A database calculus based on strong monads and partial functions

Eerke Boiten and Paul Hoogendijk*

Draft March '95; ext.abs. submitted to 5th DBPL

1 Introduction

The research areas of database programming languages and formal program development have a lot to offer to each other. In particular, both areas share a great interest in data types – in more theoretical approaches, both have reached into category theory, with its "functor=datatype" motto. So-called bulk- or collection types have been investigated in both research communities.

This paper sketches an approach to calculi for databases which provide multiple data types. The setup has a categorical flavour, but no more than some intuitive understanding of a few basic concepts of category theory is expected of the reader.

The first layer of the calculus consists of operations on basic values that correspond to the traditional tuples of relational databases. These operations are provided by a category with special (viz. associative and semi-commutative) products. The labels of the tuples are provided by the categorical typing.

In the second layer, these operations are lifted to operations on sets, resulting in the traditional relational algebra operators. The collection of operators that allows us to do that turns out to represent an algebra with sets as "first-class citizens", the nested relational algebra. The resulting algebra is, if we abstract from sets, one of strong monads with some additional requirements.

The final section discusses the possibilities of combining and interfacing several such monads.

2 Special products

Textbooks on databases generally present two views on tuples in relational algebra: they can be ordered or labeled. The ordered view corresponds more closely to categorical notions of product but the labeled one is much more convenient for defining operators like join and select. In this approach we aim for the best of both worlds by using a special categorical product.

Let us introduce "normal" categorical products, and in particular: our notation for them, first. We write the typing of arrows in such a way that it looks natural for compositions: if $f : A \leftarrow B$ and $g : B \leftarrow C$, then $f \circ g : A \leftarrow C$.

Definition 1 (Product) For each pair of objects A and B, an object $A \times B$ exists with arrows $\ll_{A,B}$ and $\gg_{A,B}$ such that for all arrows $f : A \leftarrow C$ and $g : B \leftarrow C$ there is a unique arrow called $f \vartriangle g$ with type $A \times B \leftarrow C$ such that for all arrows h of that type we have:

*Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email:{eerke,paulh}@win.tue.nl

$$f \vartriangle g = h \quad \equiv \quad f = \ll_{A,B} \circ h \quad \wedge \quad g = \gg_{A,B} \circ h$$

In the interpretation of arrows as functions, this simply means that $(f \vartriangle g).x = (f.x\,, g.x)$, $\ll_{A,B}.(x\,, y) = x$ and $\gg_{A,B}.(x\,, y) = y$. The reason that we have taken a somewhat unusual notation for the projections is that they are *not* projections as in relational algebra, those being at a higher level (see Section 3). The product operator $\times$ extends to arrows, with $(f \times g).(x\,, y) = (f.x\,, g.y)$, making $\times$ a *(bi-)functor* (check that if $f : A \leftarrow C$ and $g : B \leftarrow D$, then $f \times g : A \times B \leftarrow C \times D$, that $id_A \times id_B = id_{A \times B}$ and that $\times$ distributes through compositions).

It is well known that product is associative "up to isomorphism", i.e. there is a one-to-one mapping between $A \times (B \times C)$ and $(A \times B) \times C$, one half of which is the arrow $ass_{A\,,B\,,C}$ defined by

$$ass_{A\,,B\,,C} : (A \times B) \times C \leftarrow A \times (B \times C)$$

$$ass_{A\,,B\,,C} = (id_A \times \ll_{B,C}) \vartriangle (\gg_{B,C} \circ \gg_{A,B \times C})$$

Having to work with this complicated function all the time is troublesome, and we desperately want to get rid of all the brackets for products. Thus we adopt an extra axiom that declares product to be strictly associative, i.e. up to equality instead of up to isomorphism.[1]

To further smoothen the structure, let us also give product a strict unit element. For that, we need the definition of a terminal object in a category.

Definition 2 (Terminal) In a category the object X is terminal if for every object A in the category there is a unique arrow $!_A : X \leftarrow A$.

Since terminal objects are (like products) unique up to isomorphism, we designate one of those to be *the* terminal object, and denote it by $\mathbb{1}$. If the category has terminal objects, one can prove that the objects A, $\mathbb{1} \times A$ and $A \times \mathbb{1}$ are isomorphic. As additional axioms we will take equality of those isomorphisms to the identity as well.

Axiom 3 (Associative product) $ass_{A\,,B\,,C} = id_{A \times B \times C}$

Axiom 4 (Left and right unit) $!_A \vartriangle id_A = id_A = id_A \vartriangle !_A$

Note that from the typing of the respective arrows it follows that $\times$ is associative on objects as well, this allows the notation $A \times B \times C$ in Axiom 3. Associativity of $\times$ on both objects and arrows implies that $\times$ is now an associative bifunctor as well.

The reader might notice that we now have a product closely resembling a list of its factors - but it is not, since we do not distinguish between lists of length 1 and their elements. Except for a minor detail (viz. that we cannot be sure whether elements of a tuple are products themselves or not) we have now arrived at the "ordered" approach to tuples. We do also have labels, viz. the types of the arrows. To arrive at the "labeled" approach to tuples, we need to introduce one more aspect: that it does not matter in which order the labels occur in a tuple.

We make one small proviso, however. A tuple in which one label occurs twice makes little sense – but a product of some type with itself does. And for values of type $A \times A$ we have no way of distinguishing a pair $(a\,, b)$ from the corresponding pair $(b\,, a)$ except by the order in which the elements occur. Disallowing products of the type $A \times A$ seems awkward, so we introduce the "semi-commutativity" axiom below, which states that values of *different* type may be exchanged in products.

[1] That this is sound practice is proved in [14]: for any category with products an equivalent category with associative products exists.

Axiom 5 (Semi-commutativity) Let $swap_{A,B} = \gg_{A,B} \vartriangle \ll_{A,B}$, the isomorphism between $A \times B$ and $B \times A$. If A and B are *relatively prime*, then $swap_{A,B} = id_{A \times B}$.

The definition of "relatively prime" is postponed a few lines. First we require the existence of unique prime factorizations of all types.

Definition 6 (Prime) An object A is prime iff for all B and C such that $A = B \times C$ either $B = \mathbb{1}$ or $C = \mathbb{1}$.

Definition 7 (Unique prime factorization) An object A has a unique prime factorization iff for all B, C, D, E such that $A = B \times C \wedge A = D \times E$ where B and D are different and prime, a unique F exists such that $C = D \times F$ and $E = B \times F$.

Axiom 8 (UPF) All objects have a unique prime factorization.

These axioms have as a consequence that the following equations uniquely define division / and a divides predicate $\div$:

$$X = A/B \quad \equiv \quad A = B \times X$$

$$A \div B \quad \equiv \quad \exists(X : : A = B \times X)$$

Divisibility $\div$ induces a lattice, greatest common divisor gcd and lcm are glb and lub, respectively, in that lattice. Now two objects are *relatively prime* if their gcd is $\mathbb{1}$, as one would expect.

The result of all this is that product types take a position somewhere between bags and lists. Product is fully commutative on the *objects*, types can be viewed as bags of prime factors, where elements of the factors that occur more than once are ordered, but there is no ordering between the elements of different factors. (This observation also shows that it is indeed possible to construct a model satisfying these axioms.)

Note: the unique prime factorization axiom bears some relation to the 1NF assumption, which would additionally require all factorizations to be *finite*. Prime types at this point seem to correspond to non-composite types; however, this is only because we have considered no type formers besides product (yet).

At this point we can introduce some of the operations of the relational database algebra at the "element" level. To start with, projections $\ll_{A,B}$ and $\gg_{A,B}$ were already defined. Also, if we have predicates we can define selections: the selection predicates can be used as *partial identity* functions that are defined whenever the predicate holds. In particular we have the selection $equal_A : A \times A \leftarrow A \times A$ which is defined exactly on those pairs (a, b) of type $A \times A$ such that $a = b$. As an abbreviation for the inverse function of the duplicating function $id_A \vartriangle id_A$ we use $diag_A = \ll_{A,A} \circ equal_A$. The (right) domain of a partial function f is denoted by the partial identity $f>$ which is defined as the "smallest" partial identity p satisfying $f \circ p = f$.

Now we have introduced partial functions as well, a small remark on products is in order. We extend $\vartriangle$ and $\times$ to partial functions in a strict way, i.e. $f \vartriangle g$ and $f \times g$ are only defined when both f and g are defined. For the corresponding extension of Definition 1 see Section 4 . (Defined in this way, $\times$ is a functor.)

Binary operations are "just" unary operations on a product domain. The natural join between values of types A and B is an operation of type $D \leftarrow A \times B$ where D is the least common multiple of A and B. We define it (as a partial function defined only on a limited subset of types) as follows:

$$join_{A,B} = id_{A/C} \times id_{B/C} \times diag_C$$

where A/C and B/C are both required to be relatively prime with $C = gcd.(A\,,B)$. Note that if A and B are types with all labels different (which is what one would normally expect), then the condition is certainly fulfilled. If A and B are relatively prime, C will be $\mathbb{1}$ and the join will indeed be the Cartesian product.

At this point all standard operators of relational algebra that operate elementwise have been defined at the element (tuple) level. Now we need to move on to sets (tables).

3 Lifting to sets

Informally, we would like to define the operators at the set level as fairly simple set comprehensions. For example projecting a set S that has scheme $A \times B$ on A is:

$$\Pi_{A\,,B}.S = \{\ll_{A,B}.x \mid x \in S\}$$

There is a functor which allows us to describe exactly set comprehensions like this. Consider the covariant powerset functor P, whose operation on objects returns the set of all subsets of an object[2], and whose operation on arrows is the higher order function "apply-to-all" or "map" which we will denote by map_{P} for uniformity with later constructions.[3] Using map_{P} we could define $\Pi_{A\,,B}$ by

$$\Pi_{A\,,B} = map_{\mathsf{P}}.\ll_{A,B}$$

and this is indeed what we want. (Also we have $\Pi\,'_{A\,,B} = map_{\mathsf{P}}.\gg_{A,B}$ which may differ from $\Pi_{B\,,A}$ when A and B are not relatively prime.)

However using map_{P} for comprehensions is not good enough in general. Projections $\ll_{A,B}$ happen to be total on the appropriate domains, but the other operations are partial and that needs to be taken into account. The lifted definition of join as a comprehension is:

$$S \bowtie_{A,B} T = \{join_{A\,,B}.(x\,,y) \mid x \in S \wedge y \in T \wedge \textsc{Defined}.(join_{A\,,B}.(x\,,y))\}$$

where Defined is a meta-predicate accounting for the partiality of *join*. The definition of projection could be stated in this form as well, since there Defined is *true*. Also, if we denote the partial identity function representing a predicate Q by $select_Q$, the lifted definition for selection is also in this form

$$\sigma_Q.S = \{x \mid x \in S \wedge \textsc{Defined}.(select_Q.x)\}$$

when we observe that for appropriate x it holds that $x = select_Q.x$.

The function map_{P} cannot be used for lifting these partial functions. Somehow, we wish to obtain *total* functions on sets, discounting the partialities.

Fortunately there is a construction which allows us to do just that, using some more basic functions on sets. The trick is to "pack" (for each element of the set) the results of the partial function using the singleton set former, return the empty set for values on which the function is undefined, and flatten the resulting set of sets. If we denote "big union" by $flatten_{\mathsf{P}} : \mathsf{P}A \leftarrow \mathsf{PP}A$, singleton former by $unit_{\mathsf{P}} : \mathsf{P}A \leftarrow A$ and the constant function returning the empty set by $zero_{\mathsf{P}}$, we have

$$lift_{\mathsf{P}}.f = flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.\mathsf{Tot}_{\mathsf{P}}.f \quad \text{where } (\mathsf{Tot}_{\mathsf{P}}.f).x = \begin{cases} unit_{\mathsf{P}}.f.x & \text{if } \textsc{Defined}.(f.x) \\ zero_{\mathsf{P}}.x & \text{otherwise} \end{cases}$$

[2] This is only on the level of typing, though. The powerset operator [6] is *not* available as a computational construct in our setting.

[3] We have moved away from real abstract category theory here by mentioning the "subsets" of an object. A categorically sound definition of the powerset functor exists, see [5] or some other textbook. Presenting it here would not be helpful for understanding the construction, however.

There are a few nice properties of $lift_{\mathsf{P}}$ that are easy to prove:

$$lift_{\mathsf{P}}.id_A = id_{\mathsf{P}A} \tag{9}$$

$$lift_{\mathsf{P}}.f \circ lift_{\mathsf{P}}.g = lift_{\mathsf{P}}.(f \circ g) \tag{10}$$

$$lift_{\mathsf{P}}.f = map_{\mathsf{P}}.f \quad \text{if } f \text{ is a total function} \tag{11}$$

Note that (9) is also a consequence of (11), and that (9) and (10) make $lift_{\mathsf{P}}$ (the arrow part of) a functor that coincides with P on objects.[4]

The nicest thing about $lift_{\mathsf{P}}$ being a functor is basically that it is possible to use (10): compositions on the set level are built up from compositions on the element level, and thus we can profit optimally from the calculus for special products and related partial functions which we used before.

Now we have a definition for selections:

$$\sigma_Q = lift_{\mathsf{P}}.select_Q$$

but we are not ready to present the lifted version of join yet. The function $lift_{\mathsf{P}}.join_{A,B}$ has type $\mathsf{P}C \leftarrow \mathsf{P}(A \times B)$ for some type C, but $\bowtie_{A,B}$ has type $\mathsf{P}C \leftarrow \mathsf{P}A \times \mathsf{P}B$. We need a transformation from $\mathsf{P}A \times \mathsf{P}B$ to $\mathsf{P}(A \times B)$ now, viz. the "crossproduct". For this, we will use one more basic function on sets: the function that pairs all elements of a set with one particular value. In category theory, this function is called the *strength* of the powerset functor. Defined as a comprehension, it is

$$str_{\mathsf{P}}.(S, x) = \{(y, x) \mid y \in S\}$$

In general the strength of a functor F is a natural transformation from $\mathsf{F}A \times B$ to $\mathsf{F}(A \times B)$, which has interesting links with the concept of *membership* [4]. A related function, definable in terms of the strength, is $stl_{\mathsf{P}}.(x, S) = \{(x, y) \mid y \in S\}$:

$$stl_{\mathsf{P},A,B} = map_{\mathsf{P}}.swap_{B,A} \circ str_{\mathsf{P},B,A} \circ swap_{A,\mathsf{P}B}$$

The crossproduct can then be computed by two applications of these functions: one of each, one nested. (At the level of sets, it does not matter which one is chosen in which position, since there is no order between the elements.)[5]

$$cross_{\mathsf{P}} = flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.stl_{\mathsf{P}} \circ str_{\mathsf{P}}$$

With this we can define join at the set level:

$$\bowtie_{A,B} = lift_{\mathsf{P}}.join_{A,B} \circ cross_{\mathsf{P},A,B}$$

At this point all standard operators of relational algebra that operate elementwise have been lifted to sets. Finally, there are the "normal" set operations. Union is not definable with the current set of primitive operations so we add a primitive $union_{\mathsf{P}}$ with one of its characteristic properties: that it forms a monoid with $zero_{\mathsf{P}}$. Intersection can be defined using cross-product:

$$\cap_A = lift_{\mathsf{P}}.diag_A \circ cross_{\mathsf{P},A,A}$$

and set difference is also expressible using equality (in fact by encoding boolean *false* as the empty set[6]):

$$-_A = lift_{\mathsf{P}}.(\ll_{A,\mathsf{P}A} \circ (\ll_{A,\mathsf{P}A} \vartriangle (diag_{\mathsf{P}A} \circ (zero_{\mathsf{P},A} \vartriangle (lift_{\mathsf{P}}.diag_A \circ stl_{\mathsf{P},A,A}))))) \circ str_{\mathsf{P},A,\mathsf{P}A}$$

[4] Category theorists may notice that $\mathsf{Tot}_{\mathsf{P}}$ is also a functor, viz. from the base category into the corresponding Kleisli category. They might also be interested to know that $\mathsf{Tot}|\mathsf{F}$ can for any F easily be expressed without applications and case distinctions if the base category is a boolean division allegory [5].

[5] Note that here follows another Kleisli composition.

[6] Even though we allowed equality test for all types, it must be taken into account that it is an expensive operation on sets.

4 Sets as first class objects: the nested relational calculus

In the previous sections we built up a collection of operations that allowed us to define the flat relational calculus in a "categorical" way. From now on we will concentrate on the categorical calculus and investigate what that calculus can express, and how we can generalize it to other datatypes besides sets.

Let us summarize what functions belong to the calculus. A, B, C are assumed to be objects in the category, and they are (like all free variables in the premises) implicitly universally quantified. After each set of constructs is a set of equalities that (are demanded to) hold. The choice for exactly this set of axioms is motivated by category theory (the axioms are such that they characterize the categorical constructs that are mentioned in the boldface labels) and other presentations of this calculus [13, 2]. Of course these axioms hold for the interpretation we have given. In order to save on typing constraints in the axioms, it is assumed that all arrows are typed in a way that makes the compositions and other constructs well-defined. Arrows f and g are arbitrary, p denotes a partial identity arrow.[7]

Category

$$\frac{}{id_A : A \leftarrow A} \qquad \frac{f : A \leftarrow B \ , \ g : B \leftarrow C}{f \circ g : A \leftarrow C}$$

$$id_B \circ f = f \circ id_A = f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

Domain

$$\frac{f : A \leftarrow B}{f\!> : B \leftarrow B}$$

$$f \circ p = f \equiv f\!> \circ p = f\!>$$
$$f\!> \circ g = g \circ (f \circ g)\!>$$

Product

$$\frac{f : A \leftarrow C \ , \ g : B \leftarrow C}{f \vartriangle g : A \times B \leftarrow C} \qquad \frac{}{\ll_{A,B} : A \leftarrow A \times B} \qquad \frac{}{\gg_{A,B} : B \leftarrow A \times B}$$

$$f \vartriangle g = h \ \equiv \ f \circ g\!> = \ll_{A,B} \circ h \ \wedge \ g \circ f\!> = \gg_{A,B} \circ h$$

Terminal

$$\frac{}{!_A : \mathbb{1} \leftarrow A}$$

$$f : \mathbb{1} \leftarrow A \ \equiv \ f = !_A$$

Associative semicommutative product

$$(id_A \times \ll_{B,C}) \vartriangle (\gg_{B,C} \circ \gg_{A,B \times C}) = id_{A \times B \times C}$$
$$id_A \vartriangle !_A = !_A \vartriangle id_A = id_A$$
$$\gg_{A,B} \vartriangle \ll_{A,B} = id_{A \times B} \Leftarrow A \text{ and } B \text{ are relatively prime}$$

Equality test

$$\frac{}{equal_A : A \times A \leftarrow A \times A}$$

$$\ll_{A,A} \circ equal_A = \gg_{A,A} \circ equal_A$$
$$equal_A \circ (id_A \vartriangle id_A) = id_A \vartriangle id_A$$

Functor(P)

$$\frac{f : A \leftarrow B \quad \wedge \quad f\!> = id_B}{map_{\mathsf{P}}.f : \mathsf{P}A \leftarrow \mathsf{P}B} \qquad \frac{f : A \leftarrow B}{lift_{\mathsf{P}}.f : \mathsf{P}A \leftarrow \mathsf{P}B}$$

$$map_{\mathsf{P}}.(f \circ g) = map_{\mathsf{P}}.f \circ map_{\mathsf{P}}.g$$
$$map_{\mathsf{P}}.id_A = id_{\mathsf{P}A}$$

[7] Characterisation of partial identities is left out here.

$$lift_{\mathsf{P}}.(f \circ g) = lift_{\mathsf{P}}.f \circ lift_{\mathsf{P}}.g$$
$$lift_{\mathsf{P}}.id_A = id_{\mathsf{P}A}$$
$$map_{\mathsf{P}}.f = lift_{\mathsf{P}}.f \Leftarrow f\!> = id$$

Monad(P) $\dfrac{}{flatten_{\mathsf{P},A} : \mathsf{P}A \leftarrow \mathsf{PP}A}$ $\qquad$ $\dfrac{}{unit_{\mathsf{P},A} : \mathsf{P}A \leftarrow A}$ $\qquad$ $\dfrac{f : \mathsf{P}B \leftarrow A}{ext_{\mathsf{P}}.f : \mathsf{P}B \leftarrow \mathsf{P}A}$

$$ext_{\mathsf{P}}.f = flatten_{\mathsf{P},B} \circ map_{\mathsf{P}}.f$$
$$ext_{\mathsf{P}}.flatten_{\mathsf{P},A} = flatten_{\mathsf{P},A} \circ flatten_{\mathsf{P},\mathsf{P}A}$$
$$ext_{\mathsf{P}}.unit_{\mathsf{P},A} = flatten_{\mathsf{P},A} \circ unit_{\mathsf{P},\mathsf{P}A} = id_{\mathsf{P}A}$$
$$unit_{\mathsf{P},B} \circ f = map_{\mathsf{P}}.f \circ unit_{\mathsf{P},A}$$
$$ext_{\mathsf{P}}.(map_{\mathsf{P}}.f) = map_{\mathsf{P}}.f \circ flatten_{\mathsf{P},A}$$

Monoid(P) $\dfrac{}{zero_{\mathsf{P}} : \mathsf{P}A \leftarrow B}$ $\qquad$ $\dfrac{}{union_{\mathsf{P}} : \mathsf{P}A \leftarrow \mathsf{P}A \times \mathsf{P}A}$

$$union_{\mathsf{P}} \circ (union_{\mathsf{P}} \times id) = union_{\mathsf{P}} \circ (id \times union_{\mathsf{P}})$$
$$zero_{\mathsf{P}} \circ ! = zero_{\mathsf{P}}$$
$$union_{\mathsf{P}} \circ (zero_{\mathsf{P}} \vartriangle id) = id$$
$$union_{\mathsf{P}} \circ (id \vartriangle zero_{\mathsf{P}}) = id$$
$$map_{\mathsf{P}}.f \circ zero_{\mathsf{P}} = zero_{\mathsf{P}}$$
$$ext_{\mathsf{P}}.zero_{\mathsf{P}} = zero_{\mathsf{P}}$$
$$ext_{\mathsf{P}}.f \circ union_{\mathsf{P}} = union_{\mathsf{P}} \circ (ext_{\mathsf{P}}.f \times ext_{\mathsf{P}}.f)$$

Strength(P) $\dfrac{}{str_{\mathsf{P},A,B} : \mathsf{P}(A \times B) \leftarrow \mathsf{P}A \times B}$

$$str_{\mathsf{P}} \circ map_{\mathsf{P}}.f \times g = map_{\mathsf{P}}.(f \times g) \circ str_{\mathsf{P}}$$
$$str_{\mathsf{P},A,\mathbb{1}} = id_A$$
$$str_{\mathsf{P},A,B \times C} = str_{\mathsf{P},A \times B,C} \circ str_{\mathsf{P},A,B} \times id_C$$
$$str_{\mathsf{P},A,B} \circ (unit_{\mathsf{P},A} \times id_B) = unit_{\mathsf{P},A \times B}$$
$$str_{\mathsf{P},A,B} \circ (flatten_{\mathsf{P},A} \times id_B) = flatten_{\mathsf{P},A \times B} \circ map_{\mathsf{P}}.str_{\mathsf{P},A,B} \circ str_{\mathsf{P},\mathsf{P}A,B}$$

This collection of operations completes our calculus for sets. The expressive power of this calculus is that of the nested relational algebra [10] with equality test, as is shown in the presentation of this calculus in [2]. The nesting and unnesting operators, specified using comprehensions, are:

$$nest.S = \{(x, T) \mid T = \{y \mid (x, y) \in S\} \wedge T \neq \emptyset\}$$
$$unnest.T = \{(x, y) \mid \exists (S : (x, S) \in T : y \in S)\}$$

In our calculus they are the beauty:

$$unnest_{\mathsf{P},A,B} = ext_{\mathsf{P}}.stl_{\mathsf{P},A,B}$$

and the beast (translated from [11]):

$$nest_{\mathsf{P},A,B} = map_{\mathsf{P}}.(\ll_{A,\mathsf{P}(A\times B)} \vartriangle f) \circ str_{\mathsf{P},A,\mathsf{P}(A\times B)} \circ (\Pi_{A,B} \vartriangle id_{\mathsf{P}(A\times B)}) \quad \text{where}$$
$$f = \Pi'_{A\times A,B} \circ lift_{\mathsf{P}}.(equal_A \times id_B) \circ stl_{\mathsf{P},A,A\times B}$$

Note that this nested algebra came about naturally from the wish to express the standard operations of relational algebra as extensions of operations on tuples. That we are now able to express the entire nested relational algebra tells us that this algebra is in a sense the smallest orthogonal extension of the flat relational algebra, or in other words, that "first normal form" is a rather artificial restriction in this setup.

5 Collection types

From our insistence to write the conventional operators on sets as operations on a monad it should come as no surprise that everything above can be generalized to an arbitrary monad with the right additional properties, as specified in the laws given in the summary of the calculus above. Indeed we already wrote **Functor(P)**, **Monad(P)**, etcetera, to suggest that these constructs and laws could be viewed as being parameterized by an arbitrary functor P. Following Wadler [13] we could call this structure a *strong ringad*.

Trinder [12] and Tannen et al [2] have argued that this collection of operations forms the essence of so-called *bulk-* or *collection types*. Indeed besides sets, also *lists* and *bags (multisets)* with the obvious operations satisfy these laws. (Note, for example, that commutativity and idempotence of $union_{\mathsf{P}}$, i.e. the properties that differentiate sets from bags and lists, played no rôle so far.)

However, not all operations defined above generalize from sets to other collection types all that well. Consider for example nesting/unnesting on lists, where L is the list functor. For sets we had that $unnest_{\mathsf{P}} \circ nest_{\mathsf{P}} = id$. For lists $nest_{\mathsf{L}}$ is an injective function, but $unnest_{\mathsf{L}}$ is not its left inverse:

$$
\begin{array}{ll}
 & unnest_{\mathsf{L}}.nest_{\mathsf{L}}.[(a,5),(b,1),(a,7)] \\
= & \quad \{ \text{ definition } nest_{\mathsf{L}} \ \} \\
 & unnest_{\mathsf{L}}.[(a,[5,7].),(b,[1]),(a,[5,7])] \\
= & \quad \{ \text{ definition } unnest_{\mathsf{L}} \ \} \\
 & [(a,5),(a,7),(b,1),(a,5),(a,7)]
\end{array}
$$

One could argue that either the nest operation on lists should discard the duplicates (making it no longer injective since order information is lost) or that unnesting should do so (which means that unnest no longer has a simple definition).

6 Interfacing monads, partial information

Now the point has come where we need to tie all this up and move towards our announced goal: a core query language that includes multiple datatypes. The crucial aspect in that is to have expressions in which the various datatypes interact. So far we have only shown that we can have the same calculus for each strong ringad separately. Below we will discuss various ways of changing the datatypes, first concentrating on $\mathsf{G} \leftarrow \mathsf{F}$ for strong ringads F and G, and later on $\mathsf{GF} \leftarrow \mathsf{FG}$.

A rather pessimistic approach to this issue is based on the observation that all well-known examples of collection types can be viewed as implementations of sets. This implies that for each such type F a polymorphic function (i.e., a natural transformation) $setify_{\mathsf{F}} : \mathsf{P}A \leftarrow \mathsf{F}A$ exists. We could impose the existence of *setify* as an additional requirement on collection types[8]. But even on the known examples there are more useful operations, e.g. there is also a polymorphic transformation from lists to bags. Generalizing this we could end up with something like a category of collection types, with polymorphic data type transformations as arrows and possibly the powerset type P as a terminal object.

Having polymorphic conversion functions between arbitrary datatypes would be just perfect, but is also asking for too much. For example, to define a deterministic conversion from bags to lists one needs an order on the element type, which need not exist. As can be checked by type analysis

[8] The paper [4] gives a definition of $setify_{\mathsf{F}}$ in terms of the membership relation of F , which is strongly related to the strength of F. So, within the current context, this may not be a severe restriction.

on the available constructs, the conversions that do exist are not definable in the current query language, they will need to be added explicitly.

A different class of useful operations for interfacing several datatypes is formed by operations that commute functors, i.e. that convert FG-structures into GF-structures. Our group has studied such operations [1], and called them "zips", after the well known operator that turns a pair of lists of equal length into a list of pairs. The special case where G is the powerset functor P has been studied by de Moor [3] under the name of "cross operators". Most zips are not definable in our language, an exception being $zip_{\mathsf{P},\times}$, which is identical to $cross_{\mathsf{P}}$ on its domain.[9] (In general $cross_{\mathsf{F}}$ is not related to $zip_{\mathsf{F},\times}$ though, as can be observed for F being the list functor L.) Another interesting example is $zip_{\mathsf{F},\mathsf{F}}$ which is a kind of transpose operator: $zip_{\mathsf{L},\mathsf{L}}$ turns a list of m lists of length n into a list of n lists of length m in the obvious way.[10]

A final example of the use of zips is in the approach to query languages for databases with partial information advocated by Libkin and Wong[9, 8]. They combine a collection type (sets or bags) with a version of sets that has a non-standard interpretation (but is otherwise identical), the so-called or-sets. An or-set conceptually represents one of the values in it. For normalisation of expressions containing tuples, ordinary sets/bags and or-sets, the *flatten* and *str* operations for or-sets can be used (check that these preserve the conceptual meaning of such an expression). At the heart of normalisation is an operation called α of type $\mathsf{Or}(\mathsf{P}A) \leftarrow \mathsf{P}(\mathsf{Or}A)$ which essentially translates conjunctive normal form into disjunctive normal form. It can be defined in terms of the corresponding operator for bags, $b_\alpha : \mathsf{Or}(\mathsf{Bag}A) \leftarrow \mathsf{Bag}(\mathsf{Or}A)$ which is $zip_{\mathsf{Bag},\mathsf{Or}}$ on bags of non-empty or-sets. It should be noted, though, that unlike any of the other operators, *zip* can have a complexity that is exponential in the size of its arguments.

Altogether, it seems that more research is needed into zips and other data type transformations before a more conclusive form of a query language with multiple collection types can be established. We expect that research done in this group on zips [1] and on the so-called Boom hierarchy [7] of datatypes will provide a solid basis for this further research.

7 Summary and conclusions

Monads with strength and a monoid structure imposed on them are a nice way to structure a basic query language. This is by no means a new observation. New in our presentation, however, is how this calculus is motivated. From the desire to incorporate the traditional flat relational algebra into the language the necessary operators come out naturally[11]. The repertoire of operations that results is the nested relational algebra with equality test. That algebra can be parametrised on the monad involved, resulting in an algebra for general collection types. However, not all operations generalise that well from sets to arbitrary monads, as we have shown with the example of nest/unnest on lists.

The most interesting aspect of this algebra is the prospect of combining multiple datatypes in one query language. This, however, needs much more research.

Finally we wish to point out that, even though we gave a specific model for the constructions here, viz. going from tuples to tables, because of the categorical setup, the construction can be repeated at any level. The elements of a tuple could be values of a collection type, and also we could construct databases as tuples of tables this way.

[9] The crossproduct of one empty and one non-empty set is empty, but the zip of those two is undefined since they are not of the same shape.

[10] For functors that preserve pullbacks and covers, e.g. polynomial functors, but not P, we have proved that $zip_{\mathsf{F},\mathsf{G}}$ is an arrow f such that $f \circ map_{\mathsf{F}}.stl_{\mathsf{G}} \circ str_{\mathsf{F}} = map_{\mathsf{G}}.str_{\mathsf{F}} \circ stl_{\mathsf{G}}$. This clearly relates the definition of *cross* with the alternative one where "the other index runs faster".

[11] Pun intended.

Acknowledgements

We would like to thank Val Tannen for encouraging us to write this paper when he visited Eindhoven last year, and for providing us with a few interesting papers and references. We thank our colleagues in the Mathematics of Program Construction section for their comments on this paper and on presentations we have given of this material. Also, we thank Leonid Libkin for his quick reaction to our questions and his comments.

References

[1] R.C. Backhouse, H. Doornbos, and P. Hoogendijk. A class of commuting relators. In *Lecture Notes of the STOP 1992 Summer School on Constructive Algorithmics, Ameland*. STOP, 1992. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`.

[2] Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *Proceedings of International Conference on Database Theory*, volume 646 of *Lecture Notes in Computer Science*, pages 140–154, Berlin, 1992. Springer-Verlag.

[3] O. de Moor. *Categories, Relations and Dynamic Programming*. PhD thesis, Oxford University Laboratory, Programming Research Group, April 1992.

[4] Peter Freyd, Paul Hoogendijk, and Oege de Moor. Membership of datatypes. Unpublished manuscript, 1993.

[5] P.J. Freyd and A. Scedrov. *Categories, Allegories*. North-Holland, 1990.

[6] Marc Gyssens and Dirk Van Gucht. A comparison between algebraic query languages for flat and nested databases. *Theoretical Computer Science*, 87:263–286, 1991.

[7] P.F. Hoogendijk and R.C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Science of Computer Programming*, 22:67–105, 1994.

[8] L. Libkin. Normalizing incomplete databases. In *PODS-95*, 1995.

[9] L. Libkin and L. Wong. Semantic representations and query languages for or-sets. In *PODS-93*, pages 155–166, 1993.

[10] H.-J. Schek and M.H. Scholl. The relational model with relation-valued attributes. *Information systems*, 11(2):137–147, 1986.

[11] Val Tannen. Tutorial: Languages for collection types. Slides for the 13th ACM Conference on Principles of Database Systems. Available via World Wide Web on `file://ftp.cis.upenn.edu/pub/papers/db_research/pods94t_slide.ps.Z`, 1994.

[12] P.W. Trinder. Comprehensions – a query notation for DBPLs. In *Proceedings of the 1990 Glasgow Database Workshop*, pages 95–102, Glasgow, Scotland, March 1990.

[13] Philip Wadler. Notes on monads and ringads. Unpublished note, 1990.

[14] R.F.C. Walters. The free category with products on a multigraph. *J. Pure & Applied Algebra*, 62:199–203, 1989.