{R}\ \} \\
& \mathbf{Set}\ (\mathsf{L}\ A_1, X) \times \mathbf{Set}\ (\mathsf{L}\ A_2, X) \\
\cong & \quad \{\ + \dashv \Delta\ \} \\
& \mathbf{Set}\ (\mathsf{L}\ A_1 + \mathsf{L}\ A_2, X)
\end{array}
$$

Since - $\times B$ is a left adjoint, this gives us the special case $(A_1 + A_2) \times B \cong A_1 \times B + A_2 \times B$ of distributing a product over a sum.

8.2 Projections

Relational projections π_i are a special case of the functorial action $\mathsf{F}\ i$ of the functor F representing database tables (whether that functor be Bag, Bag_*, or Table K), reading i as both the names of some columns and the function that extracts those fields from a single record. Equivalences for projections are therefore special cases of the functor laws. For example, assuming columns i are a subset of columns j, then field extraction i subsumes j (that is, $i \cdot j = i$), and so

$$
\begin{array}{ll}
& \pi_i \cdot \pi_j \\
= & \quad \{\ \text{projections as functorial action}\ \} \\
& \text{Bag}\ i \cdot \text{Bag}\ j \\
= & \quad \{\ \text{functors preserve composition (2b)}\ \} \\
& \text{Bag}\ (i \cdot j) \\
= & \quad \{\ \text{subsumption: } i \cdot j = i\ \} \\
& \text{Bag}\ i \\
= & \quad \{\ \text{projection as functorial action}\ \} \\
& \pi_i
\end{array}
$$

8.3 Selections

We represent the relational selection σ_p as *filter* p, reading p as a predicate, that is, a $\mathbb{B}$-valued function. We can simplify and combine filters as follows, where *false*, *true*, and conjunction are lifted to predicates.

$$\mathit{filter}\ \mathit{false}\ s = \emptyset \tag{20a}$$

$$\mathit{filter}\ \mathit{true}\ s = s \tag{20b}$$

$$\mathit{filter}\ (p \wedge q)\ s = \mathit{filter}\ p\ (\mathit{filter}\ q\ s) \tag{20c}$$

All of these equalities are easy to show, since *filter* p, constant functions and the identity are all homomorphisms, and two homomorphisms from Free A are equal iff they coincide on singleton bags (4). In particular, the last equality shows that consecutive selections can be fused.

The property that *filter* p is a dinatural transformation (7d) shows how to commute a selection σ_p and a projection π_i, when predicate p depends only on columns i: then as boolean predicates,

$p \cdot i = p$, and so

$$\begin{array}{cl} & \sigma_p \cdot \pi_i \\ = & \quad \{ \text{ selections and projections } \} \\ & \mathit{filter}\ p \cdot \mathsf{Bag}\ i \\ = & \quad \{ \text{ dinaturality (7d) } \} \\ & \mathsf{Bag}\ i \cdot \mathit{filter}\ (p \cdot i) \\ = & \quad \{ \text{ subsumption: } p \cdot i = p \ \} \\ & \mathsf{Bag}\ i \cdot \mathit{filter}\ p \\ = & \quad \{ \text{ selections and projections } \} \\ & \pi_i \cdot \sigma_p \end{array}$$

8.4 Aggregations

Selection $\sigma_p = \mathit{reduce}\ (\mathsf{Free}\ A) \cdot \mathsf{Bag}\ (\mathit{guard}\ p)$ is a particular homomorphism; aggregations more generally are also of this form, as we saw in Section 3, and enjoy similar properties, for similar reasons. For example, a homomorphism $h = \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ f$ for monoid $\mathsf{M} = (M, \epsilon, \otimes)$ can always absorb a projection:

$$\begin{array}{cl} & h \cdot \pi_i \\ = & \quad \{ \text{ homomorphism, projection } \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ f \cdot \mathsf{Bag}\ i \\ = & \quad \{ \text{ functor composition (2b) } \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ (f \cdot i) \end{array}$$

and a selection:

$$\begin{array}{cl} & h \cdot \sigma_p \\ = & \quad \{ \text{ selection } \} \\ & h \cdot \mathit{reduce}\ (\mathsf{Free}\ A) \cdot \mathsf{Bag}\ (\mathit{guard}\ p) \\ = & \quad \{ \ h : \mathsf{Free}\ A \longrightarrow \mathsf{M}, \text{ naturality (6b) } \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ h \cdot \mathsf{Bag}\ (\mathit{guard}\ p) \\ = & \quad \{ \text{ functor composition (2b) } \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ (h \cdot \mathit{guard}\ p) \\ = & \quad \{ \ \mathit{guard} \ \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ (\lambda a \rightarrow h\ (\mathbf{if}\ p\ a\ \mathbf{then}\ \wr\, a \,\int\ \mathbf{else}\ \emptyset)) \\ = & \quad \{ \text{ homomorphisms: } h\ \wr\, a \,\int = f\ a \text{ (5b)}, h\ \emptyset = \epsilon \text{ (5a) } \} \\ & \mathit{reduce}\ \mathsf{M} \cdot \mathsf{Bag}\ (\lambda a \rightarrow \mathbf{if}\ p\ a\ \mathbf{then}\ f\ a\ \mathbf{else}\ \epsilon) \end{array}$$

8.5 Comprehensions

The monad laws justify certain manipulations of monad comprehensions. For example, associativity of qualifier sequencing follows from associativity of monad multiplication μ:

$$\begin{aligned}
& \wr\, e \mid (q, q'), q'' \,\int \\
=\;& \quad \{ \text{ sequencing of qualifiers (9d) } \} \\
& \mu \;\wr\, \wr\, e \mid q'' \int \mid q, q' \,\int \\
=\;& \quad \{ \text{ sequencing of qualifiers (9d) again } \} \\
& \mu\, (\mu \;\wr\, \wr\, \wr\, e \mid q'' \int \mid q' \int \mid q \int) \\
=\;& \quad \{ \text{ associativity (8c): } \mu \cdot \mu = \mu \cdot \mathsf{Bag}\ \mu \} \\
& \mu\, (\mathsf{Bag}\ \mu \;\wr\, \wr\, \wr\, e \mid q'' \int \mid q' \int \mid q \int) \\
=\;& \quad \{ \text{ functors over comprehension (10) } \} \\
& \mu \;\wr\, \mu \;\wr\, \wr\, e \mid q'' \int \mid q' \int \mid q \,\int \\
=\;& \quad \{ \text{ sequencing of qualifiers (9d) } \} \\
& \mu \;\wr\, \wr\, e \mid q', q'' \int \mid q \,\int \\
=\;& \quad \{ \text{ sequencing of qualifiers (9d) again } \} \\
& \wr\, e \mid q, (q', q'') \,\int
\end{aligned}$$

(Indeed, this implication is an equivalence [Wadler 1992].) The above shows how to flatten an inner comprehension nested in the term part of an outer comprehension; for one nested in the qualifier part, we have:

$$\begin{aligned}
& \wr\, e \mid p \leftarrow \wr\, e' \mid q \int \int \\
=\;& \quad \{ \text{ generators (9c) } \} \\
& \mathsf{Bag}\ (\lambda p \to e) \;\wr\, e' \mid q \,\int \\
=\;& \quad \{ \text{ functors over comprehension (10) } \} \\
& \wr\, (\lambda p \to e)\ e' \mid q \,\int
\end{aligned}$$

8.6 Joins

Join order optimization is one of the most powerful optimization techniques, even though it is known to be insufficient for generating worst-case optimal join implementations [Ngo et al. 2012]. It can be shown that extending *merge* from binary to *n*-ary functions, combined with the generic trie implementation of *ix* based on (15a)–(15d) [Henglein and Hinze 2013] and symbolic Cartesian product constructors, yields worst-case optimal joins that dramatically outperform conventional query processors on cyclic join queries, specifically triangle queries, also in practice [Henglein and Larsen 2010].

Cartesian product and equijoin are associative and commutative (up to canonical isomorphisms), and so queries involving two or more joins offer many opportunities for rearranging the query in order to improve performance. The pivotal observation is the efficient implementation of equijoin:

$$\begin{aligned}
x \;{}_f\!\bowtie_g\; y &= \wr\, (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \,\int \\
&= \mathit{elems}\ (\mathsf{Map}_*\ \mathsf{K}\ (\times)\ (\mathit{merge}\ (x\ `\mathit{indexBy}`\ f, y\ `\mathit{indexBy}`\ g)))
\end{aligned}$$

as discussed in Section 6.

The comprehension version is much easier to manipulate than the combinator-style one. For example, here is the proof that join is commutative, up to the isomorphism *swap* $(a, b) = (b, a)$ on pairs:

$$\begin{array}{ll}
& x \;{}_{f}{\bowtie}_{g}\; y \\
= & \quad \{ \text{equijoin as comprehension} \} \\
& \wr\, (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \,\int \\
= & \quad \{ \text{Bag is a commutative monad (12)} \} \\
& \wr\, (a, b) \mid b \leftarrow y, a \leftarrow x, f\ a == g\ b \,\int \\
= & \quad \{ \mathit{swap}; == \text{ is symmetric} \} \\
& \wr\, \mathit{swap}\ (b, a) \mid b \leftarrow y, a \leftarrow x, g\ b == f\ a \,\int \\
= & \quad \{ \text{functors over comprehension (10)} \} \\
& \mathsf{Bag}\ \mathit{swap}\ \wr\, (b, a) \mid b \leftarrow y, a \leftarrow x, g\ b == f\ a \,\int \\
= & \quad \{ \text{equijoin as comprehension} \} \\
& \mathsf{Bag}\ \mathit{swap}\ (y \;{}_{g}{\bowtie}_{f}\; x)
\end{array}$$

Similarly, join is associative, up to the nested pair isomorphism $\mathit{assoc}\ ((a, b), c) = (a, (b, c))$:

$$\begin{array}{ll}
& (x \;{}_{f}{\bowtie}_{g}\; y) \;{}_{g \cdot \mathit{snd}}{\bowtie}_{h}\; z \\
= & \quad \{ \text{equijoin as comprehension} \} \\
& \wr\, ((a, b), c) \mid (a, b) \leftarrow \wr (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \int, c \leftarrow z, g\ b == h\ c \,\int \\
= & \quad \{ \text{comprehensions} \} \\
& \wr\, ((a, b), c) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b, c \leftarrow z, g\ b == h\ c \,\int \\
= & \quad \{ \mathit{assoc}; c \text{ not free in } f, g \} \\
& \wr\, \mathit{assoc}\ (a, (b, c)) \mid a \leftarrow x, b \leftarrow y, c \leftarrow z, g\ b == h\ c, f\ a == g\ b \,\int \\
= & \quad \{ \text{functors over comprehension (10)} \} \\
& \mathsf{Bag}\ \mathit{assoc}\ \wr\, (a, (b, c)) \mid a \leftarrow x, b \leftarrow y, c \leftarrow z, g\ b == h\ c, f\ a == g\ b \,\int \\
= & \quad \{ \text{equijoin as comprehension} \} \\
& \mathsf{Bag}\ \mathit{assoc}\ (x \;{}_{f}{\bowtie}_{g \cdot \mathit{fst}}\; (y \;{}_{g}{\bowtie}_{h}\; z))
\end{array}$$

8.7 Combining Joins with Other Operations

A projection $\pi_{i \times j}$ can be promoted through an equijoin ${}_{f}{\bowtie}_{g}$ if it is compatible with the join criterion, that is, if there exist f', g' such that $(f\ a == g\ b) \Leftrightarrow (f'\ (i\ a) == g'\ (j\ b))$:

$$\begin{array}{ll}
& \pi_{i \times j}\ (x \;{}_{f}{\bowtie}_{g}\; y) \\
= & \quad \{ \text{projection, equijoin} \} \\
& \mathsf{Bag}\ (i \times j)\ \wr\, (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \,\int \\
= & \quad \{ \text{functors over comprehension (10)} \} \\
& \wr\, (i \times j)\ (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \,\int \\
= & \quad \{ \text{product} \} \\
& \wr\, (i\ a, j\ b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \,\int \\
= & \quad \{ \text{compatibility} \} \\
& \wr\, (i\ a, j\ b) \mid a \leftarrow x, b \leftarrow y, f'\ (i\ a) == g'\ (j\ b) \,\int \\
= & \quad \{ \text{functors} \} \\
& \wr\, (a', b') \mid a' \leftarrow \mathsf{Bag}\ i\ x, b' \leftarrow \mathsf{Bag}\ j\ y, f'\ a' == g'\ b' \,\int \\
= & \quad \{ \text{projection, equijoin} \} \\
& (\pi_i\ x) \;{}_{f'}{\bowtie}_{g'}\; (\pi_j\ y)
\end{array}$$

Similarly, if a selection σ_p after a join uses a predicate p that can be factorised to act independently on the components of the pair—that is, there exist q, r with $p\ (a, b) = q\ a \wedge r\ b$ with b not free in

$q\ a$ and a not free in $r\ b$—then the selection can be promoted through the join:

$$
\begin{array}{ll}
& \sigma_p\ (x\ {}_f{\bowtie}_g\ y) \\
= & \quad \{\text{ selection, equijoin }\} \\
& \Lbag (a, b) \mid (a, b) \leftarrow \Lbag (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \Rbag, p\ (a, b) \Rbag \\
= & \quad \{\text{ factorising predicate }\} \\
& \Lbag (a, b) \mid (a, b) \leftarrow \Lbag (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b \Rbag, q\ a \wedge r\ b \Rbag \\
= & \quad \{\text{ comprehensions }\} \\
& \Lbag (a, b) \mid a \leftarrow x, b \leftarrow y, f\ a == g\ b, q\ a, r\ b \Rbag \\
= & \quad \{\ b \text{ is not free in } q\ a\ \} \\
& \Lbag (a, b) \mid a \leftarrow x, q\ a, b \leftarrow y, r\ b, f\ a == g\ b \Rbag \\
= & \quad \{\text{ comprehensions }\} \\
& \Lbag (a, b) \mid a \leftarrow \Lbag a \mid a \leftarrow x, q\ a \Rbag, b \leftarrow \Lbag b \mid b \leftarrow y, r\ b \Rbag, f\ a == g\ b \Rbag \\
= & \quad \{\text{ selection, equijoin }\} \\
& \sigma_q\ x\ {}_f{\bowtie}_g\ \sigma_r\ y
\end{array}
$$

To sum up, not only do adjunctions provide the structures required for database operations, but they also give rise to the properties we need for optimizations.

9 CONCLUSION

Comprehensions as a notation for queries, generalizable to various kinds of collection monad, have for decades been a staple of programming interfaces to data. Monads enjoy rich mathematical structure, deriving from category theory, and manifesting as a body of equivalences that provides a basis for many query optimizations. However, the monadic structure alone does not provide a good story about joins in queries; and reducing joins to cartesian products leads to extremely inefficient implementations. Efficient implementations—taking time linear in the size of the input—require a different explanation. We have presented such an explanation; it is just as rich and elegant as the naive monadic story, being similarly based on a few simple adjunctions.

Our construction depends crucially on the interplay between multiple monads. Bags are a sweet spot for expressivity of queries: lists would preserve too many distinctions, in terms of ordering of results—so denying some opportunities for query optimization, in particular for joins. On the other hand, sets collapse too many distinctions, in terms of multiplicity of elements—thereby denying non-idempotent aggregations such as count and sum. But bags alone are insufficient for representing the indexing and merging that is necessary for an efficient implementation of joins; for that, we also need maps. And for maps to be aggregatable, they need to be finite. And finally, since finite maps are not a plain monad, we had to introduce the refined notion of graded monads.

This necessary interplay between heterogeneous monads has the unfortunate consequence that we can no longer write a query involving a join as a traditional comprehension, because these can only be homogeneous. Nevertheless, as we showed in Section 7, recent extensions to Haskell's comprehension notation to support grouping [Peyton Jones and Wadler 2007] and merging [Plasmeijer and van Eekelen 1995] support the necessary heterogeneity that allows us to express equijoins efficiently in the comprehension notation.

This means that we do not have to resort to the combinator style just to get efficient implementation of joins; comprehensions still suffice as a surface syntax. Nevertheless, the combinator style is arguably more appropriate as an intermediate representation; database management systems traditionally translate input queries from the comprehension style ('relational calculus') to combinator style ('relational algebra') in preparation for query optimization (although it is possible to carry out the query transformations directly in the comprehension style, as for example Cheney et al.

[2013] do). And conversely, it is possible to reconstruct joins from queries written in the ordinary comprehension style, albeit at the cost of considerable analysis.

The investigations presented here are a first step towards a greater goal. We have shown that adjunctions give rise to all the combinators needed to give a rigorous explanation of the selections, projections, and joins of relational algebra, providing both the basis for an efficient implementation and the justification for the body of transformations needed in query optimization. We are optimistic that these constructions might form a firm foundation for an intermediate representation for implementing database connection frameworks such as Microsoft's LINQ [Meijer 2007; Syme 2006]. In general, this work lends evidence to the argument that expressive type systems (here, notably the use of type classes for datatype-generic programming) are important for accommodating embedded domain-specific languages.

ACKNOWLEDGEMENTS

We are grateful to the members of IFIP Working Groups 2.1 on *Algorithmic Languages and Calculi* and 2.11 on *Program Generation* for valuable discussions about this work at meetings; special thanks are due to Sam Lindley, Conor McBride, and Jacques Carette. We also thank the anonymous reviewers of earlier versions for their helpful comments. This work has been funded by UK EPSRC project *Unifying Theories of Generic Programming* (grant number EP/J010995/1) and Innovation Fund Denmark project *HIPERFIT* (grant number 10-092299).

REFERENCES

Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit Parallelism through Deep Language Embedding. In *SIGMOD record*. ACM, 47–61. https://doi.org/10.1145/2723372.2750543

Steve Awodey. 2006. *Category Theory*. Oxford University Press.

Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. 1994. Comprehension Syntax. *SIGMOD Record* 23, 1 (1994), 87–96. https://doi.org/10.1145/181550.181564

Stanley Burris and Simon Lee. 1993. Tarski's High School Identities. *Amer. Math. Monthly* 100, 3 (March 1993), 231–236.

Jacques Carette. 2018. Email correspondence. (May 2018). Personal communication.

James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-Integrated Query. In *International Conference on Functional Programming*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 403–416. https://doi.org/10.1145/2500365.2500586

Eugenia Cheng. 2015. *Cakes, Custard, and Category Theory*. Profile Books.

Richard H. Connelly and F. Lockwood Morris. 1995. A Generalization of the Trie Data Structure. *Mathematical Structures in Computer Science* 5, 3 (September 1995), 381–418. https://doi.org/10.1017/S0960129500000803

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming without Tiers. In *Formal Methods for Components and Objects (Lecture Notes in Computer Science)*, Vol. 4709. Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

John Darlington. 1975. Application of Program Transformation to Program Synthesis. In *IRIA Symposium on Proving and Improving Programs*. Arc-et-Senans, France, 133–144.

Christopher J. Date. 2004. *An Introduction to Database Systems* (8th ed.). Pearson.

Mary Fernandez, Jerome Simeon, and Philip Wadler. 2001. A Semi-Monad for Semi-Structured Data. In *International Conference on Database Theory (Lecture Notes in Computer Science)*, Jan Van den Bussche and Victor Vianu (Eds.), Vol. 1973. Springer, 263–300. https://doi.org/10.1007/3-540-44503-X_18

Marcelo Fiore, Roberto Di Cosmo, and Vincent Balat. 2006. Remarks on Isomorphisms in Typed Lambda Calculi with Empty and Sum Types. *Annals of Pure and Applied Logic* 141, 1-2 (2006), 35–50. https://doi.org/10.1016/j.apal.2005.09.001

Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. 2016. Towards a Formal Theory of Graded Monads. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*. Springer-Verlag, 513–530. https://doi.org/10.1007/978-3-662-49630-5_30

Jeremy Gibbons. 2016. Comprehending Ringads. In *A List of Successes that can Change the World (Lecture Notes in Computer Science)*, Sam Lindley, Conor McBride, Don Sannella, and Phil Trinder (Eds.), Vol. 9600. Springer, 132–151. https://doi.org/10.1007/978-3-319-30936-1_7

George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. 2011a. Haskell Boards the Ferry: Database-Supported Program Execution for Haskell. In *Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Jurriaan Hage and Marco T. Morazán (Eds.), Vol. 6647. Springer, 1–18. https://doi.org/10.1007/978-3-642-24276-2_1

George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. 2011b. Bringing Back Monad Comprehensions. In *Haskell Symposium*. ACM, 13–22. https://doi.org/10.1145/2034675.2034678

Glasgow 2015. *Glasgow Haskell Compiler Users' Guide, Version 7.10.1*. https://downloads.haskell.org/~ghc/7.10.1/docs/html/users_guide/index.html.

Torsten Grust and Marc H. Scholl. 1999. How to Comprehend Queries Functionally. *Journal of Intelligent Information Systems* 12, 2-3 (1999), 191–218. https://doi.org/10.1023/A:1008705026446

Fritz Henglein and Ralf Hinze. 2013. Distributive Sorting and Searching: From Generic Discrimination to Generic Tries. In *Asian Symposium on Programming Languages and Systems (Lecture Notes in Computer Science)*, Chung-chieh Shan (Ed.), Vol. 8301. Springer, 315–332. https://doi.org/10.1007/978-3-319-03542-0_23

Fritz Henglein and Ken Friis Larsen. 2010. Generic Multiset Programming with Discrimination-Based Joins and Symbolic Cartesian Products. *Higher-Order and Symbolic Computation* 23, 3 (2010), 337–370. https://doi.org/10.1007/s10990-011-9078-8

Ralf Hinze. 2000. Generalizing Generalized Tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. https://doi.org/10.1017/S0956796800003713

Ralf Hinze. 2003. Fun with Phantom Types. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave Macmillan, 245–262.

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Principles of Programming Languages*. ACM, 633–645. https://doi.org/10.1145/2535838.2535846

Leonid Libkin and Limsoon Wong. 1997. Query Languages for Bags and Aggregate Functions. *J. Comput. System Sci.* 55, 2 (1997), 241–272. https://doi.org/10.1006/jcss.1997.1523

Conor McBride and Ross Paterson. 2008. Functional Pearl: Applicative Programming with Effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Erik Meijer. 2007. Confessions of a Used Programming Language Salesman. In *Object-Oriented Programming: Systems, Languages and Applications*. ACM, 677–694. https://doi.org/10.1145/1297027.1297078

Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-Case Optimal Join Algorithms. In *Principles of Database Systems*. ACM, 37–48. https://doi.org/10.1145/2213556.2213565

Dominic A. Orchard, Tomas Petricek, and Alan Mycroft. 2014. The Semantic Marriage of Monads and Effects. *CoRR* abs/1401.5391 (2014).

Simon Peyton Jones and Philip Wadler. 2007. Comprehensive Comprehensions. In *Haskell Workshop*. ACM, 61–72. https://doi.org/10.1145/1291201.1291209

Benjamin C. Pierce. 1991. *Basic Category Theory for Computer Scientists*. MIT Press.

Rinus Plasmeijer and Marko van Eekelen. 1995. *Concurrent Clean Language Report (Version 1.0)*. Technical Report. University of Nijmegen. ftp://ftp.science.ru.nl/pub/Clean/old/Clean10/doc/refman.ps.gz.

Jacob T. Schwartz, Robert B. K. Dewar, Ed Dubinsky, and Edmond Schonberg. 1986. *Programming with Sets: An Introduction to SETL*. Springer, New York.

Patricia Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD record*. ACM, 23–34. https://doi.org/10.1145/582095.582099

Ross Street. 1972. Two Constructions on Lax Functors. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 13, 3 (1972), 217–264. http://eudml.org/doc/91107

Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-Extensible and Efficient Language-Integrated Query. In *Partial Evaluation and Program Manipulation*, Martin Erwig and Tiark Rompf (Eds.). ACM, 37–48. https://doi.org/10.1145/2847538.2847542

Don Syme. 2006. Leveraging .NET Meta-programming Components from F#: Integrated Queries and Interoperable Heterogeneous Execution. In *ML Workshop*. ACM, New York, NY, USA, 43–54. https://doi.org/10.1145/1159876.1159884

Philip W. Trinder. 1991. Comprehensions, a Query Notation for DBPLs. In *Database Programming Languages*. 55–68.

Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

Limsoon Wong. 2000. Kleisli, a Functional Query System. *Journal of Functional Programming* 10, 1 (2000), 19–56.

A GRADED MONADS FROM ADJUNCTIONS

In this appendix, we outline a special case of Street's constructions [Street 1972], showing the connection between graded monads, strict actions, and adjunctions. This connection is due to Fujii et al. [2016]. We include it here for completeness.

An M-graded monad (T, η, μ) over a monoid $\mathsf{M} = (M, \epsilon, \otimes)$ in category $\mathscr{C}$ is an indexed family of endofunctors T_m for $m : M$, with natural transformations $\eta : \mathsf{Id} \dot{\to} \mathsf{T}_\epsilon$ and $\mu_{m,n} : \mathsf{T}_m \circ \mathsf{T}_n \dot{\to} \mathsf{T}_{m \otimes n}$ such that the following diagrams commute:

$$\begin{tikzcd}
\mathsf{T}_m \arrow[r, "\mathsf{T}_m\ \eta"] \arrow[dr, equal] & \mathsf{T}_m \circ \mathsf{T}_\epsilon \arrow[d, "\mu_{m,\epsilon}"] \\
 & \mathsf{T}_m
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T}_\epsilon \circ \mathsf{T}_m \arrow[d, "\mu_{\epsilon,m}"'] & \mathsf{T}_m \arrow[l, "\eta\ \mathsf{T}_m"'] \arrow[dl, equal] \\
\mathsf{T}_m &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T}_m \circ \mathsf{T}_n \circ \mathsf{T}_p \arrow[r, "\mathsf{T}_m\ \mu_{n,p}"] \arrow[d, "\mu_{m,n}\ \mathsf{T}_p"'] & \mathsf{T}_m \circ \mathsf{T}_{n \otimes p} \arrow[d, "\mu_{m,n\otimes p}"] \\
\mathsf{T}_{m\otimes n} \circ \mathsf{T}_p \arrow[r, "\mu_{m\otimes n,p}"'] & \mathsf{T}_{m\otimes n\otimes p}
\end{tikzcd}$$

A strict M-action in $\mathscr{C}$ is an M-graded monad for which η and $\mu_{m,n}$ are identities. (Equivalently, a strict M-action in $\mathscr{C}$ is a functor $A : \mathscr{M} \to \mathbf{Cat}$ such that $A* = \mathscr{C}$, where $\mathbf{Cat}$ is the category of functors and natural transformations, and treating monoid M as a category $\mathscr{M}$ with one object $*$. An M-graded monad in $\mathscr{C}$ is similarly a *lax functor* $\mathsf{T} : \mathscr{M} \to \mathbf{Cat}$—that is, a generalisation of functors in which the coherence of identity and composition is given by arrows rather than being equalities.)

We will show that adjunctions $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \to \mathscr{D}$ determine a relationship between strict M-actions A on $\mathscr{D}$ and M-graded monads T on $\mathscr{C}$: we can transfer an A across $\mathsf{L} \dashv \mathsf{R}$ to yield T; and conversely, given T, we can construct $\mathscr{D}$, an adjunction $\mathsf{L} \dashv \mathsf{R}$, and an A whose transfer across $\mathsf{L} \dashv \mathsf{R}$ coincides with T.

Given an adjunction $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \to \mathscr{D}$, and a strict M-action A on $\mathscr{D}$, we define M-graded monad T on $\mathscr{C}$ as follows. We define the family of endofunctors by $\mathsf{T}_m = \mathsf{R} \circ \mathsf{A}_m \circ \mathsf{L}$. The unit η is just the unit

$$\eta : \mathsf{Id} \dot{\to} \mathsf{R} \circ \mathsf{L} = \mathsf{R} \circ \mathsf{A}_\epsilon \circ \mathsf{L} = \mathsf{T}_\epsilon$$

of the adjunction. The multiplications are derived from the counit ε of the adjunction by

$$\mu_{m,n} = \mathsf{R}\ \mathsf{A}_m\ \varepsilon\ \mathsf{A}_n\ \mathsf{L} : \mathsf{T}_m \circ \mathsf{T}_n = \mathsf{R} \circ \mathsf{A}_m \circ \mathsf{L} \circ \mathsf{R} \circ \mathsf{A}_n \circ \mathsf{L} \dot{\to} \mathsf{R} \circ \mathsf{A}_m \circ \mathsf{A}_n \circ \mathsf{L} = \mathsf{T}_{m\otimes n}$$

One can view the T so constructed as the transfer of A along $\mathsf{L} \dashv \mathsf{R}$; it is straightforward to check that it does indeed satisfy the conditions to be a graded monad.

Conversely, given an M-graded monad T on $\mathscr{C}$, can we construct a category $\mathscr{D}$, an adjunction $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \to \mathscr{D}$, and a strict M-action A on $\mathscr{D}$ such that the transfer of A along $\mathsf{L} \dashv \mathsf{R}$ coincides with T? Street [1972] showed that there is in fact a whole category of such constructions, with two canonical such: an 'initial' one and a 'final' one. We show the latter here.

We are given a monoid $\mathsf{M} = (M, \epsilon, \otimes)$ and an M-graded monad T on $\mathscr{C}$. A (second) way of viewing M as a category $\mathscr{M}'$ is as a *discrete* category (the objects are the elements of M, and the only arrows are identity arrows), but as a *strict monoidal* category to capture the monoidal structure. We define a category $\mathscr{C}^{\mathsf{T}}$ as follows. The objects are pairs (F, f) consisting of a functor $\mathsf{F} : \mathscr{M}' \to \mathscr{C}$ and a family of arrows $f_{m,n} : \mathsf{T}_m\ (\mathsf{F}\ n) \to \mathsf{F}\ (m \otimes n)$ such that the first two diagrams below commute.

$$\begin{tikzcd}
\mathsf{F}\ m \arrow[r, "\eta\ (\mathsf{F}\ m)"] \arrow[dr, equal] & \mathsf{T}_\epsilon\ (\mathsf{F}\ m) \arrow[d, "f_{\epsilon,m}"] \\
 & \mathsf{F}\ m
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{T}_m\ \mathsf{T}_n\ (\mathsf{F}\ p) \arrow[r, "\mathsf{T}_m\ f_{n,p}"] \arrow[d, "\mu_{m,n}\ (\mathsf{F}\ p)"'] & \mathsf{T}_m\ (\mathsf{F}\ (n \otimes p)) \arrow[d, "f_{m,n\otimes p}"] \\
\mathsf{T}_{m\otimes n}\ (\mathsf{F}\ p) \arrow[r, "f_{m\otimes n,p}"'] & \mathsf{F}\ (m \otimes n \otimes p)
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{T}_m\ (\mathsf{F}\ n) \arrow[r, "\mathsf{T}_m\ (k\ n)"] \arrow[d, "f_{m,n}"'] & \mathsf{T}_m\ (\mathsf{G}\ n) \arrow[d, "g_{m,n}"] \\
\mathsf{F}\ (m \otimes n) \arrow[r, "k\ (m\otimes n)"'] & \mathsf{G}\ (m \otimes n)
\end{tikzcd}$$

The arrows $k : (\mathsf{F}, f) \to (\mathsf{G}, g)$ are natural transformations $k : \mathsf{F} \dot{\to} \mathsf{G}$ such that the third diagram above commutes. We define functors $\mathsf{L} : \mathscr{C} \to \mathscr{C}^{\mathsf{T}}, \mathsf{R} : \mathscr{C}^{\mathsf{T}} \to \mathscr{C}$ by

$$\begin{aligned}
\mathsf{L}\ \mathsf{F} &= (m \mapsto \mathsf{T}_m\ \mathsf{F}, (m,n) \mapsto \mu_{m,n}\ \mathsf{F}) & \mathsf{R}\ (\mathsf{F}, f) &= \mathsf{F}\ \epsilon \\
\mathsf{L}\ (k : \mathsf{F} \to \mathsf{G}) &= \mathsf{T}_m\ k : \mathsf{T}_m\ \mathsf{F} \to \mathsf{T}_m\ \mathsf{G} & \mathsf{R}\ (k : (\mathsf{F}, f) \to (\mathsf{G}, g)) &= k\ \epsilon : \mathsf{F}\ \epsilon \to \mathsf{G}\ \epsilon
\end{aligned}$$

One may straightforwardly verify that this gives rise to an adjunction $\mathsf{L} \dashv \mathsf{R}$, with unit $\eta\ \mathsf{F} : \mathsf{F} \to \mathsf{T}_\epsilon\ \mathsf{F}$, and counit $\epsilon\ (\mathsf{G}, g) = m \mapsto g_{m,\epsilon} : \mathsf{T}_m\ (\mathsf{G}\ \epsilon) \to \mathsf{G}\ m$. Then we define functor $\hat{\mathsf{T}} : \mathscr{M} \to \mathbf{Cat}$ by $\hat{\mathsf{T}} * = \mathscr{C}^{\mathsf{T}}$ and $\hat{\mathsf{T}}\ (p : * \to *) : \mathscr{C}^{\mathsf{T}} \to \mathscr{C}^{\mathsf{T}}$ with

$$\begin{aligned}
\hat{\mathsf{T}}\ p\ (\mathsf{F}, f) &= (n \mapsto \mathsf{F}\ (n \otimes p), (m,n) \mapsto f_{m,n\otimes p}) \\
\hat{\mathsf{T}}\ p\ (k : (\mathsf{F}, f) \to (\mathsf{G}, g)) &= n \mapsto k\ (np) : \mathsf{F}\ (n \otimes p) \to \mathsf{G}\ (n \otimes p)
\end{aligned}$$

Again, one may verify that $\hat{\mathsf{T}}$ is a strict M-action on $\mathscr{C}^{\mathsf{T}}$, and that the transfer of $\hat{\mathsf{T}}$ along $\mathsf{L} \dashv \mathsf{R}$ coincides with T.

In fact, this is all a generalisation of the case for ordinary monads. When $M = \{*\}$, the trivial monoid, then M-graded monad T reduces to an ordinary monad, and $\mathscr{C}^{\mathsf{T}}$ to the Eilenberg–Moore category for T mentioned in Section 2. The transfer of any monad V on $\mathscr{D}$ over an adjunction $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \to \mathscr{D}$ yields a monad $\mathsf{R} \circ \mathsf{V} \circ \mathsf{L}$ on $\mathscr{C}$. In particular, a strict M-action is simply the identity monad Id, and the transfer of Id over $\mathsf{L} \dashv \mathsf{R}$ yields monad $\mathsf{R} \circ \mathsf{Id} \circ \mathsf{L} = \mathsf{R} \circ \mathsf{L}$.

B A PROTOTYPE IMPLEMENTATION

In this appendix, we present a prototype implementation in Haskell of the categorical constructions presented in the paper. Our goal is not only to show that the theory translates elegantly into an implementation, but also to outline some of the design decisions that need to be made.

B.1 Bags

For simplicity, we represent bags by lists; equality should be taken up to permutations.

```
newtype Bag a = Bag {elements :: [a]}
  deriving (Functor, Monad)
```

We want Bag to be an instance of *Functor*, so that we can use *fmap*, and of *Monad*, so that we can use bag comprehensions; we adopt the list instances of these two classes.

The empty bag and bag union are also defined in terms of the empty list and concatenation:

```
∅ :: Bag a
∅ = Bag [ ]
(⊎) :: Bag a → Bag a → Bag a
x ⊎ y = Bag (elements x ++ elements y)
```

B.2 Commutative Monoids

For aggregations, we require a type class representing monoids.

```
class Monoid m where
  ϵ   :: m
  (⊗) :: m → m → m
    -- ⊗ should be associative, and ϵ its neutral element
```

(The only way to express the laws in standard Haskell is by comments.) Commutative monoids are monoids in which ⊗ is additionally commutative; the class *CMonoid* has no more methods, but imposes an extra law.

```
class Monoid m ⇒ CMonoid m
    -- ⊗ should be commutative
```

Bags are obviously a commutative monoid:

```
instance Monoid (Bag a) where
  ϵ   = ∅
  (⊗) = (⊎)
instance CMonoid (Bag a)
```

We can reduce a bag whose elements are drawn from a commutative monoid:

```
reduceBag :: CMonoid m ⇒ Bag m → m
reduceBag = foldr (⊗) ϵ · elements
```

(Note that the target monoid is determined implicitly from its carrier *m*.) For example, filtering a bag by a predicate is a reduction:

```
filter :: (a → 𝔹) → Bag a → Bag a
filter p = reduceBag · fmap (guard p)
  where guard p a = if p a then return a else ∅
```

B.3 Pointed Sets

We use a type class as an interface to pointed sets, with the usual provisos: we can use the same name *null* for all base points, but a given type can be pointed in at most one way. (We use *null* for the point, and rename to *isNil* the standard Haskell function *null* :: [*a*] → 𝔹 that tests for an empty list.)

```
class Pointed a where
  null   :: a
  isNull :: a → 𝔹    -- is the argument null?
instance Pointed () where
  null = ()
  isNull () = True
instance (Pointed a, Pointed b) ⇒ Pointed (a, b) where
  null = (null, null)
  isNull (a, b) = isNull a ∧ isNull b
instance Pointed (Bag a) where
  null = ∅
  isNull = isNil · elements
```

```
class (Functor (Map k)) ⇒ Key k where
  data Map k :: ∗ → ∗
  empty     :: (Pointed v) ⇒ Map k v
  isEmpty   :: (Pointed v) ⇒ Map k v → 𝔹
  single    :: (Pointed v) ⇒ (k, v) → Map k v
  merge     :: (Map k v1, Map k v2) → Map k (v1, v2)
  merge°    :: Map k (v1, v2) → (Map k v1, Map k v2)
  merge° x = (fmap fst x, fmap snd x)
  dom       :: (Pointed v) ⇒ Map k v → Bag k
  cod       :: Pointed v ⇒ Map k v → Bag v
  cod t     = reduce (fmap return t)
  lookup    :: Map k v → (k → v)
  ix        :: Bag (k, v) → Map k (Bag v)
  ix°       :: Map k (Bag v) → Bag (k, v)
  reduce    :: (Pointed v, CMonoid v) ⇒ Map k v → v
  reduce    = reduceBag · cod
```

Fig. 6. Declaration of finite maps

We have added an explicit test for nullness; an alternative design would have been to omit the *isNull* method, and assume an *Eq* type constraint instead, so that we could simply compare with *null*.

Some types have an obvious base point. To add a base point to an arbitrary type, we can use Haskell's Maybe.

```
instance Pointed (Maybe a) where
  null = Nothing
  isNull (Just _) = False
  isNull Nothing = True
```

B.4 Finite Maps

We will capture finite maps datatype-generically, following Hinze [2000], so we start by identifying suitable key types. Note that the two type arguments K and V of Map K V have different status: finite maps should be functorial in the parameter V, but K is instead a *type index* or p*hantom type* [Hinze 2003].

The type class *Key* shown in Figure 6 sets out the requirements on a type K for it to be a suitable key type. The associated type constructor Map K is then determined by K; note the *Functor* class context. The methods *reduce* and *cod* are interdefinable, so any instance need only define one of these two methods; and there is a default definition for *merge*°. For simplicity, we have defined *dom* to yield a bag, so we don't have to provide a separate datatype of sets.

We have not included an *insert* operation, as it is asymmetric—it is not clear what it should do for unit keys. But we could assemble an insertion operation using *single* and *merge*:

```
insert (k, v) t = fmap first (single (k, v) `merge` t)
```

where

```
first :: (Pointed a) ⇒ (a, a) → a
first (a, b) = if isNull a then b else a
```

```
instance Key () where    -- unit type
  newtype Map () v = Lone v
  empty                    = Lone null
  isEmpty (Lone v)         = isNull v
  single ((), v)           = Lone v
  merge (Lone v₁, Lone v₂) = Lone (v₁, v₂)
  dom (Lone v)             = ⦃ () | not (isNull v) ⦄
  cod (Lone v)             = ⦃ v | not (isNull v) ⦄
  lookup (Lone v) ()       = v
  ix kvs                   = Lone (fmap snd kvs)
  ix° (Lone vs)            = fmap (λv → ((), v)) vs
instance Functor (Map ()) where
  fmap f (Lone v)          = Lone (f v)

instance Key Word16 where    -- constant types
  newtype Map Word16 v = A (Array Word16 v)
  empty           = A (accumArray (curry snd) null (0, 2^16 − 1) [ ])
  isEmpty (A a)   = all isNull (elems a)
  single (k, v)   = A (accumArray (curry snd) null (0, 2^16 − 1) [(k, v)])
  merge (A a, A b) = A (listArray (0, 2^16 − 1) (zip (elems a) (elems b)))
  dom (A a)       = ⦃ k | (k, v) ← assocs a, not (isNull v) ⦄
  cod (A a)       = ⦃ v | (k, v) ← assocs a, not (isNull v) ⦄
  lookup (A a) k  = a ! k
  ix kvs          = A (accumArray (λxs x → Bag (x : elements xs)) ∅ (0, 2^16 − 1)
                                  (elements kvs))
  ix° (A a)       = ⦃ (k, v) | (k, vs) ← assocs a, v ← elements vs ⦄
instance Functor (Map Word16) where
  fmap f (A a)             = A (fmap f a)
```

Fig. 7. Datatype-generic implementation of finite maps for polynomial key types (first part)

Polynomial types are all suitable as keys. We build up the instances inductively, using the 'laws of exponents'. The code is shown in Figure 7. We start with the unit type (); the associated type instance Map () indicates that finite maps from the unit type are singletons, and the domain and codomain are either empty or a singleton, depending on whether or not the single element is *null*. We also cover the constant type *Word16* = $\{0 \,..\, 2^{16} - 1\}$; finite maps from bounded naturals are just arrays. The sum of two *Key* types is again a *Key* type; finite maps from a sum are pairs of finite maps. The product of two *Key* types is again a *Key* type; finite maps from a product are finite maps to finite maps. For the latter, we require finite maps to pointed types to be themselves pointed:

```
instance (Key k, Pointed v) ⇒ Pointed (Map k v) where
  null   = empty
  isNull = isEmpty
```

From these four components we can induce key instances for more interesting types, such as strings and dates.

```
instance (Key $k_1$, Key $k_2$) ⇒ Key (Either $k_1$ $k_2$) where    -- sum types
  newtype Map (Either $k_1$ $k_2$) v = Pair (Map $k_1$ v, Map $k_2$ v)
  empty                              = Pair (empty, empty)
  isEmpty (Pair ($t_1$, $t_2$))            = isEmpty $t_1$ ∧ isEmpty $t_2$
  single (Left $k_1$, v)               = Pair (single ($k_1$, v), empty)
  single (Right $k_2$, v)              = Pair (empty, single ($k_2$, v))
  merge (Pair ($t_1$, $t_2$), Pair ($u_1$, $u_2$)) = Pair (merge ($t_1$, $u_1$), merge ($t_2$, $u_2$))
  dom (Pair ($t_1$, $t_2$))                = fmap Left (dom $t_1$) ⊎ fmap Right (dom $t_2$)
  cod (Pair ($t_1$, $t_2$))                = cod $t_1$ ⊎ cod $t_2$
  lookup (Pair ($t_1$, $t_2$)) (Left $k_1$)    = lookup $t_1$ $k_1$
  lookup (Pair ($t_1$, $t_2$)) (Right $k_2$)   = lookup $t_2$ $k_2$
  ix kvs                             = Pair (ix (fmap (λ(Left $k_1$, v) → ($k_1$, v)) kvs),
                                             ix (fmap (λ(Right $k_2$, v) → ($k_2$, v)) kvs))
  $ix^\circ$ (Pair ($t_1$, $t_2$))                = fmap (λ($k_1$, v) → (Left $k_1$, v)) ($ix^\circ$ $t_1$) ⊎
                                       fmap (λ($k_2$, v) → (Right $k_2$, v)) ($ix^\circ$ $t_2$)
instance (Functor (Map $k_1$), Functor (Map $k_2$)) ⇒ Functor (Map (Either $k_1$ $k_2$)) where
  fmap f (Pair ($t_1$, $t_2$))             = Pair (fmap f $t_1$, fmap f $t_2$)
instance (Key $k_1$, Key $k_2$) ⇒ Key ($k_1$, $k_2$) where    -- product types
  newtype Map ($k_1$, $k_2$) v = Comp (Map $k_1$ (Map $k_2$ v))
  empty                      = Comp empty
  isEmpty (Comp t)           = isEmpty t
  single (($k_1$, $k_2$), v)         = Comp (single ($k_1$, single ($k_2$, v)))
  merge (Comp $t_1$, Comp $t_2$)     = Comp (fmap merge (merge ($t_1$, $t_2$)))
  dom (Comp t)               = $ix^\circ$ (fmap dom t)
  cod (Comp t)               = reduce (fmap cod t)
  lookup (Comp t) ($k_1$, $k_2$)     = lookup (lookup t $k_1$) $k_2$
  ix kvs                     = Comp (fmap ix (ix (fmap assoc kvs)))
  $ix^\circ$ (Comp $t_1$)             = fmap $assoc^\circ$ ($ix^\circ$ (fmap $ix^\circ$ $t_1$))
instance (Functor (Map $k_1$), Functor (Map $k_2$)) ⇒ Functor (Map ($k_1$, $k_2$)) where
  fmap f (Comp t)            = Comp (fmap (fmap f) t)
```

Fig. 7. Datatype-generic implementation of finite maps for polynomial key types (continued)

B.5 Example

Now we can implement the example from Section 1. Supposing record types for the *Customer* and *Invoice* tables:

```
data Customer = C { cid :: Identifier, name :: Name }
data Invoice  = I { iid :: Identifier, cust :: Identifier, due :: Date, amount :: Amount }
```

then the query can be implemented by:

```
example :: Bag Customer → Bag Invoice → Bag (Bag Name, Bag Amount)
example cs is =
  fmap (pair (fmap name, fmap amount)) (cod
    (fmap (pair (id, filter ((<today) · due))) (merge (cs ‘indexBy‘ cid, is ‘indexBy‘ cust))))
  where pair (f, g) (a, b) = (f a, g b)
```

(Recall that *indexBy* was defined in terms of *ix*

$$s\ \text{`}indexBy\text{`}\ f = ix\ (fmap\ (f \vartriangle id)\ s)$$

in Section 5.) So with inputs

$$\begin{array}{l}
customers :: \mathrm{Bag}\ Customer \\
customers = \wr\ C\ 101\ \texttt{"sam"}, C\ 102\ \texttt{"max"}, C\ 103\ \texttt{"pat"}\ \int \\
invoices :: \mathrm{Bag}\ Invoice \\
invoices\ \ = \wr\ I\ 201\ 101\ 20160921\ 20, I\ 202\ 101\ 20160316\ 15, I\ 203\ 103\ 20160520\ 10\ \int \\
today :: Date \\
today = 20160919
\end{array}$$

we get

$$reduceBag\ (fmap\ cp\ (example\ customers\ invoices)) = \wr(\texttt{"sam"}, 15), (\texttt{"pat"}, 10)\ \int$$

where $cp\ (x, y) = \wr(a, b) \mid a \leftarrow x, b \leftarrow y\int$.