Nested collections and polytypism

Eerke Boiten[*] and Paul Hoogendijk[†]

November 1996

Abstract

A point-free calculus of so-called "collection types" is presented, similar to the monadic calculus of Tannen, Buneman and Wong. We observe that our calculus is parametrised by a monad thus making the calculus "polytypic". A novel contribution of the paper is to discuss situations in which a single application involves more than one collection type. In particular, we outline the contribution to database research that may be obtained by exploiting current developments in polytypic programming.

1 Introduction and overview

"Collection types" such as trees, lists, and bags have been studied extensively in computing science. In particular, in the research area of formal program development, the observation (attributed by L.Meertens [19] to H.Boom) that these types form a hierarchy has proved fruitful. The most important aspect of this so-called Boom hierarchy is that a calculus of higher order functions (like map, reduce and filter) can be defined on all its types. This calculus is commonly known as the Bird-Meertens Formalism (BMF) [19, 5], and it is widely used for the development and description of functional and parallel [21] programs. A generalisation of this calculus was found in the category theoretic [18] and relational [14] approaches to abstract data types, and it was observed that the Boom hierarchy types form an instance of another popular category theoretic concept, the *monad*. This provided a new syntax and calculus for comprehensions on these types.

In the area of databases, the interest in these types arises from the quest for query languages for databases containing structured data. The traditional "flat" relational database model [8] only describes sets of tuples, whose attributes are assumed to be of atomic type

[*]Now at Computing Laboratory, University of Kent, Canterbury Kent CT2 7NF, UK, email: `E.A.Boiten@ukc.ac.uk`; research was carried out at Eindhoven University of Technology.

[†]Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email:`paulh@win.tue.nl`.

(the so-called First Normal Form). Various "nested" relational calculi[1] (i.e. calculi providing for set-valued attributes) have been proposed [20, 2, 9]. The most general of these is the "monadic" calculus described by Tannen, Buneman and Wong [6, 22]. (The observation that collection types "are" monads with additional properties ("ringads") can be attributed to Wadler [24] and Trinder [23].) They prove that their calculus (using the set monad) is equivalent to "the" nested relational calculus [9].

The calculus we present here is inspired by (and can be instantiated to) that "monadic calculus". Thus, this paper does not claim to directly advance research in database query languages. Rather, our intent is to present the "state of the art" in that area in such a way that it connects more easily with recent developments in formal program development. This explains the differences between our calculus and the monadic calculus: our calculus is non-extensional ("point-free") to facilitate equational reasoning; the monad (functor) involved is an explicit parameter; and instead of an underlying signature of basic functions we assume a category of partial functions with a few extra properties. The importance of these differences has to do with the emerging interest in the area of so-called polytypic programs, the current focus of research in the BMF [3, 4, 11, 15, 12]. Such programs are parametrised by type constructors (as opposed to polymorphic programs which are parametrised by types). Beginning with the work of Malcolm [18] it has been observed that several programming concepts and building blocks can be profitably formulated in polytypic terms thus enhancing their (re)usability.

The presentation of the elements of our calculus is such that it amounts to a constructive proof that it is (in our setting) the smallest orthogonal calculus that can describe the flat relational calculus – or, in other words, our calculus (instantiated with the set monad) *is* the extension of the flat relational calculus with sets as "first class citizens". The presentation is in two layers: the first layer identifies "tuples" and the basic operations on them in a category of partial functions with a special product. The second layer "lifts" these operations to operations on sets of tuples. Partiality of the basic operations requires some special attention in this lifting procedure.

In the final section of this paper we describe issues for further research, and their relation to research issues in formal program development.

2 Special products: the tuple operations

In relational database theory, a tuple is either a function from a set of labels to a set of "values", or a member of a product type. We choose the latter approach, confident that the strict categorical typing will provide the labels.

We work in a category of partial functions, and 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 $f \circ g : A \leftarrow C$. Sets are represented as identity arrows, as usual. In particular, for each arrow $f : A \leftarrow B$

[1]Any reference to "relational calculus" in this paper (except for this footnote...) should be taken to refer to *relational database* calculus (and the corresponding algebra) rather than Tarski's calculus of binary relations, even though that plays a crucial rôle in polytypic programming.

we assume the existence of an arrow $f{>} : B \leftarrow B$ ("domain") which equals the identity on $f's$ domain and is undefined elsewhere. We assume a particular product, whose unique mediating arrow (written as $f \vartriangle g$) is characterised by the following

Axiom 1 (Product) $\quad f \vartriangle g = h \Leftrightarrow f \circ g{>} = \ll_{A,B} \circ\, h \wedge g \circ f{>} = \gg_{A,B} \circ\, h$

assuming that $f : A \leftarrow C$, $g : B \leftarrow C$, with $\ll_{A,B}$ and $\gg_{A,B}$ the projections on $A \times B$. We make product a bifunctor by setting $f \times g = (f \circ \ll) \vartriangle (g \circ \gg)$. We take $\mathbb{1}$ to be a terminal object in the category, with $!_A$ the unique total arrow of type $\mathbb{1} \leftarrow A$.

In database theory, tuples are elements of n-ary products for arbitrary n. To define unique n-ary products, we make the binary product associative with unit element $\mathbb{1}$. This is axiomatised by equating the relevant isomorphisms to the appropriate identities:

Axiom 2 $\quad ass_{A,B,C} =_{def} (id_A \times \ll_{B,C}) \vartriangle (\gg_{B,C} \circ \gg_{A,B \times C}) = id_{A \times B \times C}$

Axiom 3 $\quad !_A \vartriangle id_A = id_A = id_A \vartriangle !_A$

Arrows can only be equal if their types are; thus $\times$ is associative on objects as well, which justifies writing $A \times B \times C$ without brackets above.

From the desire to address *any* field of an n-ary product directly (for example the B field in $A \times B \times C$) with a single projection it follows that product should be commutative as well. However, if there are multiple fields of one and the same type it seems likely that we would wish to distinguish those. So we introduce the following

Axiom 4 (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 $swap_{A,B} = id_{A \times B}$.

Two types are relatively prime when their greatest common divisor is $\mathbb{1}$. In order to be able to define division, /, and greatest common divisors, we complete our axiomisation of product by postulating:

Axiom 5 All objects have a unique prime factorisation.

Now we can address any field directly provided it occurs once: the projection on B in A is $\ll_{B,A/B}$. Since sets are represented by partial identity arrows, it makes sense also to represent predicates for selection ("filtering") in the same way. In particular, we assume the existence of $equal_A : A \times A \leftarrow A \times A$ which is defined exactly on those pairs (a,b) of type $A \times A$ such that $a = b$. As an abbreviation for the inverse of the duplicating function $id_A \vartriangle id_A$ we use $diag_A = \ll_{A,A} \circ equal_A$. These give us the building blocks for projection and selection on the level of sets in the next section. A final basic operation is (natural) *join* of two tuples, which is only defined if they have matching values for fields of the same label (type), and in that case contains the combination of all their fields. Using division, we define it as

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

where C is the greatest common divisor of A and B, and A/C and B/C are both required to be relatively prime with C. The result type of $join_{A,B}$ is the least common multiple of A and B. For A and B relatively prime, the join equals the Cartesian product, as one might expect.

3 Second layer: lifting to sets

Intuitively, we would like to define the operators at the set level as fairly simple set comprehensions. For example, projecting a set S over the type $A \times B$ on A is (using . for function application):

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

The "map" or "apply-to-all" operation on sets will do exactly this, i.e. $\Pi_{A,B} = \mathit{map}_{\mathsf{P}}.\ll_{A,B}$ where we index the map operation with the functor involved, here it is P, for powerset. (Also we have $\Pi'_{A,B} = \mathit{map}_{\mathsf{P}}.\gg_{A,B}$ which may differ from $\Pi_{B,A}$ when A and B are not relatively prime.)

However using $\mathit{map}_{\mathsf{P}}$ for comprehensions is not good enough in general. For total operations, like projections $\ll_{A,B}$ there is no problem. However, some of the other functions we would like to lift to the level of sets are partial (for example, predicates encoded as partial identity functions) and that needs to be taken into account. The lifted definition of selection as a comprehension is (assuming select_Q is the partial identity representing the predicate Q)

$$\sigma_Q.S = \{x \mid x \in S \wedge \textsc{Def}.(\mathit{select}_Q.x)\}$$

where $\textsc{Def}$ is a meta-predicate accounting for the partiality of select_Q. The function $\mathit{map}_{\mathsf{P}}$ would produce the wrong result if we used it to lift such functions to the level of sets: it would deliver functions that are undefined whenever any of the elements in the set does not satisfy the predicate.

Using some more basic functions on sets we can resolve this problem. The results of the partial function will be "packed" using the singleton set former ($\mathit{unit}_{\mathsf{P}} : \mathsf{P}A \leftarrow A$), we return the empty set ($\mathit{zero}_{\mathsf{P}} : \mathsf{P}A \leftarrow B$) for values on which the function is undefined, and flatten ($\mathit{flatten}_{\mathsf{P}} : \mathsf{P}A \leftarrow \mathsf{PP}A$) the resulting set of sets:

$$\mathit{lift}_{\mathsf{P}}.f = \mathit{flatten}_{\mathsf{P}} \circ \mathit{map}_{\mathsf{P}}.\mathsf{Tot}_{\mathsf{P}}.f$$

where

$$(\mathsf{Tot}_{\mathsf{P}}.f).x = \begin{cases} \mathit{unit}_{\mathsf{P}}.f.x & \text{if } \textsc{Def}.(f.x) \\ \mathit{zero}_{\mathsf{P}}.x & \text{otherwise} \end{cases}$$

It is easy to prove that this makes $\mathit{lift}_{\mathsf{P}}$ (the arrow part of) a functor that coincides with $\mathit{map}_{\mathsf{P}}$ on objects and total functions.[2] $\mathit{lift}_{\mathsf{P}}$ being a functor is a kind of healthiness criterion: it means we can use equational reasoning on the level of function compositions for expressions involving $\mathit{lift}_{\mathsf{P}}$, in particular distribution of functors over composition.

Using $\mathit{lift}_{\mathsf{P}}$, we can define selection by $\sigma_Q = \mathit{lift}_{\mathsf{P}}.\mathit{select}_Q$. For join, however, we need to assume one more basic function. The function $\mathit{lift}_{\mathsf{P}}.\mathit{join}_{A,B}$ has type $\mathsf{P}C \leftarrow \mathsf{P}(A \times B)$ for some type C, but the natural join $\bowtie_{A,B}$ 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". This will be defined using the

[2]Actually, $\mathsf{Tot}_{\mathsf{P}}$ is also a functor, viz. from the base category into the corresponding Kleisli category. $\mathsf{Tot}_{\mathsf{F}}$ can for any F be expressed without applications and case distinctions if the base category is a boolean division allegory [13].

function that pairs all elements of a set with one particular value, also known as the *strength* of the powerset functor. Defined as a comprehension, it is

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

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 $\mathit{stl}_{\mathsf{P}}.(x\,,S) = \{(x\,,y) \mid y \in S\}$:

$$\mathit{stl}_{\mathsf{P}\,,A\,,B} = \mathit{map}_{\mathsf{P}}.\mathit{swap}_{B\,,A} \circ \mathit{str}_{\mathsf{P}\,,B\,,A} \circ \mathit{swap}_{A\,,\mathsf{P}B}$$

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.)[3]

$$\mathit{cross}_{\mathsf{P}} = \mathit{flatten}_{\mathsf{P}} \circ \mathit{map}_{\mathsf{P}}.\mathit{stl}_{\mathsf{P}} \circ \mathit{str}_{\mathsf{P}}$$

With this we can define join at the set level:

$$\bowtie_{A,B} \; = \mathit{lift}_{\mathsf{P}}.\mathit{join}_{A\,,B} \circ \mathit{cross}_{\mathsf{P}\,,A\,,B}$$

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

$$\cap_A = \mathit{lift}_{\mathsf{P}}.\mathit{diag}_A \circ \mathit{cross}_{\mathsf{P}\,,A\,,A}$$

and set difference is also expressible using equality (in fact by encoding boolean *false* as the empty set[4]). Let *notmem* be the partial identity function which is only defined on pairs of an element and a set such that the element is not in the set:

$$\mathit{notmem} = (\mathit{diag}_{\mathsf{P}A} \circ (\mathit{zero}_{\mathsf{P}\,,A} \vartriangle (\mathit{lift}_{\mathsf{P}}.\mathit{diag}_A \circ \mathit{stl}_{\mathsf{P}\,,A\,,A})))_{>}$$

we then have that

$$-_A = \mathit{lift}_{\mathsf{P}}.(\ll_{A,\mathsf{P}A} \circ \mathit{notmem}) \circ \mathit{str}_{\mathsf{P}\,,A\,,\mathsf{P}A}$$

[3]Note that here follows another Kleisli composition.

[4]Even though we allowed equality test for all types, it must be taken into account that it is an expensive operation on sets. Comparing to the empty set is still "cheap".

4 The complete calculus (a single monad)

Let us summarise the calculus as we have defined it thus far.

The first layer consists of a category of partial functions with a special product (associative with unit $\mathbb{1}$, semi-commutative, unique prime factorisation), which includes arrows $equal_A$ for all objects A, and $f{>}$ for all arrows f.

The second (monadic) layer consists of

– a functor P from the category of the first layer into a category of total functions, whose arrow part is denoted by $lift_{\mathsf{P}}$;

– operations $unit_{\mathsf{P}} : \mathsf{P}A \leftarrow A$ and $flatten_{\mathsf{P}} : \mathsf{P}A \leftarrow \mathsf{PP}A$.

Because these operations on sets satisfy the relevant properties for all $f : B \leftarrow A$:

$$flatten_{\mathsf{P},A} \circ map_{\mathsf{P}}.flatten_{\mathsf{P},A} = flatten_{\mathsf{P},A} \circ flatten_{\mathsf{P},\mathsf{P}A}$$

$$flatten_{\mathsf{P},A} \circ map_{\mathsf{P}}.unit_{\mathsf{P},A} = flatten_{\mathsf{P},A} \circ unit_{\mathsf{P},\mathsf{P}A} = id_{\mathsf{P}A}$$

$$unit_{\mathsf{P},B} \circ f = map_{\mathsf{P}}.f \circ unit_{\mathsf{P},A}$$

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

the structure up to this point is a *monad*.

Additionally, we have

– an operation $str_{\mathsf{P},A,B} : \mathsf{P}(A \times B) \leftarrow (\mathsf{P}A) \times B$

Because this operation on sets satisfies the relevant properties:

$$str_{\mathsf{P}} \circ (map_{\mathsf{P}}.f) \times g = map_{\mathsf{P}}.(f \times g) \circ str_{\mathsf{P}}$$

$$str_{\mathsf{P},A,\mathbb{1}} = id_A$$

$$str_{\mathsf{P},A,B\times C} = str_{\mathsf{P},A\times B,C} \circ (str_{\mathsf{P},A,B} \times id_C)$$

$$str_{\mathsf{P},A,B} \circ (unit_{\mathsf{P},A} \times id_B) = unit_{\mathsf{P},A\times B}$$

$$str_{\mathsf{P},A,B} \circ (flatten_{\mathsf{P},A} \times id_B) = flatten_{\mathsf{P},A\times B} \circ map_{\mathsf{P}}.str_{\mathsf{P},A,B} \circ str_{\mathsf{P},\mathsf{P}A,B}$$

the structure up to this point is a *strong monad*.

Finally, we also have

– operations $zero_{\mathsf{P}} : \mathsf{P}A \leftarrow B$ and $union_{\mathsf{P}} : \mathsf{P}A \leftarrow \mathsf{P}A \times \mathsf{P}A$

satisfying the following properties:

$$union_{\mathsf{P}} \circ (union_{\mathsf{P}} \times id) = union_{\mathsf{P}} \circ (id \times union_{\mathsf{P}})$$

$$zero_{\mathsf{P}} \circ ! = zero_{\mathsf{P}}$$

$$union_{\mathsf{P}} \circ (zero_{\mathsf{P}} \vartriangle id) = id$$

$$union_{\mathsf{P}} \circ (id \vartriangle zero_{\mathsf{P}}) = id$$

$$map_{\mathsf{P}}.f \circ zero_{\mathsf{P}} = zero_{\mathsf{P}}$$

$$flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.zero_{\mathsf{P}} = zero_{\mathsf{P}}$$

$$flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.f \circ union_{\mathsf{P}} = union_{\mathsf{P}} \circ ((flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.f) \times (flatten_{\mathsf{P}} \circ map_{\mathsf{P}}.f))$$

making the entire structure a *strong ringad* [24].

This equals the calculus $\mathcal{M}_{\cup}(=, cond)$ of Tannen et al [6, 7], whose expressive power equals that of the nested relational algebra with equality test [20], see [6, 7] for a proof. An important property of this calculus is that the complexity of a query is exponential in the size of its input. The nesting and unnesting operations, 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:

$$unnest_{\mathsf{P}, A, B} = flatten_{\mathsf{P}, A \times B} \circ map_{\mathsf{P}}.stl_{\mathsf{P}, A, B}$$

and the beast (translated from [22]):

$$nest_{\mathsf{P},A,B} = map_{\mathsf{P}}.(\ll_{A,\mathsf{P}(A\times B)} \vartriangle f) \circ str_{\mathsf{P},A,\mathsf{P}(A\times B)} \circ (\mathit{\Pi}_{A,B} \vartriangle id_{\mathsf{P}(A\times B)}) \quad \text{where}$$

$$f = \Pi'_{A\times A, B} \circ lift_{\mathsf{P}}.(equal_A \times id_B) \circ stl_{\mathsf{P},A,A\times B}$$

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. An informal corollary of this is the conclusion that "first normal form" is a rather artificial restriction (in this setup).

Moreover, we have not needed any other properties of the operations used, besides those listed above. Thus, a monadic calculus of this form can be defined for any strong ringad. Important examples of these are *lists* and *bags* with their obvious (certainly for Boom hierarchy adepts) operations. For this reason, Trinder [23] and Tannen et al [6, 22] have argued that strong ringads describe the essence of so-called *bulk-* or *collection types* in databases.

As an aside, not all operations defined above generalise 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:

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

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).

5 Multiple monads: research issues

So far, we have only shown that we can define the same calculus for each monad separately. However, ideally one should have several of these collection types available in one language, and be able to write mixed expressions. From simply combining the set and list versions of the calculus, we already get expressions like $map_{\mathsf{P}}.map_{\mathsf{L}}.f : \mathsf{PL}B \leftarrow \mathsf{PL}A$ but there is no option yet to move from one datatype to another in general. To our knowledge, this has not been an issue for research in the database programming languages community.

Can general conversion functions between arbitrary strong ringads exist? Unfortunately, the answer to that question is "no". For example, to define a deterministic conversion from bags to lists one needs an order on the element type, which need not exist. On the other hand, 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 – [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. However, more polymorphic transformations are known to exist, 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 (extending the Boom hierarchy), with polymorphic data type transformations as arrows and possibly the powerset type P as a terminal object with $setify_{\mathsf{F}}$ as its unique arrow.

A completely different class of useful operations for interfacing several data types is formed by operations that commute functors, i.e. that convert FG-structures into GF-structures. Our group has studied such operations [3], 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 [10] 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.[5] (In general $cross_{\mathsf{F}}$ is not related

[5]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.

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.[6]

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 [17, 16]. 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 expressions). 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 formal program development on zips [3], the Boom hierarchy [19, 14] of datatypes, and polytypic programming in general [4, 11, 15] will provide a basis for this further research.

6 Concluding remarks

We have presented a monadic calculus for querying nested collections, inspired by (and in some sense equivalent to) the ones defined by Tannen, Buneman and Wong [6, 7] and Trinder [23]. Our presentation was designed to connect theories from formal program development with the state of the art in database query languages. Thus, a set up of functors over partial functions with associated natural transformations was chosen. Large portions of equational (point-free) calculi for category theory, for (total) functions [19, 5], and for binary relations [1] are directly applicable to our calculus. Much of these will translate directly to well-known or possibly even new optimisations of (nested) relational database queries. The calculus instantiated to a single monad appears to be complete and well understood, cf. [7]. However, having the monad (functor) as an explicit parameter induces the question of how these calculi could be combined for several monads, leading to a query language for databases involving multiple collection types. Our partial answer to this question showed that this strongly relates to several issues currently studied in the area of formal program development, most importantly to polytypism [4].

We hope that future research will continue the cross-fertilisation of these two areas, with investigation of data types, their operations, and transformations between them taking a central place.

[6]For $zip_{\mathsf{F},\mathsf{G}}$ we have $zip_{\mathsf{F},\mathsf{G}} \circ map_{\mathsf{F}}.stl_{\mathsf{G}} \circ str_{\mathsf{F}} = map_{\mathsf{G}}.str_{\mathsf{F}} \circ stl_{\mathsf{G}}$. This clearly relates the definition of *cross* with the alternative one where “the other index runs faster”.

Acknowledgements

Val Tannen's support in our exploration of this research area is much appreciated. Our colleagues in the Mathematics of Program Construction section in Eindhoven and Simon Thompson at UKC gave many useful comments on this paper and other presentations we have given of this material. We thank Leonid Libkin for his helpful responses to our questions on or-sets.

References

[1] C.J. Aarts, R.C. Backhouse, P. Hoogendijk, T.S. Voermans, and J. van der Woude. A relational theory of datatypes. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, September 1992.

[2] S. Abiteboul, C. Beeri, M. Gyssens, and D. van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*, pages 117–138. Springer Verlag, 1987.

[3] 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`.

[4] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *J. of Functional Programming*, 6(1):1–28, January 1996.

[5] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design. NATO ASI Series Vol. F36*, pages 5–42. Springer-Verlag, Berlin, 1987.

[6] V. Breazu-Tannen, P. Buneman, and L. 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.

[7] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*. To appear.

[8] E.F. Codd. A relational model for large shared databank. *Communications of the ACM*, 13(6):377–387, 1970.

[9] L.S. Colby. A recursive algebra for nested relations. *Information Systems*, 15(5):567–582, 1990.

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

[11] Oege de Moor. A generic program for sequential decision processes. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, *Programming Languages: Implementations, Logics and Programs. 7th International Symposium, PLIPS '95 Utrecht, The Netherlands, September 1995*, volume 982 of *Lecture Notes in Computer Science*, pages 1–23. Springer Verlag, 1995.

[12] H. Doornbos. *Reductivity arguments and program construction.* PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, June 1996.

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

[14] 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.

[15] J. Jeuring. Polytypic pattern matching. In S. Peyton Jones, editor, *Proceedings Functional Programming Languages and Computer Architecture, FPCA '95*, June 1995.

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

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

[18] G.R. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[19] L.G.L.T. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proc. CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334, 1986.

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

[21] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *NATO ARW "Software for Parallel Computation"*, volume 106 of *Series F*. NATO ASI Workshop on Software for Parallel Computation, Cetraro, Italy, June 1992, Springer-Verlag NATO ASI, 1993.

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

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

[24] P. Wadler. Notes on monads and ringads. Unpublished note, 1990.