The Programming of Algebra

Henglein, Fritz; Kaarsgaard, Robin; Mathiesen, Mikkel Kragh

Published in:
Proceedings of the Ninth Workshop on Mathematically Structured Functional Programming

DOI:
10.4204/EPTCS.360.4

Publication date:
2022

Document version:
Final published version

The Programming of Algebra

Fritz Henglein
DIKU, Department of Computer Science, University of Copenhagen

Robin Kaarsgaard
School of Informatics, University of Edinburgh

Mikkel Kragh Mathiesen
DIKU, Department of Computer Science, University of Copenhagen

We present module theory and linear maps as a powerful generalised and computationally efficient framework for the relational data model, which underpins today's relational database systems. Based on universal constructions of modules we obtain compact and computationally efficient data structures for data collections corresponding to union and deletion, repeated union, Cartesian product and key-indexed data. Free modules naturally give rise to *polysets*, which generalise multisets and facilitate expressing database queries as multilinear maps with asymptotically efficient evaluation on polyset constructors. We introduce *compact maps* as a way of representing infinite (poly)sets constructible from an infinite base set and its elements by addition and subtraction. We show how natural joins generalise to algebraic joins, while intersection is implemented by a novel algorithm on *nested* compact maps that carefully avoids visiting parts of the input that do not contribute to the eventual output. Our algebraic framework leads to a *worst-case optimal* evaluation of cyclic relational queries, which is known to be impossible using textbook query optimisers that operate on lists of records only.

1 Introduction

Query languages, like any other programming language, should be

- *efficient*, in that they should produce query results as efficiently as possible in terms of time, space and energy consumed;
- *expressive*, in that they should admit expressing solutions to many problems; and
- *reasonable*, in that they should provide reasoning principles about a query's semantics to support compositional checking of functional and security properties, correctness of transformations and optimisations, query synthesis and more.

These aspects are sometimes presented as irreconcilable, suggesting, for example, that expressivity and reasonableness necessarily come at the cost of efficiency; or they may be studied in isolation of each other (expressiveness and correctness "modulo" efficiency, or efficient query engine implementation with specialised functionality "modulo" concerns what this does to the validity of reasoning principles in the core language). We propose that efficiency, expressiveness and reasoning can mutually support each other via *structure (algebraic and categorical)*.

Intuitively, our approach consists of identifying purely algebraic operations that can be implemented by symbolic simplification that exploits their algebraic properties for run-time efficiency, and factoring expensive exception and equality checking into separate operations that are invoked explicitly (e.g. checking whether a set is empty before deleting from it or whether two elements are equal or not) *only when* this is required, rather than melding the two operations together into a single operation. To give meaning to the result of an exception-less deletion operation we introduce a data model that encompasses and goes beyond sets and multisets: *polysets*, whose elements carry any integral multiplicity, not only 1 (sets) or positive integers (multisets). Indeed, multiplicities can be replaced by arbitrary rings and even algebras.

Jeremy Gibbons and Max S. New (Eds.): Ninth Workshop on Mathematically Structured Functional Programming (MSFP 2022)
EPTCS 360, 2022, pp. 71–92, doi:10.4204/EPTCS.360.4

We recognise finite polysets as the elements of the free module generated by a (usually infinite) set X over $\mathbb{Z}$. More generally, we show that *modules* over a commutative ring permit a variety of constructions that capture the core operations in query processing, including aggregation. They are all characterised by *universal properties*, which not only justify singling them out as natural primitive operations, but also provide algebraic rewrite rules for run-time efficiency. Overall, our approach of building terms through free structures and then interpreting them using their universal properties is not unlike that taken by algebraic effects (see, e.g., [23]).

Our approach is based on using free modules to represent generalised relations, biproducts for records, copowers for finite maps, tensor products for relational Cartesian products, and compact maps for infinite collections with wildcards. All of these constructions use arbitrary sets and modules to form new modules. We show that operations such as selection, projection, union, and intersection (and more) are not only very natural to express and reason about, but also lead us to highly efficient implementations of operations such as relational Cartesian products and, most significantly, arbitrary natural and outer joins. In the case of joins, the asymptotic efficiency attained is not even achievable using conventional relational query optimisation technology.

Although universal properties characterise these constructions uniquely *extensionally*, some *data structure representations* are better than others. A particular strength of our approach is how it uses prolific symbolic operator representations at run time to avoid unnecessary costly normalisation to a standard data structure, unless a particular context makes it absolutely necessary. For example, representing a tensor product symbolically gives a quadratic space compression and speed-up compared to normalising it to a list of atomic pairs; representing scalar multiplication symbolically eliminates iterated adding; representing additions symbolically facilitates operating on the summands independently, without first turning the tree of additions into a list of leaves. More generally, this also applies to scalar multiplication so that the implementation of a linear map boils down to *folding*, the composition of mapping the leaf elements to the target module and interpreting symbolic scalar multiplication and addition in the target module (which may also use symbolic scalar multiplication and addition).

Efficient evaluation hinges on the algebraic operations and their properties in a crucial way. While lazy evaluation only delays standard evaluation, but does not avoid normalisation to a standard data structure, unnormalised representations using symbolic constructors let our evaluator invoke specialised evaluation rules depending on the particular way data are used; for example, computing the weight # of a polyset (corresponding to the cardinality of a multiset) contains the clauses $\#(s_1 \otimes s_2) = \#s_1 \cdot \#s_2$ and $\#(t_1 + t_2) = \#t_1 + \#t_2$. In particular, the tensor product $\otimes$ need not first be multiplied out to a quadratically bigger set of pairs of elements from s_1 and s_2. Since the output of a join consists of a data dependent number of tensor products, this is not straightforwardly achievable by static preprocessing of a query.

Selective simplification is explicitly indicated and forced by efficient implementations of explicit natural isomorphisms to modules that correspond to data structures for efficient associative access. This is where tensor products, copowers/finite maps and in particular compact maps, an algebraic generalisation of maps with a non-zero default value, come into play.

Implementation. These techniques are very general and can be applied and implemented in many ways, including as a library in a functional or logic language, a design pattern, or as a domain-specific language in its own right. In this paper, we include an implementation in Haskell using type families to achieve both extensibility and efficiency. All of the examples in this paper can be found in the Haskell source code.

The main contributions of this paper are:

- An algebraic framework, based on universal constructions of modules and linear maps, into which generalised databases and queries can not only be interpreted, but evaluated highly efficiently.
- An algebraic product, formally a (weighted, unital) commutative algebra, to express intersection on values and, via embedding of finite collections into the corresponding compact collections, of natural joins, which in this case turn out to yield *worst-case optimal* query evaluation. This is illustrated for triangle queries, the canonical example of cyclic queries, which have only asymptotically suboptimal query implementations via classical relational query optimisation. (The proof of worst-case optimality for algebraic joins, which include and extend relational joins, is beyond the scope of this paper.)
- Efficient evaluation of expressions (queries) that leverages generic trie implementations of compact maps for efficient lookup, carefully scheduled enumeration for products (intersection and thus join), and aggressively exploiting algebraic simplifications. In particular, all relevant operations of and on modules are purely symbolic when constructed, and only simplified according to the *specific* context in which they are used.
- An implementation of all of the above in Haskell.

This paper is structured as follows. Section 2 defines the relevant algebraic structures, shows how to build modules and lists the most important isomorphisms. Section 3 shows how linear maps can be used as an alternative to relational algebra. Section 4 explains the simple, yet efficient approach to simplifying module terms.

2 Mathematical Preliminaries

We assume familiarity with fundamental (linear) algebraic structures including rings, modules, and (bi)linear maps, and refer the unfamiliar reader to the appendix. We recall here the definition of a (weighted, unital, commutative) algebra.

Definition 1 (Algebra)**.** *A* commutative algebra *S over commutative ring K is a commutative ring which is also a module over K. The addition and multiplication operations in the ring S must coincide with those of the modules structure in their common domain. When the algebra multiplication has a unit e satisfying $e \cdot v = v \cdot e = v$ for all v of S, we say that the algebra is* unital*. If additionally there is a linear map $\# : S \to K$ then S is a* weighted algebra*.*

For an element x of a weighted algebra $\#x$ is pronounced the *weight* of x. As a proviso, we will henceforth leave out "commutative"; all of our rings and algebras will be commutative.

2.1 Meet the Modules

We now present various useful ways of constructing modules. The presentation is guided by category theory, but we will not introduce it explicitly. Rather, our focus is on *universal properties*, a system for abstractly defining objects as the most general solution given some condition.

Trivial modules. The simplest module is the zero module $\mathbf{0}$ consisting of just a single 0 element. All operations are trivial and determined by the axioms. It satisfies the following universal property: for all modules U there is a unique linear map $0 : \mathbf{0} \to U$, and a unique linear map $0 : U \to \mathbf{0}$.

The first interesting example of a module is the ring K itself with operations inherited from the ring structure. This is called the *scalar* module. It also has a weighted algebra structure with multiplication inherited from the ring and weight given by the identity map. The scalars satisfy the universal property that for any module U and element $u : U$ there exists a unique linear map $f : K \to U$ such that $f(1) = u$.

Intuitively the property states that any linear map from K is completely determined by what it does to 1. We are therefore justified in defining linear maps by pattern matching on 1, for example defining some $f : K \to K$ by $f(1) = 42$. By linearity $f(r) = f(r \cdot 1) = r \cdot f(1) = r \cdot 42$ for any r, so we have exactly the information we need to determine the value of f at any point.

Free modules. The *free module* over a set A, denoted $\mathbf{F}_K[A]$, is the module generated by elements of A. By *generated* we mean that for every element $a : A$ there is an element $\langle a \rangle : \mathbf{F}_K[A]$ such that the set $\{\langle a \rangle\}_{a \in A}$ of all such elements form an orthonormal basis for $\mathbf{F}_K[A]$. Furthermore, $\mathbf{F}_K[A]$ contains 0 and is closed under addition and scalar multiplication. For instance, there are elements like $3 \cdot \langle a \rangle + 5 \cdot \langle b \rangle$. Two elements of $\mathbf{F}_K[A]$ are equal if and only if they are forced to be equal by the module axioms and properties of K. Hence, $3 \cdot \langle a \rangle + 5 \cdot \langle b \rangle = 5 \cdot \langle b \rangle + 3 \cdot \langle a \rangle$ by commutativity, but $\langle a \rangle \neq 0$ (whenever $1 \neq 0$ in K) since there is no way to show $\langle a \rangle = 0$ from the axioms alone (a formal argument for this is surprisingly tricky, though). In particular $\langle \cdot \rangle$ is injective so $\langle a \rangle = \langle b \rangle$ implies $a = b$.

An element of $\mathbf{F}_K[A]$ is best thought of as a generalised finite multiset. Any such element can be written uniquely as a basis expansion $\sum_{a:A}(r_a \cdot \langle a \rangle)$ where the number of non-zero r_a is finite. Depending on the nature of K there is a different interpretation of what 'generalised finite multiset' means.

- When K is $\mathbb{F}_2$ elements $\mathbf{F}_K[A]$ are finite sets. A set like $\{a, b, c\}$ is written as $\langle a \rangle + \langle b \rangle + \langle c \rangle$.
- When K is $\mathbb{Z}$, elements of $\mathbf{F}_K[A]$ are finite *polysets*. A polyset like $\{a^3, b^{-2}, c^5\}$ is written as $3\langle a \rangle - 2\langle b \rangle + 5\langle c \rangle$.
- When K is $\mathbb{R}$ elements of $\mathbf{F}_K[A]$ are generalised finite fuzzy sets, whose membership function is not limited to $[0, 1]$. A fuzzy set like $\{a/0.3, b/0.2, c/0.5\}$ is written as $0.3\langle a \rangle + 0.2\langle b \rangle + 0.5\langle c \rangle$.

The free module $\mathbf{F}_K[A]$ satisfies the following universal property: for any module V and function $f : A \to V$ there is a unique linear map $\widehat{f} : \mathbf{F}_K[A] \to V$ such that $\widehat{f} \circ \langle \cdot \rangle = f$. In essence, to define a *linear map* out of $\mathbf{F}_K[A]$ it suffices to identify the target module and define a *map* out of A, and this map can be chosen *freely*. We can think of this as definition by pattern matching, and write linear maps like

$$\begin{aligned} g &: \mathbf{F}_K[\mathbf{Str}] \to \mathbf{F}_K[\mathbf{Str}] \\ g(\langle s \rangle) &= \langle \mathbf{reverse}(s) \rangle \end{aligned}$$

In this case $g = \widehat{f}$ where $f(s) = \langle \mathbf{reverse}(s) \rangle$. Note that due to linearity we also get the equations

$$g(0) = 0 \qquad g(x+y) = g(x) + g(y) \qquad g(r \cdot x) = r \cdot g(x)$$

but we do not need to handle these cases as they are forced by the condition of linearity. Thus, we get to treat $\mathbf{F}_K[A]$ as an inductive type and 'pretend' that it only contains elements of A, even though it does contain many more elements than that.

Finally, we are going to equip $\mathbf{F}_K[A]$ with a bilinear operator and a weight function to make it into a weighted algebra. There is more than one possible choice, but only one makes sense for our purposes:

$$\begin{aligned} \langle a \rangle \cdot \langle a \rangle &= \langle a \rangle \\ \langle a \rangle \cdot \langle b \rangle &= 0 \quad (\text{for } a \neq b) \\ \#\langle a \rangle &= 1 \end{aligned}$$

Note that we are defining multiplication by pattern matching in each argument, implicitly appealing to the universal property of $\mathbf{F}_K[A]$ twice. This operation can be thought of as a variant of the Kronecker delta function: If the arguments a, b are equal, we return that unique value as a singleton; if they are unequal, we 'fail' by returning 0. When applied to sets it computes the set intersection; for multisets and polysets, however, the multiplicities of common elements are multiplied. For instance, for

$$(3\langle a\rangle + 2\langle b\rangle + 5\langle c\rangle) \cdot (7\langle b\rangle + 4\langle c\rangle + 2\langle d\rangle) = (2\cdot 7)\langle b\rangle + (5\cdot 4)\langle c\rangle = 14\langle b\rangle + 20\langle c\rangle$$

The missing component in $\mathbf{F}_K[A]$ being a *unital* weighted algebra is the unit, an element 1 such that $1 \cdot x = x = x \cdot 1$. If A is finite we can take $1 = \sum_{a:A} \langle a\rangle$, but for infinite A this sum is not well-defined. Thus, we now proceed to show how $\mathbf{F}_K[A]$ can be extended to account for this deficiency, which in turn paves the way to efficient representation of certain infinite sets and—eventually—efficient algebraic join computations.

Compact free. We have seen that the free module over a set A does an excellent job of representing finite subsets of A. However, it lacks the ability to represent complements and in particular there is no general way to represent the subset containing every inhabitant of A. Algebraically, the multiplication on $\mathbf{F}_K[A]$ does not have a unit element for infinite A.

To rectify these deficiencies we introduce the *compact free module*, $\mathbf{F}^*_K[A]$, constructed by taking the free module $\mathbf{F}_K[A]$ and adjoining a distinct element 1. We think of 1 as *symbolising* the potentially infinite sum $\sum_{a:A}\langle a\rangle$. However, even when A is finite, 1 is by definition distinct from $\sum_{a:A}\langle a\rangle$.

The choice of never identifying 1 with $\sum_{a:A}\langle a\rangle$ has several advantages. We do not have to know whether A is finite or infinite to decide if 1 is linearly independent from the other generators. It allows a compact symbolic representation of this sum when A is finite but large. And finally it gives us the following universal property: for any module V together with a map $f : A \to V$ and an element $u : V$ there is a unique linear map $\widehat{f} : \mathbf{F}^*_K[A] \to V$ such that $\widehat{f} \circ \langle\cdot\rangle = f$ and $\widehat{f}(1) = u$.

In terms of pattern matching this amounts to having cases for $\langle a\rangle$ and 1. With this addition $\mathbf{F}^*_K[A]$ has a unital weighted algebra structure, where multiplication of generators works just like for $\mathbf{F}_K[A]$ and 1 is the multiplicative unit. Explicitly:

$$\begin{array}{lll} \langle a\rangle \cdot \langle a\rangle = \langle a\rangle & 1 \cdot y = y & \#\langle a\rangle = 1 \\ \langle a\rangle \cdot \langle b\rangle = 0 \quad (\text{for } a \neq b) & x \cdot 1 = x & \#1 = 1 \end{array}$$

This enables us to represent not just finite sets, but also *cofinite* sets: the subsets of A that contain all but a finite number of elements. For example, the set $A \setminus \{a, b\}$ is written as $1 - (\langle a\rangle + \langle b\rangle)$. When we introduce tensor products later we will see how even more interesting subsets can be represented compactly in this manner.

Biproduct. The *biproduct* of modules U and V is a module $U \oplus V$ consisting of pairs of elements from U and V with operations defined pointwise. For instance given $(u_1, v_1), (u_2, v_2) : U \oplus V$ we define

$$(u_1, v_1) + (u_2, v_2) = (u_1 + u_2, v_1 + v_2)$$

Categorically speaking, the biproduct is both a binary product and a binary coproduct such that injection and projection is compatible. For details see Appendix A.3.

Finite map. Suppose we are given a set A and a module U. The *finite map* module $A \Rightarrow U$ consists of maps $A \to U$ with *finite support*, i.e. maps which produce a non-zero value at finitely many elements of A. It is the module generated by elements of the form $a \mapsto u$ where $a : A$ and $u : U$, subject to the requirement that $a \mapsto \cdot$ is a linear map for any a. Any finite map can be written as $\sum_{a:A}(a \mapsto u_a)$ where $u_a = 0$ for all but finitely many a.

The finite map module satisfies the following universal property: for any module V together with a family of maps $f_a : U \to V$ there exists a unique linear map $\mathbf{case}(f) : (A \Rightarrow U) \to V$ such that $\mathbf{case}(f) \circ (a \mapsto \cdot) = f_a$ for all $a : A$. This allows pattern matching similar to the free module. For example:

$$\begin{aligned} f &: (A \Rightarrow K) \to \mathbf{F}_K[A] \\ f(a \mapsto 1) &= \langle a \rangle \end{aligned}$$

Recall that the use of 1 on the left-hand side is pattern matching on scalars.

Intuitively, we think of $A \Rightarrow U$ as elements of U indexed by A. For instance, suppose we have a multiset of strings and we want to index it by string lengths. The indexed space would be $\mathbb{N} \Rightarrow \mathbf{F}_K[\mathbf{Str}]$ with the indexing being done as follows.

$$\begin{aligned} \text{indexbylength} &: \mathbf{F}_K[\mathbf{Str}] \to (\mathbb{N} \Rightarrow \mathbf{F}_K[\mathbf{Str}]) \\ \text{indexbylength}(\langle s \rangle) &= \mathbf{length}(s) \mapsto \langle s \rangle \end{aligned}$$

We can also go in the other direction and forget the index.

$$\begin{aligned} \mathbf{sum} &: (A \Rightarrow U) \to U \\ \mathbf{sum}(a \mapsto u) &= u \end{aligned}$$

Like the free module the finite map module is a weighted algebra, but lacks a unit when the index set is infinite. Multiplication and weight are defined by:

$$\begin{aligned} (a \mapsto u) \cdot (a \mapsto v) &= a \mapsto (u \cdot v) \\ (a \mapsto u) \cdot (b \mapsto v) &= 0 \quad (\text{for } a \neq b) \\ \#(a \mapsto u) &= \#u \end{aligned}$$

Compact map. Finite maps, like free modules, do not possess a unit element when the index set is infinite. More generally, they do not possess constant mappings that have the same (non-zero) value everywhere. The solution is similar: adjoin distinct elements to account for this deficiency.

Thus, the *compact map* module $A \Rightarrow^* U$ is obtained by adding elements of the form $* \mapsto u$ for any $u : U$. The $*$ represents a *wildcard* which matches anything. Intuitively $* \mapsto u$ symbolises $\sum_{a:A}(a \mapsto u)$ but is by definition always distinct, just like for the compact free module.

The compact map module satisfies the following universal property: for any module V together with a family of maps $f_a : U \to V$ as well as a map $f_* : U \to V$ there exists a unique linear map $\mathbf{case}(f) : (A \Rightarrow^* U) \to V$ such that $\mathbf{case}(f) \circ (a \mapsto \cdot) = f_a$ for all $a : A$ and $\mathbf{case}(f) \circ (* \mapsto \cdot) = f_*$.

The unital weighted algebra structure on compact maps is given as follows.

$$\begin{aligned} (a \mapsto u) \cdot (a \mapsto v) &= a \mapsto (u \cdot v) & 1_{A \Rightarrow^* U} &= * \mapsto 1_U \\ (a \mapsto u) \cdot (b \mapsto v) &= 0 \quad (\text{for } a \neq b) & & \\ (a \mapsto u) \cdot (* \mapsto v) &= a \mapsto (u \cdot v) & \#(a \mapsto u) &= \#u \\ (* \mapsto u) \cdot (a \mapsto v) &= a \mapsto (u \cdot v) & \#(* \mapsto u) &= \#u \\ (* \mapsto u) \cdot (* \mapsto v) &= * \mapsto (u \cdot v) & & \end{aligned}$$

Lookup in a compact map works like for finite maps, except $*$ is also a valid key. To see how this works in practice, suppose we have an element:

$$x = (* \mapsto 2) + (a \mapsto 3) + (b \mapsto -2)$$

Consider the following lookups:

$$x(*) = 2 \qquad x(a) = 2+3 = 5 \qquad x(b) = 2-2 = 0 \qquad x(c) = 2$$

It is evident that the $* \mapsto 2$ component of x serves as a *baseline* and a component like $a \mapsto 3$ determines how much the value at *a deviates* from that baseline. In particular, the value at b deviates by exactly the negative of the baseline value, the result being that b is contained 0 times in x when viewed as a polyset. This is in contrast to the more common construct of having finite maps with a *default value* for keys not explicitly listed. In particular, compact map addition is commutative: it does not matter in which order the maps are listed.

Tensor product. The *tensor product* of modules U and V is a module $U \otimes V$ generated by elements of the form $u \otimes v$ with $u : U$ and $v : V$, subject to the requirement that $\otimes$ is bilinear, i.e. linear in each argument separately. More explicitly, linearity in the first argument requires

$$0 \otimes v = 0 \qquad (u_1 + u_2) \otimes v = u_1 \otimes v + u_2 \otimes v \qquad (r \cdot u) \otimes v = r \cdot (u \otimes v)$$

Linearity in the second argument is analogous. In general, any element of the tensor product can be written as a sum of $\otimes$-pairs, i.e. $\sum_i (u_i \otimes v_i)$. Compare this with the biproduct, which is also generated by pairs of elements. An element $\sum_i (u_i, v_i)$ can always be reduced to $(\sum_i u_i, \sum_i v_i)$, since the biproduct pair constructor is linear in both arguments *simultaneously*. The tensor product is linear in each argument *separately* and elements can only be simplified in some circumstances, such as

$$u_1 \otimes v_1 + u_1 \otimes v_2 + u_2 \otimes v_1 + u_2 \otimes v_2 = (u_1 + u_2) \otimes (v_1 + v_2)$$

This kind of simplification can often reduce the size of a term from quadratic to linear, but recognising such opportunities is not easy. Consequently, when a term is in this kind of simplified, compact form we should not expand it unless absolutely necessary!

The tensor product satisfies the following universal property: for any module W and bilinear map $f : U \times V \to W$ there exists a unique linear map $g : U \otimes V \to W$ such that $g \circ \otimes = f$.

The consequence is that any bilinear map can be written as a linear map from the tensor product. It also justifies defining linear maps by pattern matching on $\otimes$, for example:

$$\begin{aligned} f &: U \otimes V \to V \otimes U \\ f(u \otimes v) &= v \otimes u \end{aligned}$$

This is only valid if the definition is linear in each argument separately. In particular we cannot simply define a projection $\pi_1 : U \otimes V \to U$ as $\pi_1(u \otimes v) = u$, as that would not be linear in the second argument. In this respect the tensor product is different from an ordinary product type. If V is equipped with a weighted algebra structure, however, we can define:

$$\begin{aligned} \pi_1 &: U \otimes V \to U \\ \pi_1(u \otimes v) &= \#v \cdot u \end{aligned}$$

$$\begin{aligned}
\mathbf{F}_K[0] &\cong \mathbf{0} & \mathbf{F}_K[1] &\cong K \\
0 \Rightarrow U &\cong \mathbf{0} & 1 \Rightarrow U &\cong U \\
A \Rightarrow \mathbf{0} &\cong \mathbf{0} & A \Rightarrow K &\cong \mathbf{F}_K[A] \\
\mathbf{F}_K[A+B] &\cong \mathbf{F}_K[A] \oplus \mathbf{F}_K[B] & \mathbf{F}_K[A \times B] &\cong \mathbf{F}_K[A] \otimes \mathbf{F}_K[B] \\
(A+B) \Rightarrow U &\cong (A \Rightarrow U) \oplus (B \Rightarrow U) & (A \times B) \Rightarrow U &\cong A \Rightarrow B \Rightarrow U \\
A \Rightarrow (U \oplus V) &\cong (A \Rightarrow U) \oplus (A \Rightarrow V) & A \Rightarrow (U \otimes V) &\cong (A \Rightarrow U) \otimes V \\
A \Rightarrow U &\cong \mathbf{F}_K[A] \otimes U & A \Rightarrow^* U &\cong \mathbf{F}_K^*[A] \otimes U \\
\mathbf{F}_K^*[A] &\cong \mathbf{F}_K[A] \oplus K & A \Rightarrow^* U &\cong (A \Rightarrow U) \oplus A
\end{aligned}$$

Table 1: A selection of natural isomorphisms

The weight serves to dispose of data, which is not possible for an arbitrary module in general. Finally, we equip the tensor product with an algebra structure. Suppose U and V are algebras. All operations are defined pointwise:

$$\begin{aligned}
(u_1 \otimes v_1) \cdot (u_2 \otimes v_2) &= (u_1 \cdot u_2) \otimes (v_1 \cdot v_2) \\
1_{U \otimes V} = 1_U \otimes 1_V \qquad & \#(u \otimes v) = \#u \cdot \#v
\end{aligned}$$

Note in particular how computing the weight of a tensor product simply reduces to computing the weight of each factor. This saves us from having to expand $u \otimes v$ at all. When u and v are themselves large sums this saves a considerable amount of work.

2.2 Functors and isomorphisms

All of the constructions we have seen are *functors*, structure-preserving maps between categories. Briefly, a functor acts not just on objects (sets, modules, etc.), but also on maps between objects. For instance $\mathbf{F}_K[A]$ is functorial in A, so given any map between sets $f : A \to B$ we have $\mathbf{F}_K[f] : \mathbf{F}_K[A] \to \mathbf{F}_K[B]$.

Let f be a map between sets and α, β be linear maps. The functors act as follows:

$$\begin{aligned}
\mathbf{F}_K[f](\langle a \rangle) &= \langle f(a) \rangle & \mathbf{F}_K^*[f](\langle a \rangle) &= \langle f(a) \rangle \\
\mathbf{F}_K^*[f](1) &= 1 & (\alpha \otimes \beta)(u \otimes v) &= \alpha(u) \otimes \beta(v) \\
(f \Rightarrow^* \alpha)(* \mapsto u) &= * \mapsto \alpha(u) & (f \Rightarrow^* \alpha)(a \mapsto u) &= f(a) \mapsto \alpha(u) \\
(f \Rightarrow \alpha)(a \mapsto u) &= f(a) \mapsto \alpha(u)
\end{aligned}$$

Generally, these actions can be derived mechanically and there is only one reasonable choice. The simplicity is deceptive, though, and it is easy to overlook how much one gets for free with a categorical approach. Perhaps the clearest example of this is $\otimes$. It is usually called the Kronecker product, defined as a complicated block matrix expression and relegated to advanced linear algebra courses. The functorial action, by contrast, could not be simpler.

The module constructions we have presented are related in various ways. More precisely, there are a number of *natural isomorphisms*. An isomorphism $\varphi : U \cong V$ is simply a linear map $\varphi : U \to V$ together with an inverse linear map $\varphi^{-1} : V \to U$. Naturality can be stated precisely using category theory, but for our purposes it is sufficient to think of 'natural' as 'polymorphic'.

When defining an isomorphism φ we write both directions simultaneously using the syntax $p \leftrightarrow q$ to mean $\varphi(p) = q$ and $p = \varphi^{-1}(q)$. A selection of isomorphisms can be seen in Table 1. We draw particular

attention to the relationship between free modules and tensor products given by the isomorphism:

$$\begin{array}{rcl} \mathbf{F}_K[A \times B] & \cong & \mathbf{F}_K[A] \otimes \mathbf{F}_K[B] \\ \langle (a,b) \rangle & \leftrightarrow & \langle a \rangle \otimes \langle b \rangle \end{array}$$

The tensor product of free modules can itself be written as a free module. However, the pattern matching notation belies the true cost of this conversion. The typical element of $\mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$ is a sum of terms like $(\sum_i r_i \langle a_i \rangle) \otimes (\sum_j s_j \langle b_j \rangle)$ which converts to $\sum_i \sum_j r_i s_j \langle (a_i, b_j) \rangle$, a quadratic increase in size. Converting back again yields $\sum_i \sum_j r_i s_j (\langle a_i \rangle \otimes \langle b_j \rangle)$. This is extensionally equal to the original term, but much larger. Hence, passing through the free module is not free! We will generally prefer to stay on the right side of this isomorphism, only converting when necessary. Fortunately, the isomorphisms shown thus far demonstrate that any polynomial type can be expressed as a module using $\mathbf{0}$, K, $\oplus$ and $\otimes$.

2.2.1 Free Modules Everywhere?

At this point we have seen that any module constructed using any combination of $\mathbf{0}$, K, $\oplus$, $\otimes$, $\mathbf{F}_K[\cdot]$, $\mathbf{F}_K^*[\cdot]$ and $\Rightarrow$ is isomorphic to some free module. Indeed, if K is a field—in which case modules are vector spaces—the classical mathematician would remark that by the Axiom of Choice every vector space is isomorphic to a free one. Why do we not simply consider only free modules then?

Firstly, these isomorphisms only concern the module structure. The algebra structure differs in many cases. For instance, we saw that $\mathbf{F}_K^*[A] \cong \mathbf{F}_K[A] \oplus K$ as modules, but certainly *not* as algebras (if the right-hand side even *has* an algebra structure, which is only the case when A is finite). Secondly, even if two modules are *extensionally* equal, i.e. isomorphic, they need not be *intensionally* equal. The clearest example of this is $\mathbf{F}_K[A \times B] \cong \mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$ where the right-hand side can express certain large terms much more compactly and converting to the left-hand side generally yields asymptotically larger terms.

3 Linear Algebra as a Query Language

Relational algebra serves as the traditional formalism underpinning query languages. We propose the theory of modules and the linear maps between them as an appealing generalised framework for expressing queries. So far we have seen that linear algebra can express a more general class of sets than the usual kinds of sets and multisets employed in query languages; in particular, it is possible to have sets with negative multiplicities, cofinite sets and more. We now present a 'Rosetta Stone' showing how the operations of relational algebra have corresponding linear maps on modules.

Selection In relational algebra *selection* restricts a set of tuples to the subset satisfying a given predicate. Suppose $P \subseteq A$. We define $\sigma_P : \mathbf{F}_K[A] \to \mathbf{F}_K[A]$ by

$$\sigma_P(\langle a \rangle) = \langle a \rangle \quad (\text{when } a \in P) \qquad\qquad \sigma_P(\langle a \rangle) = 0 \quad (\text{when } a \notin P)$$

The effect is that a generator $\langle a \rangle$ is preserved when $a \in P$ and eliminated otherwise.

Projection In relational algebra *projection* selects a subset of attributes, throwing away those not in the designated subset. Note that due to set semantics this may cause values to collapse, making the resulting

set smaller. For example projecting the A component of $\{(A:\texttt{foo},B:1),(A:\texttt{foo},B:2),(A:\texttt{bar},B:3)\}$ yields $\{(A:\texttt{foo}),(A:\texttt{bar})\}$. For a tensor product $\mathbf{F}_K[A]\otimes\mathbf{F}_K[B]$ we define the two projections as follows.

$$\pi_1 : \mathbf{F}_K[A]\otimes\mathbf{F}_K[B] \to \mathbf{F}_K[A] \qquad \pi_1(x\otimes y) = \#y\cdot x$$
$$\pi_2 : \mathbf{F}_K[A]\otimes\mathbf{F}_K[B] \to \mathbf{F}_K[B] \qquad \pi_2(x\otimes y) = \#x\cdot y$$

More complicated projections can be constructed using these two projections, the identity map, and the functorial action of $\otimes$. Note that, unlike relational algebra, all such projections preserve multiplicities.

Renaming. In relational algebra *renaming* changes the names of attributes. This makes sure everything is only done "up to choice of names", but it is also sometimes necessary in order to apply a natural join.

We do not use named attributes, but a similar concern arises with regards to the order and parenthesisation of attributes. To this end we have two natural isomorphisms, the *associator* and the *commutator*:

$$\alpha : U\otimes(V\otimes W)\cong(U\otimes V)\otimes W \qquad \beta : U\otimes V\cong V\otimes U$$

Combining these isomorphisms with the functorial action of $\otimes$, we can rearrange arbitrarily as needed.

Union and intersection. Union in relational algebra is just the usual union of sets. We define union as simply $+$. In general this form of union keeps track of multiplicities so for instance $(\langle a\rangle+\langle b\rangle)+(\langle b\rangle+\langle c\rangle)=\langle a\rangle+2\langle b\rangle+\langle c\rangle$.

In relational algebra *intersection* is typically not mentioned explicitly, but it arises as a special case of join when the two relations have the same set of attributes. In our approach intersection is the primitive upon which join is built. It is simply the product operation from the algebra structure. If $x,y:\mathbf{F}_K[A]$ then $xy:\mathbf{F}_K[A]$ is their intersection. Recall that the algebra product is a bilinear operator so multiplicities are multiplied. For instance $(2\langle a\rangle+3\langle b\rangle)(5\langle b\rangle+7\langle c\rangle)=(3\cdot5)\langle b\rangle=15\langle b\rangle$.

Cartesian product. Cartesian product in relational algebra is also the usual notion from set theory, and is traditionally viewed as an expensive operation that is best avoided if possible. Our version of the Cartesian product is the tensor product. We view it not as an operation, but as a symbolic term. In particular, we are perfectly comfortable writing down terms like

$$t = (\langle a_1\rangle+\cdots+\langle a_m\rangle)\otimes(\langle b_1\rangle+\cdots+\langle b_n\rangle)$$

without insisting that this be expanded to a canonical form as soon as possible. Such an expansion generally incurs a quadratic blow-up in expression size. Depending on what happens to t later we might never have to expand it. For instance, $\pi_2(t)$ can be computed as

$$\pi_2(t) = \#(\langle a_1\rangle+\cdots+\langle a_m\rangle)\cdot(\langle b_1\rangle+\cdots+\langle b_n\rangle) = m\cdot(\langle b_1\rangle+\cdots+\langle b_n\rangle)$$

The amount of work done was linear in the size of the symbolic term, and sublinear in the size of the hypothetically expanded form. Also note that even this result is not yet in canonical form—there is no need to distribute the scalar multiplication prematurely.

Static analysis will typically recognise opportunities like a projection immediately applied to a Cartesian product and do appropriate optimisation. By contrast, our approach does it at run time. It does not rely on a sufficiently clever analysis, it guarantees that products are not expanded until necessary and it even allows terms to be stored more efficiently in data structures between operations.

Natural join In relational algebra the *natural join* of two relations is constructed by taking their Cartesian product and keeping only the tuples where both sides agree about the values of shared attributes.

For example, consider the relations

$$\begin{aligned} x = \{&(A:a, B:1),\\ &(A:b, B:2),\\ &(A:c, B:3)\end{aligned} \qquad\qquad \begin{aligned} y = \{&(B:2, C:p)\\ &(B:3, C:q),\\ &(B:4, C:r)\}\end{aligned}$$

where the notation $(A:a, B:1)$ denotes a tuple with a value of a for attribute A and 1 for attribute B. Their join is

$$x \bowtie y = \{(A:b, B:2, C:p), (A:c, B:3, C:q)\}$$

In our system these two relations would be represented as the vectors x and y given by

$$\begin{aligned} x &: \mathbf{F}_K[\mathbf{Str}] \otimes \mathbf{F}_K[\mathbb{Z}] \qquad & x &= \langle a\rangle \otimes \langle 1\rangle + \langle b\rangle \otimes \langle 2\rangle + \langle c\rangle \otimes \langle 3\rangle \\ y &: \mathbf{F}_K[\mathbb{Z}] \otimes \mathbf{F}_K[\mathbf{Str}] & y &= \langle 2\rangle \otimes \langle p\rangle + \langle 3\rangle \otimes \langle q\rangle + \langle 4\rangle \otimes \langle r\rangle \quad . \end{aligned}$$

To compute their join we first have to inject them into a common module. This is done by going from $\mathbf{F}_K[\cdot]$ to $\mathbf{F}^*_K[\cdot]$ and adding 1's as necessary.

$$\begin{aligned} x' &: \mathbf{F}^*_K[\mathbf{Str}] \otimes \mathbf{F}^*_K[\mathbb{Z}] \otimes \mathbf{F}^*_K[\mathbf{Str}] \qquad & x' &= \langle a\rangle \otimes \langle 1\rangle \otimes 1 + \langle b\rangle \otimes \langle 2\rangle \otimes 1 + \langle c\rangle \otimes \langle 3\rangle \otimes 1 \\ y' &: \mathbf{F}^*_K[\mathbf{Str}] \otimes \mathbf{F}^*_K[\mathbb{Z}] \otimes \mathbf{F}^*_K[\mathbf{Str}] & y' &= 1 \otimes \langle 2\rangle \otimes \langle p\rangle + 1 \otimes \langle 3\rangle \otimes \langle q\rangle + 1 \otimes \langle 4\rangle \otimes \langle r\rangle \end{aligned}$$

The join of two elements of the same module is simply their intersection, which is given by multiplication. A naive approach to simplification is to apply distributivity bluntly and then simplify using identities for tensor products (see Appendix A.4 for details). In the end we get the simplified form:

$$\langle b\rangle \otimes \langle 2\rangle \otimes \langle p\rangle + \langle c\rangle \otimes \langle 3\rangle \otimes \langle q\rangle$$

This method of simplification is wasteful as we are expanding everything using distributivity only to have most components turn out to be 0. We shall later see that there is a much more efficient approach to simplifying expressions that in particular can solve basic joins like this one in linear time.

Outer join In relational algebra the *outer join* is similar to the natural join. The difference is that tuples from either input which would not occur anywhere in the output are included anyway. The missing attributes are populated with a **null** value.

Consider the previous example of a natural join. The result of the outer join would be as follows.

$$\{(A:a, B:1, C:\mathbf{null}), (A:b, B:2, C:p), (A:c, B:3, C:q), (A:\mathbf{null}, B:4, C:r)\}$$

There are also *left outer join* and *right outer join* operations, which only include extra tuples from the left and right input respectively.

In our system we use wildcards to achieve a similar effect. Note that though the wildcard bears superficial similarity to **null**, it is in fact the exact opposite: **null** is a special value that agrees with *no value* (including itself), the wildcard is a special value that agrees with *every value*. It could be argued that the main reason **null** is used to fill missing values in relational algebra is that it is the only somewhat applicable tool in the relational toolbox.

Recall that to compute the natural join of x and y we first embed them by adding wildcards to get x' and y'. The left outer join is then given by $x' \cdot (y'+1)$, the right outer join by $(x'+1) \cdot y'$, and the outer join by $(x'+1) \cdot (y'+1)$. By distributivity this is the same as $x' \cdot y' + x'$, $x' \cdot y' + y'$ and $x' \cdot y' + x' + y' + 1$ respectively. One way to think about an expression like $x' \cdot (y'+1)$ is that the added 1 ensures that every component of x' matches with *something*.

For the outer join one might want to use $(x'+1) \cdot (y'+1) - 1$ instead to avoid the last factor of 1. However, our proposed definition generalises more neatly to n-ary outer joins, which can be written simply as $(x_1+1) \cdots (x_n+1)$.

Aggregation In (extensions of) relational algebra *aggregation* computes values like sum or maximum over an attribute. For instance, given the relation $\{(A:p, B:2), (A:p, B:3), (A:q, B:4)\}$ aggregating B by sum would yield $\{(A:p, B:5), (A:q, B:4)\}$, while aggregating B by maximum would yield $\{(A:p, B:3), (A:q, B:4)\}$.

We take a different view of aggregation, as something that happens implicitly due to module semantics. This was evident when discussing projection, where the discarded attributes are automatically counted with multiplicities. The specifics of aggregation depends on the nature of the ring K.

Take the example above and represent it as follows: $2\langle p\rangle + 3\langle p\rangle + 4\langle q\rangle = (2+3)\langle p\rangle + 4\langle q\rangle$. If K is $\mathbb{Z}$ with ordinary arithmetic this reduces to $5\langle p\rangle + 4\langle q\rangle$. Note how a tuple such as $(A:p, B:2)$ was represented as $2\langle p\rangle$ and not as $\langle p\rangle \otimes \langle 2\rangle$. We *moved* the attribute into the ring.

Not all aggregations can be expressed directly as a ring structure, though. For instance, integers extended with infinity and equipped with minimum and maximum operations only form a *semiring* since negation is impossible. Our approach can easily work with semirings as well, but as we shall see shortly negation plays an important role and should not be given up so easily.

Instead, we express the aggregation using nested free modules to group data. Each grouping is an element of $\mathbf{F}_{\mathbb{F}_2}[A]$, i.e. an ordinary finite set. The example above would be represented as follows: $\langle p\rangle \otimes \langle\langle 2\rangle + \langle 3\rangle\rangle + \langle q\rangle \otimes \langle\langle 4\rangle\rangle$. Non-linear aggregations like minimum are modelled as functions (*not* linear maps) $\min : \mathbf{F}_{\mathbb{F}_2}[\mathbb{Z}] \to \mathbb{Z}_\infty$ with $\min(\langle a_1\rangle + \cdots + \langle a_n\rangle) = \min\{a_1, \ldots, a_n\}$. Here $\mathbb{Z}_\infty$ is $\mathbb{Z}$ with ∞ adjoined so that $\min\{\}$ is well-defined. Aggregating by minimum on the second attribute we get:

$$(\mathrm{id}_{\mathbf{F}_K[\mathbf{Str}]} \otimes \mathbf{F}_K[\min])(\langle p\rangle \otimes \langle\langle 2\rangle + \langle 3\rangle\rangle + \langle q\rangle \otimes \langle\langle 4\rangle\rangle) = \langle p\rangle \otimes \langle 2\rangle + \langle q\rangle \otimes \langle 4\rangle$$

In general, nested free modules allow any non-linear function to be included in an otherwise linear query. Besides aggregations, this includes operations like turning negative multiplicities into zeroes.

These considerations demonstrate that linear maps are exactly the maps that can be easily adapted to a distributed setting. The non-linear maps, on the other hand, depend on having the entire dataset at hand and subjected to at least some degree of simplification, which implies that synchronisation is necessary.

Domain computations. Relational algebra proper does not provide a way to transform data, but in practical realisations this is usually possible. For instance, given the relation $\{(A \mapsto \text{foo}), (A \mapsto \text{bar})\}$ we might transform it using a function **upper** that maps strings to uppercase, getting the result $\{(A \mapsto \text{FOO}), (A \mapsto \text{BAR})\}$. In our system this is achieved by the functorial action of $\mathbf{F}_K[\cdot]$. In particular given $\mathbf{upper} : \mathbf{Str} \to \mathbf{Str}$ we have $\mathbf{F}_K[\mathbf{upper}] : \mathbf{F}_K[\mathbf{Str}] \to \mathbf{F}_K[\mathbf{Str}]$. We can therefore apply it to an element as follows: $\mathbf{F}_K[\mathbf{upper}](\langle \text{foo}\rangle + \langle \text{bar}\rangle) = \langle \mathbf{upper}(\text{foo})\rangle + \langle \mathbf{upper}(\text{bar})\rangle = \langle \text{FOO}\rangle + \langle \text{BAR}\rangle$.

Insert and delete. For persistent relations there is the question of how to do updates. Relational algebra is based on set theory, so updates can be done using set union and difference.

In our system all updates are done by addition. Deleting an element amounts to adding its negative. For example, say the database contains $\langle a\rangle + \langle b\rangle$ and we want to add c and delete b. This update is computed as $(\langle a\rangle + \langle b\rangle) + (\langle c\rangle - \langle b\rangle) = \langle a\rangle + \langle c\rangle$.

Note that there is no conceptual distinction between *databases* and *database updates*, since deletions are simply represented as negative elements. Furthermore, since $+$ is commutative the order of updates does not matter. For instance, suppose the database only contains $\langle a\rangle$ and we run the update above: $\langle a\rangle + (\langle c\rangle - \langle b\rangle) = \langle a\rangle - \langle b\rangle + \langle c\rangle$. This leaves a database with a negative occurrence of b. If we then insert b afterwards we end up with $\langle a\rangle + \langle c\rangle$ again. In a sense this is the easiest possible conflict resolution strategy: just accept the data from every update and add it all together! Of course, there are times when we might want to ensure that the database does not contain negative multiplicities by, say, setting them all to 0 using an explicit operation. In that case—and that case only—we separate updates into before and after that given point.

4 Algebraic Evaluation

We now turn to the question of evaluation. More precisely, the question of *simplifying* module terms. What kind of simplified form do we have in mind; is simplification even necessary? That depends on the context of use, but generally a simplified form involves some of the following:

- No zeroes in additions or multiplications, and no multiplications except $r \cdot \langle a\rangle$ where $r : K$.
- No repeated generators, e.g. $\langle a\rangle + \langle b\rangle + \langle a\rangle$ should be $2\langle a\rangle + \langle b\rangle$.
- No sums of biproducts, e.g. $(u_1, v_1) + (u_2, v_2)$ should be $(u_1 + u_2, v_1 + v_2)$.
- No tensor products where either factor is a sum.

The kind of observation we wish to make dictates which requirements are necessary. For example, the most elementary observation we can make is to ask if a term is equal to 0. In query terms this corresponds to the question of satisfiability. We certainly need to avoid repeated generators as otherwise they might happen to cancel out, e.g. $\langle a\rangle - 2\langle a\rangle + \langle a\rangle$ is a non-trivial representation of 0. On the other hand, if the term is a tensor product we do not need to expand it, since $u \otimes v$ is 0 precisely when either factor is 0.

In discussing term simplification it is important to distinguish between *intensional* and *extensional* equality. Two terms are intensionally equal when they are written the same way (modulo renaming). They are extensionally equal when they can be shown to be equal using the presented algebraic identities. For example, $0+0+0+0$ and 0 are extensionally equal which can be argued using the identity $x+0=x$ repeatedly. However, they are not intensionally equal and indeed we would consider the latter a more compact version of the former.

4.1 Finite maps

Before dealing with simplification of arbitrary terms we consider finite maps in particular. They have a great deal of structure to exploit and also play a vital rôle in making simplification efficient.

Suppose we have some term $x : A \Rightarrow U$ and *assume that we already have a method for simplifying terms from* U. To simplify x we proceed depending on the nature of A. Recall that we have the following isomorphisms at our disposal:

$$\mathbf{cp}_0 : 0 \Rightarrow U \cong \mathbf{0} \qquad \mathbf{cp}_+ : (A + B) \Rightarrow U \cong (A \Rightarrow U) \oplus (B \Rightarrow U)$$
$$\mathbf{cp}_1 : 1 \Rightarrow U \cong U \qquad \mathbf{cp}_\times : (A \times B) \Rightarrow U \cong A \Rightarrow B \Rightarrow U$$

The idea is that we wrap the term in the appropriate isomorphism. To start, suppose $A = 1$ so $x : 1 \Rightarrow U$. We can then write x as $\mathbf{cp}_1^{-1}(\mathbf{cp}_1(x))$. The term $\mathbf{cp}_1(x)$ is then simplified to get some term $y : U$ and the simplified form of x becomes $\mathbf{cp}_1^{-1}(y)$.

Now suppose $A = A_1 + A_2$ so $x : (A_1 + A_2) \Rightarrow U$. We write x as $\mathbf{cp}_+^{-1}(\mathbf{cp}_+(x))$. The term $\mathbf{cp}_+(x)$ reduces to some term $(y_1, y_2) : (A_1 \Rightarrow U) \oplus (A_2 \Rightarrow U)$ (a biproduct can always be reduced to a pair by simple component-wise addition). At this point y_1 and y_2 are finite maps with index sets A_1 and A_2 respectively, so we apply the procedure recursively to get simplified forms z_1 and z_2. The simplified form of x then becomes $\mathbf{cp}_+^{-1}(z_1, z_2)$.

Finally, suppose $A = A_1 \times A_2$ so $x : (A_1 \times A_2) \Rightarrow U$. In this case $\mathbf{cp}_\times(x)$ has type $A_1 \Rightarrow A_2 \Rightarrow U$. Recursively we have a procedure to simplify terms from $A_2 \Rightarrow U$; and given this we also get a procedure to simplify $\mathbf{cp}_\times(x)$ to get some term y. The simplified form of x becomes $\mathbf{cp}_\times^{-1}(y)$.

What about sets that are not sums or products? Recursively defined sets do not pose any extra challenge; we omit the details. Function spaces, i.e. $A = A_1 \to A_2$, are not amenable to simplification in general and we exclude them from consideration. Finally, there are primitives like n-bit integers. These can be handled using specific methods. In the case of finite precision integers an efficient solution is to represent finite maps as Patricia tries. For strings a good choice would be a radix trie.

An example of this simplication procedure in action can be found in Appendix B.

4.2 Simplification

We now sketch the general approach to simplification of terms. Suppose we have terms $u_1, \ldots, u_n : \mathbf{F}_K^*[A_1] \otimes \cdots \otimes \mathbf{F}_K^*[A_m]$ and we want to simplify the product $u_1 \cdots u_n$. The first step is to exploit the isomorphism $A \Rightarrow^* U \cong \mathbf{F}_K^*[A] \otimes U$ for each A_i to get corresponding simplified terms $v_1, \ldots, v_n : A_1 \Rightarrow^* \cdots \Rightarrow^* A_m \Rightarrow^* K$. Simplification then proceeds by dealing with one level of this nested map structure at a time. One of the terms will act as enumerator and the others will act as filters. Find the v_i whose expression as a sum contains the fewest components. Without loss of generality assume that this is v_1. For ease of presentation assume also that A_1 is a primitive set so we can write the simplified form of v_1 as $(\sum_{1 \le i \le k} a_i \mapsto v_{1,i}) + (* \mapsto v_{1,*})$. We apply distributivity to move everything under this summation, and we turn the intersections with $v_2 \cdots v_n$ into lookups.

$$\begin{aligned} v_1 v_2 \cdots v_n &= ((\textstyle\sum_{1 \le i \le k} a_i \mapsto v_{1,i}) + (* \mapsto v_{1,*}))v_2 \cdots v_n \\ &= (\textstyle\sum_{1 \le i \le k} (a_i \mapsto v_{1,i})v_2 \cdots v_n) + (* \mapsto v_{1,*})v_2 \cdots v_n \\ &= (\textstyle\sum_{1 \le i \le k} a_i \mapsto v_{1,i}v_2(a_i) \cdots v_n(a_i)) + (* \mapsto v_{1,*}v_2(*) \cdots v_n(*)) \end{aligned}$$

At this point there are intersection subterms of type $A_2 \Rightarrow^* \cdots \Rightarrow^* A_n \Rightarrow^* K$, so we can apply the procedure recursively. In the base case when $m = 0$ the problem is reduced to simple multiplication in K. Finally, the resulting iterated compact map is turned back into a tensor product of compact free modules.

The essence of this approach is that we deal with one attribute at a time and simplify with regards to that attribute across the entire term. Distributivity is only applied as necessary to proceed. This general strategy works for any term, not just multiplications.

4.3 Worst-case optimality

The particular case of a multiplication of sums correspond to *conjunctive queries* in relational algebra. Conjunctive queries have a simple structure, but are difficult to evaluate efficiently. The main obstacle to efficiency is what is known as cyclic queries. The simplest one is the triangle query, which given

attributes A, B and C asks to compute the join of $x : \mathbf{F}_K[A] \otimes \mathbf{F}_K[B]$, $y : \mathbf{F}_K[A] \otimes \mathbf{F}_K[C]$ and $z : \mathbf{F}_K[B] \otimes \mathbf{F}_K[C]$. If all three inputs have size n it can be shown (e.g. using fractional edge covers [9]) that the output has size $O(n^{\frac{3}{2}})$ and indeed our simplification algorithm, sketched above, produces the output in $O(n^{\frac{3}{2}})$ worst-case time. Traditional approaches using query planning cannot get below $O(n^2)$ worst-case time.

More generally, for any given query and given sizes of input data we can consider the worst-case output size. Roughly speaking, a *worst-case optimal* algorithm solves the join problem in time proportional to the largest possible output size for an input of a given size. This is realistically the best we can hope for since conjunctive queries are powerful enough to encode SAT-like problems.

Our simplification procedure, when applied to the special case of conjunctive queries, is indeed worst-case optimal. This is tricky to show since computation steps that compute a multiplicative subterm vv' that *eventually* yields 0 are wasted: they contribute nothing to the output! This is why it is important to choose the v_i whose sum contains the fewest components above.

5 Discussion

We have developed an algebraic theory based on mathematical modules and a number of associated categorical constructions as a simultaneously expressive, reasonable *and* efficient purely functional programming framework for generalised relational query languages. Its efficiency is based on employing symbolic operators to retain data in *un*normalised form at run time, exploiting their universal properties to perform optimising algebraic rewriting. Informally, the functions operate on quotient types modulo algebraic equivalences *behaviourally*, but their efficient *execution* crucially depends on the specifics of these representations.

We have shown how controlled algebraic rewriting combined with efficient implementation of linear isomorphisms can be used to implement generalised database joins that perform asymptotically as well as can be hoped for. This hinges crucially on an extra element adjoined to free modules and finite maps, which has an intuitive interpretation both algebraically (as multiplicative unit for intersection) and as a wildcard ("compatible with anything") and thus facilitates compactly representing infinite relations in combination with subtraction and tensor product.

5.1 Implementation

The concepts we have presented are implemented as a Haskell library. Modules are represented by a data type `Space`. Using the Data Kinds extension this will serve as a parameter for the type of elements:

```
data Vec r v = Zero | Add (Vec r v) (Vec r v)
             | Mul r (Vec r v) | Gen (Gen r v)
data family Gen r (v :: Space) :: Type
```

An element of `Vec r v` represents a term of the `r`-module `v`. Such an element is built from linear combinations of generators, represented by the type family `Gen r v`. Most instances are simple, like tensor products:

```
data instance Gen r (u :*: v) = Tensor (Vec r u) (Vec r v)
```

Finite maps, however, do not admit a both generic and efficient definition. Instead, they are defined for each index type based on natural isomorphisms. For example, the isomorphism $\mathbf{cp}_\times : (A \times B) \Rightarrow U \cong A \Rightarrow B \Rightarrow U$ motivates the definition:

```
newtype instance Gen r ((a, b) :=> v) = CopowProd (Vec r (a :=> b :=> v))
```

Thus, `CopowProd` directly represents $\mathbf{cp}_\times^{-1}$, but since we are using `newtype` it has no computational cost.

Related work Our modules and associative algebras provide a general, established and mathematically deep and well-studied reference framework for structures proposed in the literature, such as provenance semirings [8] and semiring dictionaries [25]. For example, in provenance semirings a K-relation is an element of $\mathbf{F}_K[T_{a_1} \times \ldots \times T_{a_n}]$. Replacing K by $R = \mathbb{N}[X]$, the free commutative semiring generated by X yields queries evaluated over R instead of K. This incorporates provenance for aggregations [1] since $\mathbf{F}_K[T]$ provides folding over arbitrary modules. Our extension with wildcards extends provenance from finite K-relations to infinite K-relations. Further, our approach accounts for probabilistic databases [4] as algebras $\mathbf{F}_{\mathbb{R}}[T]$, with real numbers serving as quasi-probabilities, as well as sets with negative multiplicity (see, e.g., [14, 3]).

Linear algebra has previously been proposed as a framework for interpreting and implementing querying on databases. Notably, typed matrix algebra [21, 20, 19, 15] has been used to provide operations on matrix-shaped data (see also array programming languages such as Futhark [11] and Single Assignment C [24]). Note that our approach works on infinite-dimensional spaces, not only finite-dimensional ones, and it is essential to represent linear maps as functions rather than matrices. More closely related to our work, there are expressive linear algebra-based domain-specific languages/frameworks [13, 26] that provide expressive database and data analytics operations, with evaluation to standard normalised data structures using Kiselyov's tagless final approach. These do not support efficient data structures and execution of *bi*linear maps, however, where we employ symbolic tensor products and efficient intersection, aided by compact maps for both expressiveness and efficiency.

The problem of efficient execution of relational database queries has been studied extensively, with joins in particular posing a challenge. Traditional methods rely on decomposing k-ary joins into binary joins, carefully creating a good *query plan* in order to minimise the sizes of intermediate results and outputs [7], while our approach instead hinges on sensible and efficient *expression simplification* (see [2] for a formal treatment of this problem). While it is possible to evaluate *acyclic* join queries in time linear in the size of the query, the input, and the output [28, 22] and there are methods that deal with "almost" acyclic join queries [5], this is infeasible for cyclic queries such as the triangle query, as deciding whether the output is empty is NP-hard [16]. To quantify how good a query algorithm is, the notion of *worst-case optimal complexity* was developed [17, 18], focussing on the *data complexity* of the problem, which ignores the size of the query itself [27]. A join algorithm is *worst-case optimal* if it executes in time linear in N and O, where O is the maximal size of an output of the join query applied to input relations whose sizes sum to N.

We have drawn on a great number of categorical concepts, not only to drive generalisation and identify suitable compositional constructs that make for a pleasant framework with strong and useful properties, but also to exploit term constructors as symbolic data structures and exception-less algebraic equalities for their efficient execution. It is difficult to do justice to the many works in category theory, abstract algebra and categorical functional programming we have built on. Most closely related is [6], which presents a framework for relational algebra based on the adjunctions that generate it.

Future work While logic programming does well in handling positive queries, negation is typically handled in an ad-hoc and unsatisfactory way, for example through *negation as failure*. In our setting, data with *negative* multiplicity is handled no different from data in *positive* quantity: can this give a more satisfactory treatment of negation in logic programming? Further, in the current form, our setting is quite conservative in that it is not capable of describing recursive queries, even if they are ultimately well-behaved. A possible solution to this shortcoming would be to study modules which are somehow topological (e.g., appropriately ordered, or equipped with a norm or inner product), as this could give us

access to fixed point theorems directly associated with the semantics of recursion.

Acknowledgements. This work was made possible by Independent Research Fund Denmark grant *FUTHARK: Functional Technology for High-performance Architectures* and DFF–International Postdoc 0131-00025B. We greatly appreciate and thank the three anonymous referees for their comments and recommendations.

References

[1] Yael Amsterdamer, Daniel Deutch & Val Tannen (2011): *Provenance for aggregate queries*. In: *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 153–164, doi:10.1145/1989284.1989302.

[2] Jacques Carette (2004): *Understanding Expression Simplification*. In: *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, ACM, p. 72–79, doi:10.1145/1005285.1005298.

[3] Jacques Carette, Alan P. Sexton, Volker Sorge & Stephen M. Watt (2010): *Symbolic Domain Decomposition*. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo & Alan P. Sexton, editors: *Intelligent Computer Mathematics*, Springer, pp. 172–188, doi:10.2307/1968168.

[4] Nilesh Dalvi & Dan Suciu (2007): *Efficient Query Evaluation on Probabilistic Databases*. *The VLDB Journal* 16(4), pp. 523–544, doi:10.1007/s00778-006-0004-3.

[5] Jörg Flum, Markus Frick & Martin Grohe (2002): *Query Evaluation via Tree-Decompositions*. *J. ACM* 49(6), pp. 716–752, doi:10.1145/602220.602222.

[6] Jeremy Gibbons, Fritz Henglein, Ralf Hinze & Nicolas Wu (2018): *Relational algebra by way of adjunctions*. *Proceedings of the ACM on Programming Languages: International Conference onf Functional Programming (ICFP)* 2(ICFP), p. 86, doi:10.1145/3235035.

[7] Goetz Graefe (1993): *Query Evaluation Techniques for Large Databases*. *ACM Comput. Surv.* 25(2), pp. 73–169, doi:10.1145/152610.152611.

[8] Todd J. Green, Grigoris Karvounarakis & Val Tannen (2007): *Provenance Semirings*. In: *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, Association for Computing Machinery, New York, NY, USA, pp. 31–40, doi:10.1145/1265530.1265535.

[9] Martin Grohe & Dániel Marx (2014): *Constraint Solving via Fractional Edge Covers*. *ACM Trans. Algorithms* 11(1), doi:10.1145/2636918.

[10] Fritz Henglein & Ralf Hinze (2013): *Distributive Sorting and Searching: From Generic Discrimination to Generic Tries*. In Chung-chieh Shan, editor: *Proc. 11th Asian Symposium on Programming Languages and Systems (APLAS)*, *Lecture Notes in Computer Science (LNCS)* 8301, Springer, pp. 315–332, doi:10.1007/978-3-319-03542-0_23.

[11] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein & Cosmin E. Oancea (2017): *Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, ACM, pp. 556–571, doi:10.1145/1806596.1806606.

[12] Ralf Hinze (2000): *Generalizing generalized tries*. *Journal of Functional Programming* 10(4), pp. 327–351, doi:10.1017/S0956796800003713.

[13] Oleg Kiselyov (2018): *Reconciling abstraction with high performance: A metaocaml approach*. *Foundations and Trends in Programming Languages* 5(1), pp. 1–101, doi:10.1561/2500000038_supp.

[14] Daniel Loeb (1992): *Sets with a negative number of elements*. *Advances in Mathematics* 91(1), pp. 64–74, doi:10.1016/0001-8708(92)90011-9.

[15] H.D. Macedo & J.N. Oliveira (2012): *Typing linear algebra: A biproduct-oriented approach*. Science of Computer Programming.

[16] David Maier, Yehoshua Sagiv & Mihalis Yannakakis (1981): *On the complexity of testing implications of functional and join dependencies*. Journal of the ACM (JACM) 28(4), pp. 680–695, doi:10.1145/322261.322263.

[17] H.Q. Ngo, E. Porat, C. Ré & A. Rudra (2012): *Worst-case optimal join algorithms*. In: Proceedings of the 31st symposium on Principles of Database Systems, ACM, pp. 37–48.

[18] Hung Q Ngo, Ely Porat, Christopher Ré & Atri Rudra (2018): *Worst-case optimal join algorithms*. Journal of the ACM (JACM) 65(3), pp. 1–40, doi:10.5555/795662.796318.

[19] J. Oliveira (2012): *Typed Linear Algebra for Weighted (Probabilistic) Automata*. Implementation and Application of Automata, pp. 52–65, doi:10.1007/978-3-642-13321-3_16.

[20] J.N. Oliveira (2012): *Towards a linear algebra of programming*. Formal Aspects of Computing 24(4), pp. 433–458, doi:10.1016/j.fss.2008.12.008.

[21] José Nuno Oliveira & Hugo Daniel Macedo (2017): *The data cube as a typed linear algebra operator*. In: Proceedings of The 16th International Symposium on Database Programming Languages, pp. 1–11, doi:10.1145/99370.99404.

[22] Anna Pagh & Rasmus Pagh (2006): *Scalable computation of acyclic joins*. Journal of the ACM, doi:10.1145/1142351.1142384.

[23] Matija Pretnar (2015): *An Introduction to Algebraic Effects and Handlers. Invited tutorial paper*. In: The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI), Electronic Notes in Theoretical Computer Science 319, pp. 19–35.

[24] Sven-Bodo Scholz (1994): *Single-assignment C — Functional Programming Using Imperative Style*. In: 6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England, UK, University of East Anglia, Norwich, England, UK, pp. 211–2113.

[25] Amir Shaikhha, Mathieu Huot, Jaclyn Smith & Dan Olteanu (2021): *Functional Collection Programming with Semi-Ring Dictionaries*. arXiv preprint arXiv:2103.06376.

[26] Amir Shaikhha & Lionel Parreaux (2019): *Finally, a Polymorphic Linear Algebra Language (Pearl)*. In: 33rd European Conference on Object-Oriented Programming (ECOOP 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[27] Moshe Vardi (1982): *The Complexity of Relational Query Languages (Extended Abstract)*. pp. 137–146, doi:10.1145/800070.802186.

[28] Dan E. Willard (2002): *An Algorithm for Handling Many Relational Calculus Queries Efficiently*. Journal of Computer and System Sciences 65(2), pp. 295 – 331, doi:10.1006/jcss.2002.1848.

[29] Robert Wisbauer (1991): *Foundations of module and ring theory*. 3, CRC Press.

A Appendix

We provide definitions of rings and modules. See the literature (for example [29]) for more information.

A.1 Rings

Definition 2 (Ring)**.** *A* commutative ring *K is an algebraic structure* $K = (|K|, +, -, 0, \cdot, 1)$ *such that for all* $x, y, z \in |K|$*:*

$$\begin{aligned} x+(y+z) &= (x+y)+z && (1)\\ x+y &= y+x && (2)\\ x+0 &= x && (3)\\ x+(-x) &= 0 && (4)\\ 0\cdot x &= 0 && (5)\\ 1\cdot x &= x && (6)\\ x\cdot(y\cdot z) &= (x\cdot y)\cdot z && (7)\\ x\cdot(y+z) &= (x\cdot y)+(x\cdot z) && (8)\\ x\cdot y &= y\cdot x && (9) \end{aligned}$$

Typical examples of rings are $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{C}$ with the usual addition and multiplication operations.

A.2 Modules

Definition 3 (Module)**.** *A* module *V over commutative ring K is an algebraic structure* $V = (|V|, +, 0, \cdot)$ *such that for all* $r, s \in |K|$*,* $u, v, w \in |V|$*:*

$$\begin{aligned} u+(v+w) &= (u+v)+w && (10)\\ u+v &= v+u && (11)\\ u+0 &= u && (12)\\ 0\cdot u &= 0 && (13)\\ 1\cdot u &= u && (14)\\ r\cdot(s\cdot u) &= (r\cdot s)\cdot u && (15)\\ r\cdot(u+v) &= (r\cdot u)+(r\cdot v) && (16)\\ (r+s)\cdot u &= (r\cdot u)+(s\cdot u) && (17) \end{aligned}$$

All modules support negation as well, defined by $-u = (-1)\cdot u$. If K is a field then V is a vector space in the usual sense.

Given two modules U and V, a *linear map* between them is a function $f : U \to V$ of the underlying sets such that

$$\begin{aligned} f(u+v) &= f(u)+f(v)\\ f(r\cdot u) &= r\cdot f(u) \end{aligned}$$

Given three modules U, V and W, a *bilinear map* between them is a function $f : U \times V \to W$ of the underlying sets such that

$$\begin{aligned} f(u_1+u_2, v) &= f(u_1, v)+f(u_2, v)\\ f(u, v_1+v_2) &= f(u, v_1)+f(u, v_2)\\ f(r\cdot u, v) &= r\cdot f(u, v)\\ f(u, r\cdot v) &= r\cdot f(u, v) \end{aligned}$$

In other words a bilinear map is linear in each of its arguments.

A.3 Biproducts

The biproduct satisfies the universal property of binary products: there are linear maps $p_1 : U \oplus V \to U$ and $p_2 : U \oplus V \to V$ such that for any module W with linear maps $p'_1 : W \to U$ and $p'_2 : W \to V$ there is a unique linear map $(p'_1, p'_2) : W \to U \oplus V$ such that $p_1 \circ (p'_1, p'_2) = p'_1$ and $p_2 \circ (p'_1, p'_2) = p'_2$. The p's are projections and (p'_1, p'_2) is the pair constructor. Operationally:

$$\begin{aligned} p_1(u,v) &= u \\ p_2(u,v) &= v \\ (p'_1, p'_2)(w) &= (p'_1(w), p'_2(w)) \end{aligned}$$

This universal property gives a way to define linear maps *into* the biproduct by giving the result of projecting each component. In other words, definition by copattern matching. For instance

$$\begin{aligned} f &: V \to V \oplus V \\ p_1(f(v)) &= v \\ p_2(f(v)) &= 3v \end{aligned}$$

The biproduct also satisfies the universal property of binary coproducts, obtained by simply reversing the direction of all arrows: there are linear maps $i_1 : U \to U \oplus V$ and $i_2 : V \to U \oplus V$ such that for any module W with linear maps $i'_1 : U \to W$ and $i'_2 : V \to W$ there is a unique linear map $[i'_1, i'_2] : U \oplus V \to W$ such that $[i'_1, i'_2] \circ i_1 = i'_1$ and $[i'_1, i'_2] \circ i_2 = i'_2$.

The i's are injections and $[i'_1, i'_2]$ is case analysis. Operationally:

$$\begin{aligned} i_1(u) &= (u, 0) \\ i_2(v) &= (0, v) \\ [i'_1, i'_2](u,v) &= i'_1(u) + i'_2(v) \end{aligned}$$

This universal property gives a way to define linear maps *out of* the biproduct by explaining what happens to either injection. In other words, definition by pattern matching. For instance

$$\begin{aligned} f &: V \oplus V \to V \\ f(i_1(v)) &= 2v \\ f(i_2(v)) &= v \end{aligned}$$

From a programming point of view biproducts are an unfamiliar construction: a data type that supports both pattern matching and copattern matching. They can be treated as either a sum or a product type depending on which is most convenient.

Finally, we describe the canonical algebra structure on the biproduct. Just like for free modules, the motivation for the definition is intersection of multisets. Suppose U and V are algebras. Then so is

$U \oplus V$. The definition of $1_{U\oplus V}$ uses copattern matching while $\cdot$ and # use pattern matching.

$$\begin{aligned} p_1(1_{U\oplus V}) &= 1_U \\ p_2(1_{U\oplus V}) &= 1_V \\ i_1(u)\cdot i_1(u') &= i_1(u\cdot u') \\ i_1(u)\cdot i_2(v') &= 0 \\ i_2(v)\cdot i_1(u') &= 0 \\ i_2(v)\cdot i_2(v') &= i_2(v\cdot v') \\ \#i_1(u) &= \#u \\ \#i_2(v) &= \#v \end{aligned}$$

An element $(u,v) \in U \oplus V$ can be thought of as a record value $\{1 \mapsto u, 2 \mapsto v\}$. Biproducts generalise straightforwardly from tuples to *records* via $\oplus_{i\in I} U_i$, where I is an index set I of field names i, whose associated values come from a module U_i that depends on i.

A.4 Naive intersection

The join of two elements of the same module is their intersection, which is given by multiplication.

$$x' \cdot y' = (\langle a\rangle \otimes \langle 1\rangle \otimes 1 + \langle b\rangle \otimes \langle 2\rangle \otimes 1 + \langle c\rangle \otimes \langle 3\rangle \otimes 1) \cdot (1 \otimes \langle 2\rangle \otimes \langle p\rangle + 1 \otimes \langle 3\rangle \otimes \langle q\rangle + 1 \otimes \langle 4\rangle \otimes \langle r\rangle)$$

Using distributivity we expand the multiplication.

$$\begin{aligned} & (\langle a\rangle \otimes \langle 1\rangle \otimes 1)\cdot(1 \otimes \langle 2\rangle \otimes \langle p\rangle) + (\langle a\rangle \otimes \langle 1\rangle \otimes 1)\cdot(1 \otimes \langle 3\rangle \otimes \langle q\rangle) + (\langle a\rangle \otimes \langle 1\rangle \otimes 1)\cdot(1 \otimes \langle 4\rangle \otimes \langle r\rangle) \\ + \quad & (\langle b\rangle \otimes \langle 2\rangle \otimes 1)\cdot(1 \otimes \langle 2\rangle \otimes \langle p\rangle) + (\langle b\rangle \otimes \langle 2\rangle \otimes 1)\cdot(1 \otimes \langle 3\rangle \otimes \langle q\rangle) + (\langle b\rangle \otimes \langle 2\rangle \otimes 1)\cdot(1 \otimes \langle 4\rangle \otimes \langle r\rangle) \\ + \quad & (\langle c\rangle \otimes \langle 3\rangle \otimes 1)\cdot(1 \otimes \langle 2\rangle \otimes \langle p\rangle) + (\langle c\rangle \otimes \langle 3\rangle \otimes 1)\cdot(1 \otimes \langle 3\rangle \otimes \langle q\rangle) + (\langle c\rangle \otimes \langle 3\rangle \otimes 1)\cdot(1 \otimes \langle 4\rangle \otimes \langle r\rangle) \end{aligned}$$

Next we exploit that $\cdot$ works component-wise on tensor products.

$$\begin{aligned} & (\langle a\rangle\cdot 1) \otimes (\langle 1\rangle\cdot\langle 2\rangle) \otimes (1\cdot\langle p\rangle) + (\langle a\rangle\cdot 1) \otimes (\langle 1\rangle\cdot\langle 3\rangle) \otimes (1\cdot\langle q\rangle) + (\langle a\rangle\cdot 1) \otimes (\langle 1\rangle\cdot\langle 4\rangle) \otimes (1\cdot\langle r\rangle) \\ + \quad & (\langle b\rangle\cdot 1) \otimes (\langle 2\rangle\cdot\langle 2\rangle) \otimes (1\cdot\langle p\rangle) + (\langle b\rangle\cdot 1) \otimes (\langle 2\rangle\cdot\langle 3\rangle) \otimes (1\cdot\langle q\rangle) + (\langle b\rangle\cdot 1) \otimes (\langle 2\rangle\cdot\langle 4\rangle) \otimes (1\cdot\langle r\rangle) \\ + \quad & (\langle c\rangle\cdot 1) \otimes (\langle 3\rangle\cdot\langle 2\rangle) \otimes (1\cdot\langle p\rangle) + (\langle c\rangle\cdot 1) \otimes (\langle 3\rangle\cdot\langle 3\rangle) \otimes (1\cdot\langle q\rangle) + (\langle c\rangle\cdot 1) \otimes (\langle 3\rangle\cdot\langle 4\rangle) \otimes (1\cdot\langle r\rangle) \end{aligned}$$

The multiplications now all involve only generators and 1, so they can all be simplified.

$$\begin{aligned} & \langle a\rangle \otimes 0 \otimes \langle p\rangle + \langle a\rangle \otimes 0 \otimes \langle q\rangle + \langle a\rangle \otimes 0 \otimes \langle r\rangle \\ + \quad & \langle b\rangle \otimes \langle 2\rangle \otimes \langle p\rangle + \langle b\rangle \otimes 0 \otimes \langle q\rangle + \langle b\rangle \otimes 0 \otimes \langle r\rangle \\ + \quad & \langle c\rangle \otimes 0 \otimes \langle p\rangle + \langle c\rangle \otimes \langle 3\rangle \otimes \langle q\rangle + \langle c\rangle \otimes 0 \otimes \langle r\rangle \end{aligned}$$

All tensor products with a 0 component can be eliminated due to linearity. This leaves:

$$\langle b\rangle \otimes \langle 2\rangle \otimes \langle p\rangle + \langle c\rangle \otimes \langle 3\rangle \otimes \langle q\rangle$$

B Simplification example

For primitive types we will simply write the simplified form as $y = \sum_i (a_i \mapsto u_i)$ with the understanding that each $a : A$ occurs at most once, and computing $y(a)$ can be done using work linear in the size of a (which for finite precision integers would be constant). To see all this in action consider the term

$$\begin{aligned} x &: (\mathbf{Str} \times (\mathbf{Str} + \mathbb{N})) \Rightarrow K \\ x &= ((a, \mathbf{left}(p)) \mapsto 1) + ((b, \mathbf{right}(4)) \mapsto 1) + ((a, \mathbf{right}(3)) \mapsto 1) + ((a, \mathbf{left}(p)) \mapsto 1) \end{aligned}$$

The simplification proceeds as follows:

$$
\begin{aligned}
x &= ((a,\mathbf{left}(p)) \mapsto 1) + ((b,\mathbf{right}(4)) \mapsto 1) + ((a,\mathbf{right}(3)) \mapsto 1) + ((a,\mathbf{left}(p)) \mapsto 1) \\
&\qquad \text{apply } \mathbf{cp}_{\times} \\
&= \mathbf{cp}_{\times}^{-1}((a \mapsto \mathbf{left}(p) \mapsto 1) + (b \mapsto \mathbf{right}(4) \mapsto 1) + (a \mapsto \mathbf{right}(3) \mapsto 1) + (a \mapsto \mathbf{left}(p) \mapsto 1)) \\
&\qquad \text{simplify outer finite map using method for strings} \\
&= \mathbf{cp}_{\times}^{-1}((a \mapsto (\mathbf{left}(p) \mapsto 1) + (\mathbf{right}(3) \mapsto 1) + (\mathbf{left}(p) \mapsto 1)) + (b \mapsto \mathbf{right}(4) \mapsto 1)) \\
&\qquad \text{apply } \mathbf{cp}_{+} \\
&= \mathbf{cp}_{\times}^{-1}((a \mapsto \mathbf{cp}_{+}^{-1}((p \mapsto 1,0) + (0,3 \mapsto 1) + (p \mapsto 1,0))) + (b \mapsto \mathbf{cp}_{+}^{-1}(0,4 \mapsto 1))) \\
&\qquad \text{simplify biproduct by adding pairwise} \\
&= \mathbf{cp}_{\times}^{-1}((a \mapsto \mathbf{cp}_{+}^{-1}((p \mapsto 1) + (p \mapsto 1),3 \mapsto 1)) + (b \mapsto \mathbf{cp}_{+}^{-1}(0,4 \mapsto 1))) \\
&\qquad \text{simplify inner finite map using methods for strings and integers} \\
&= \mathbf{cp}_{\times}^{-1}((a \mapsto \mathbf{cp}_{+}^{-1}(p \mapsto 2,3 \mapsto 1)) + (b \mapsto \mathbf{cp}_{+}^{-1}(0,4 \mapsto 1)))
\end{aligned}
$$

Using this method the simplified form of a finite map can be found in linear time. Compact maps can be dealt with just as easily by exploiting that $A \Rightarrow^{*} U \cong (A \Rightarrow U) \oplus U$. These isomorphisms are the basis of generic tries [10, 12].