Principles of Programming with Complex Objects and Collection Types*

Peter Buneman* Shamim Naqvi † Val Tannen * Limsoon Wong ‡

January 95

Abstract

We present a new principle for the development of database query languages that the primitive operations should be organized around types. Viewing a relational database as consisting of sets of records, this principle dictates that we should investigate separately operations for records and sets. There are two immediate advantages of this approach, which is partly inspired by basic ideas from category theory. First, it provides a language for structures in which record and set types may be freely combined: nested relations or complex objects. Second, the fundamental operations for sets are closely related to those for other "collection types" such as bags or lists, and this suggests how database languages may be uniformly extended to these new types.

The most general operation on sets, that of *structural recursion*, is one in which not all programs are well-defined. In looking for limited forms of this operation that always give rise to well-defined operations, we find a number of close connections with existing database languages, notably those developed for complex objects. Moreover, even though the general paradigm of structural recursion is shown to be no more expressive than one of the existing languages for complex objects, it possesses certain properties of uniformity that make it a better candidate for an efficient, practical language. Thus rather than developing query languages by extending, for example, relational calculus, we advocate a very powerful paradigm in which a number of well-known database database languages are to be found as natural sublanguages.

*Invited paper appeared in *Theoretical Computer Science* 149(1):3–48, 1995. Extended abstract appeared in *Proceedings of 4th International Conference on Database Theory.*

*University of Pennsyvania, Department of Computer and Information Science, 200 South 33rd Street, Philadelphia, PA 19104. `Email: {peter, val}@cis.upenn.edu`

†Bellcore, 445 South Street, Morristown, NJ 07960-1910. `Email: shamim@bellcore.com`

‡Institute of Systems Science. Heng Mui Keng Terrace, Singapore 0511. `Email: limsoon@iss.nus.sg`

1 Introduction

Overcoming the limitations of first-order logic as a database query language has always been a major focus of relational database research. The limitations are well-known: first-order logic cannot express certain simple computations on relations [4]; also, by its nature, it cannot directly express queries on structures that are not simple relations, such as nested relations or other useful database types such as bags (multisets) and lists. To overcome these limitations, there have been two general strategies. The first has been to add extra operations such as a fixpoint or *while* [14, 3]. The second has been to consider higher-order logics [1] or restricted algebras for higher-order (nested) relations [51, 57, 16]. However these extensions to first-order logic leave much to be desired; it is not clear how they fit together; they do not address the problems of bags and lists; and it is not clear when this process of extending first-order logic will stop! It should be noted that practical query languages for object-oriented databases require such extensions, as does SQL, with its aggregate operations and use of bags.

There are special problems in writing programs that operate on sets or bags. Many database systems provide an interface to conventional programming languages that allows us to write programs that iterate over some collection:

```
 program SUM: int sum = 0;
              foreach x in S do
                sum = x + sum;

program DIFF: int foo = 0;
              foreach x in S do
                foo = x - foo;
```

in which the collection `S` may typically be a list, bag (multiset) or set. The meaning of `SUM` appears obvious, but what meaning are we to attach to `DIFF` when the collection `S` is a set? The outcome depends on the order in which the set is traversed. An awkward way of dealing with this problem is to assume non-deterministic semantics, meaning that there is a set of outcomes for programs such as `DIFF`. Another possibility [27] is to assume that each set carries an intrinsic ordering that dictates the order in which the iteration progresses. Such an ordering will only be of use if it is known to the programmer, and while there is a natural order to choose for integers, generating an ordering for complex structures is non-trivial and sensitive to otherwise arbitrary choices of database design.

Practical languages that contain general-purpose iterators allow the construction of such ill-defined programs, and ensuring that a program is well defined is left to the programmer. Our approach to this problem is to characterize languages that can iterate over collections and then to look for well-defined fragments of such languages. Database query languages appear to avoid the issue by having a few built-in aggregate functions, such as `SUM`, rather than a general-purpose iterator, however, without an assumption that an iteration will never encounter the same element of a set twice, even the program `SUM` is ill-defined for sets, and that this issue arises even in query languages such as SQL.

We are therefore led to look for basic programming constructs for collections that allow us to reason about the well-definedness of programs, and propose *structural recursion* as the general paradigm, together with certain operations for constructing sets. If we add to these the relatively simple operations on records we obtain, a language for manipulating collections of records, i.e., relations when we restrict collections to sets. In fact we do better, for we obtain a language that will query structures produced by freely combining these types, i.e complex object or nested relational databases. This brings us to the main point of this paper; we find that known languages for nested relations can be cleanly described within this approach. A further benefit is that the same principles provide us with languages for other collection types, though we do not fully develop these languages here.

The process of organizing programming primitives around types is well-known in category theory, and we have found it useful to take some basic ideas from this subject. In particular, we shall present a "calculus" and an equivalent "algebra" of functions for nested relations. The algebra is inspired by a well-understood categorical construction, the *monad* (or triple). The idea that monads could be used to organize semantics of programming constructs is due to Moggi [46]. Wadler [61] showed that they are also useful in organizing syntax, in particular they explain the "list-comprehension" syntax of functional programming. Moreover Trinder and Wadler [59] showed that an extension of comprehensions can implement the (flat) relational calculus. Trinder and Watt [58, 62], have also sought after a uniform algebra for several different bulk types; in particular they have proved a number of optimizations using categorical identities. The technical development in the paper does not require familiarity with category theory; however readers interested in understanding our motivation may wish to refer to the introductory material in texts such as such as [39] or [42].

This paper does not deal with the practical aspects of the design of syntax for query languages, but focusses on the semantics of the constructs that could be used in such a langauge. We comment on some of the problems of syntax in the conclusions to this paper.

Organization

In section 2 we introduce two forms of structural recursion on collections and give conditions for their well-definedness. Because there is no general method of checking that a program satisfies these conditions, we examine a natural restriction of structural recursion that ensures well-definedness. This restriction leads immediately to a core language for nested collections which, in section 3, enables us to develop both a calculus and a functional algebra. We exhibit translations between the two languages that preserve meaning as well as preserving and reflecting their equational theories (see appendices). Hence the two can be freely combined into a single language $\mathcal{M}$, upon which we build our nested relational language.

Although $\mathcal{M}$ can express a number of familiar operations on relations and nested relations, we show, in section 4 that it cannot express empty set and set union. Adding these to $\mathcal{M}$ gives us a stronger language $\mathcal{R}$, but this language cannot express operations such as an equality test, a subset test, a membership test, relational nesting, or set intersection, that are non-monotonic with respect to a certain ordering. We show that the languages obtained by adding any one of these operations to $\mathcal{R}$ are equally expressive. These languages have polynomial time complexity. A similar but weaker result was obtained in [21] by assuming the presence of a powerset operation. We then show that $\mathcal{R}$ augmented with equality testing is equivalent to the well-known nested relational algebra of Thomas and Fischer [57]. By [49] it follows that our nested relational language is conservative with respect to flat relational algebra. That is, the queries with flat relations as input and flat relations as output are expressible in (flat) relational algebra. Because, both flat and nested relational algebra are now seen as natural fragments of a general programming paradigm, we are in a position to extend them to other collection types, though we do not do this here; see [33, 37, 35].

In section 5 we further augment the language with a powerset operation $\mathcal{R}(=, \mathsf{cond})$, to obtain the algebra of Abiteboul and Beeri [1]. In view of conservativity over relational algebra, this algebra cannot express functions such as transitive closure and parity test without a potentially expensive excursion through an powerset type. Furthermore, we show that it cannot *uniformly* compute the cardinality of a set no matter what extra arithmetic primitives are added. The power of unrestricted structural recursion is also considered in section 5. We show that it can compute powerset and hence is at least as powerful as the language of Abiteboul and Beeri. More importantly prove that efficient uniform algorithms for transitive closure, cardinality, etc. can be expressed using structural recursion (with simple arithmetic primitives.) It is not clear that such efficiency can be obtained in the Abiteboul and Beeri algebra. Lastly, we also show that under certain conditions the language of Abiteboul and Beeri can simulate structural recursion.

In section 6 we show how the axioms of a monad can be used to derive and generalize well-known optimizations for relational languages, and we also show how the categorical notion of naturality provides some very general equational techniques. We conclude by mentioning some recent practical developments from this work.

2 Structural recursion on collection types

The definitions by structural recursion that we consider follow from mathematical characterizations of certain algebras of operations on collection types. They are closely related to the familiar definitions of functions by *simple recursion* on natural numbers, for example:

$$\begin{array}{lcl} \mathit{double}(0) & = & 0 \\ \mathit{double}(n+1) & = & \mathit{double}(n) + 2 \end{array}$$

The fact that there is a function *double* satisfying these two equations, and moreover that such a function is unique, follows from the "universality" property enjoyed by the natural numbers $\mathbb{N}$ together with 0 and the successor operation $s(n) := n + 1$. Indeed the algebra $(\mathbb{N}, 0, s)$ is *initial* among similar algebras, that is, there exists a unique homomorphism from it to any such algebra. In this case, *double* is the homomorphism to $(\mathbb{N}, 0, d)$ where $d(m) := m + 2$, and the two equations above state precisely that *double* is a homomorphism. An important remark (and a necessary condition for initiality) is that any natural number can be obtained by finitely many applications of 0 and s, hence we call these operations *constructors* for the data type of natural numbers. Functions defined by simple recursion are homomorphisms with respect to these constructors.

Structural recursion is the concept that generalizes simple recursion to any data type that can be defined by a similar algebraic universality property[19]. Consequently, we devote the next subsection (2.1) to exhibiting two groups of constructors for each of the collection types that interest us. This is followed in subsection 2.2 by the presentation of two forms of structural recursion that correspond to these two groups of constructors, and their applicability conditions.

2.1 Three collection types and their constructors

The collection types of interest to us are

$$\begin{array}{rcl}\{\sigma\} & := & \text{the type of all finite sets of elements of type } \sigma, \\ \{\!|\sigma|\!\} & := & \text{the type of all finite bags of elements of type } \sigma, \\ [\sigma] & := & \text{the type of all finite lists of elements of type } \sigma.\end{array}$$

When we wish to refer generically to any of these types we will use the common notation

$$\mathsf{coll}(\sigma) \quad := \quad \text{the type of all finite collections of elements of type } \sigma.$$

The other two type constructions that we shall make use of are:

$$\begin{array}{rcl}\sigma \times \tau & := & \text{the type of all pairs } (x,y) \text{ where } x \text{ is of type } \sigma \text{ and } y \text{ is of type } \tau, \\ \sigma \to \tau & := & \text{the type of all functions with argument of type } \sigma \text{ and result of type } \tau.\end{array}$$

Which operations play the role of constructors for collection data types? We observe that there appear to be two principal ways of constructing a collection. For sets, we can obtain any finite set from the empty set $\{\}$ by finitely many insertions (notation: $\upharpoonleft$). We may alternatively start with the singleton set constructor $\{\cdot\}$ and perform finitely many unions (notation: $\cup$), adding the empty set as a special operation. There are analogous constructors for lists and bags. They are all summarized in the following table, which also includes a common notation that will allow us to give definitions for all three types simultaneously.

	Empty	Addition	Singleton	Combination
Lists	$[\,]$ *nil*	$x :: L$ *cons*	$[x]$	$L_1 @ L_2$ *append*
Bags	$\{\!|\,|\!\}$	$x \leftarrow\!\!\!+ B$ *increment*	$\{\!|x|\!\}$	$B_1 \uplus B_2$ *sum*
Sets	$\{\}$	$x \upharpoonleft S$ *insert*	$\{x\}$	$S_1 \cup S_2$ *union*
Common	empty	$\mathsf{add}(x, C)$	$\mathsf{sng}(x)$	$\mathsf{comb}(C_1, C_2)$

The list operations should be familiar. $x \leftarrow\!\!\!+ B$ is the bag operation that increments by 1 the number of occurrences of x in the bag B, while $\uplus$ sums the number of occurrences of each element.

We have therefore two groups of constructors for each of the three collection types: empty and addition form one group, while empty, singleton and combination form the other. By analogy with simple recursion on natural numbers, we will look in the next subsection at the universality properties enjoyed by the algebras $(\mathsf{coll}(\sigma), \mathsf{add}(\cdot,\cdot), \mathsf{empty})$ and $(\mathsf{coll}(\sigma), \mathsf{comb}(\cdot,\cdot), \mathsf{sng}(\cdot), \mathsf{empty})$.

2.2 Two forms of structural recursion for each collection type

The case of lists with the constructors *nil* and *cons* is an immediate generalization of the natural numbers situation. $\mathbb{N}$ is isomorphic to lists containing some fixed element c; in this case *nil* is zero and *cons* of c is the successor function. Here too we have an initial algebra, and this yields functions defined by structural recursion with respect to the constructors such as the following one:

$$\begin{array}{lcl} sum([\,]) & = & 0 \\ sum(x :: L) & = & x + sum(L) \end{array}$$

sum is the unique homomorphism between the list algebra $([\mathbb{N}], ::, [\,])$ and the algebra $(\mathbb{N}, i, 0)$ where $i(x, n) := x + n$.

As mentioned above, we have two kinds of algebraic structures on each of the three collection types: $(\mathsf{coll}(\sigma), \mathsf{add}(\cdot,\cdot), \mathsf{empty})$ and $(\mathsf{coll}(\sigma), \mathsf{comb}(\cdot,\cdot), \mathsf{sng}(\cdot), \mathsf{empty})$. [1] Each of these algebras is initial among an appropriate class of similar algebras. This gives two forms of definition by structural recursion, one for each kind of algebraic structure. The first form is

$$\begin{array}{lcl} g(\mathsf{empty}) & = & e \\ g(\mathsf{add}(x, C)) & = & i(x, g(C)) \end{array} \qquad \text{Typing: } \frac{e : \tau \qquad i : \sigma \times \tau \to \tau}{g : \mathsf{coll}(\sigma) \to \tau}$$

In this, the function g depends on i and e, so we shall use the notation $g = \mathsf{sr\_add}(i, e)$. ($\mathsf{sr\_add}$ for structural recursion on "insertion"). We shall also use the notations $\mathsf{sr\_add}_{list}$, $\mathsf{sr\_add}_{bag}$, $\mathsf{sr\_add}_{set}$ for each of the individual collection types. For lists, $\mathsf{sr\_add}$ is the familiar "fold" or "reduce" operation of functional programming languages.

The second form of structural recursion is

$$\begin{array}{lcl} h(\mathsf{empty}) & = & e \\ h(\mathsf{sng}(x)) & = & f(x) \\ h(\mathsf{comb}(C_1, C_2)) & = & u(h(C_1), h(C_2)) \end{array} \qquad \text{Typing: } \frac{e : \tau \qquad f : \sigma \to \tau \qquad u : \tau \times \tau \to \tau}{h : \mathsf{coll}(\sigma) \to \tau}$$

Here, h depends on u, e and f. We shall use the notation $h = \mathsf{sr\_comb}(u, f, e)$ for structural recursion on "union". As above, we may use $\mathsf{sr\_comb}_{list}$, $\mathsf{sr\_comb}_{bag}$, $\mathsf{sr\_comb}_{set}$ respectively for each of the individual collection types we consider.

[1] Fixing an arbitrary σ, we will consider these as homogeneous (one-sorted) algebras over infinite signatures: for each x we have a unary operation $\mathsf{add}(x, \cdot)$ and a nullary operation $\mathsf{sng}(x)$.

Some examples of these forms of structural recursion:

$$
\begin{array}{rcll}
sum(\{\!|\,|\!\}) & = & 0 & sum : \{\!|\sigma|\!\} \to \mathbb{N} \\
sum(x \nrightarrow B) & = & x + sum(B) & \\
\\
count_{bag}(\{\!|\,|\!\}) & = & 0 & count_{bag} : \{\!|\sigma|\!\} \to \mathbb{N} \\
count_{bag}(\{\!|x|\!\}) & = & 1 & \\
count_{bag}(B_1 \uplus B_2) & = & count_{bag}(B_1) + count_{bag}(B_2) & \\
\\
reverse([\,]) & = & [\,] & reverse : [\sigma] \to [\sigma] \\
reverse([x]) & = & [x] & \\
reverse(L_1 @ L_2) & = & reverse(L_2) @ reverse(L_1) & \\
\\
max(\{\}) & = & 0 & max : \{\mathbb{N}\} \to \mathbb{N} \\
max(x \upharpoonleft S) & = & max2(x, max(S)) &
\end{array}
$$

Alternatively, we could have written $count_{bag}$ as $\mathsf{sr\_add}_{bag}(i, 0)$ where $i(x,n) := 1 + n$, and max as $\mathsf{sr\_comb}_{set}(max2, \mathsf{id}, 0)$ where $\mathsf{id}(x) := x$.

Well-definedness conditions As it happens, the functions shown above are all well-defined on the stated types. Note however that the equations in the definitions by structural recursion only state that the desired functions are homomorphisms. They do not state that the algebras which are the targets of these functions belong to the class for which the collection type algebras are initial—a necessary condition for the existence of the desired functions. Indeed, a naive analog for sets of the definition of $count_{bag}$ will not work:

$$
\begin{array}{rcl}
badcount_{set}(\{x\}) & = & 1 \\
badcount_{set}(S_1 \cup S_2) & = & badcount_{set}(S_1) + badcount_{set}(S_2)
\end{array}
$$

And this doesn't work *not* because $badcount_{set}$ is some erroneous function that counts twice the elements that are in both S_1 and S_2. Rather, this doesn't work because there exists no mathematical function $badcount_{set}$ satisfying the two equations above. Indeed, if it existed, then:

$$1 = badcount_{set}(\{a\}) = badcount_{set}(\{a\} \cup \{a\}) = badcount_{set}(\{a\}) + badcount_{set}(\{a\}) = 1 + 1 = 2$$

It is therefore essential to note that the algebra of bags with the summation, singleton, and empty bag operations is initial only among algebras of the form (τ, u, f, e) such that (τ, u, e) is a commutative monoid, that for the algebra of lists with append, singleton, and nil we require just monoid structures, and that for the algebra of sets with union, singleton, and empty set we need commutative-idempotent monoids (equivalently, upper semilattices with least element). Similarly, the algebra of sets with the insertion and empty set operations is initial among similar algebras with a left-commutative and left-idempotent operation, while the for the algebra of bags with increment we require only left-commutativity.

The table below summarizes these well-definedness conditions for the structural recursion constructs. Note that the conditions are downwards cumulative. Thus, what is well-defined on sets is also well-defined on bags and lists, and what is well-defined on bags is also well-defined on lists. [2]

	$\mathsf{sr\_add}(i,e)$	$\mathsf{sr\_comb}(u,f,e)$
Lists	no condition	u associative: $u(u(a_1,a_2),a_3)=u(a_1,u(a_2,a_3))$ e identity for u: $u(a,e)=a=u(e,a)$
Bags	i left-commutative: $i(x,i(y,a))=i(y,i(x,a))$	also u commutative: $u(a_1,a_2)=u(a_2,a_1)$
Sets	also i left-idempotent: $i(x,i(x,a))=i(x,a)$	also u idempotent: $u(a,a)=a$

More examples First we observe that $\mathsf{sr\_comb}(u,f,e) = \mathsf{sr\_add}(i,e)$ where $i(x,z) := u(f(x),z)$. Hence, everything expressible with $\mathsf{sr\_comb}$ is immediately expressible with $\mathsf{sr\_add}$. [3]

A number of powerful functions can be obtained by structural recursion. For example a general form of mapping is given as follows. If $f:\sigma\to\tau$ is an arbitrary function then $\mathsf{map}(f)$ is defined by

$$\begin{array}{rcl} \mathsf{map}(f)(\mathsf{empty}) & = & \mathsf{empty} \\ \mathsf{map}(f)(\mathsf{add}(x,C)) & = & \mathsf{add}(f(x),\mathsf{map}(f)(C)) \end{array} \qquad \mathsf{map}(f):\mathsf{coll}(\sigma)\to\mathsf{coll}(\tau)$$

On sets, for example, the meaning of $\mathsf{map}(\cdot)$ is $\mathsf{map}(f)(\{a_1,\ldots,a_n\}) := \{f(a_1),\ldots,f(a_n)\}$. It is straightforward to show that $\mathsf{map}()$ is well defined for any collection type.

Carrying this further, we can define a "power" operator

$$\begin{array}{rcl} power(\mathsf{empty}) & = & \mathsf{sng}(\mathsf{empty}) \\ power(\mathsf{add}(y,C)) & = & \mathsf{comb}(power(C),\mathsf{map}(h)(power(C))) \end{array} \qquad power:\mathsf{coll}(\sigma)\to\mathsf{coll}(\mathsf{coll}(\sigma))$$

where $h(c) := \mathsf{add}(y,c)$

This gives a generalization of powerset to all collection types and, again, it can be shown to be well-defined.

Another example is

$$\begin{array}{rcl} BagToSet(\{\!|\,|\!\}) & = & \{\} \\ BagToSet(x \mathbin{+\!\!\!\!\to} B) & = & x \upharpoonleft BagToSet(B) \end{array} \qquad BagToSet:\{\!|\sigma|\!\}\to\{\sigma\}$$

$BagToSet$ can also be obtained as $\mathsf{sr\_comb}_{bag}(\cup,\{\cdot\},\{\})$. Similarly we can easily write $ListToBag$ and $ListToSet$, both with $\mathsf{sr\_add}$ and with $\mathsf{sr\_comb}$.

Section 5.2 contains further examples of functions definable by structural recursion.

[2]In fact, if we consider only $\mathsf{sr\_comb}$ and what can be from it, then our definitions and results further generalize to binary trees (by dropping the associativity requirement) hence to all the levels of Boom's hierarchy of types [43].

[3]Using some higher-order programming [8], or some exponential computations [56], one can also define $\mathsf{sr\_add}$ in terms of $\mathsf{sr\_comb}$.

Some identities Because initiality postulates *unique* homomorphisms and because the composition of two homomorphisms is also a homomorphism it follows immediately that if i is left-commutative then

$$\mathsf{sr\_add}_{list}(i,e) = \mathsf{sr\_add}_{bag}(i,e) \circ \mathit{ListToBag}$$

and in particular

$$\mathit{ListToSet} = \mathit{BagToSet} \circ \mathit{ListToBag}.$$

It also follows that if i is left-commutative and left-idempotent then

$$\mathsf{sr\_add}_{bag}(i,e) = \mathsf{sr\_add}_{set}(i,e) \circ \mathit{BagToSet}.$$

Analogous identities for $\mathsf{sr\_comb}$ are easily seen to hold.

The bad news Unfortunately, even for simple programming languages featuring structural recursion (together with a few basic constructs such as the ability to manipulate pairs), asking whether equational conditions like associativity, left-idempotence, commutativity, etc., are true about program phrases is *undecidable*, in fact, not even recursively axiomatizable [8, 53].

A programming languages based on full-fledged structural recursion on collections is therefore not an r.e. language—not an easy sell! As we will see however, in database programming this is not necessarily a serious inconvenience, as most programming can be done with restrictions that are always well-defined.

2.3 A restriction of structural recursion

In view of the fact that checking the well-definedness of structural recursion is not decidable, we consider the following limited form that is *always* well-defined, for each of sets, bags, and lists.

$$\begin{array}{lcl} h(\mathsf{empty}) & = & \mathsf{empty} \\ h(\mathsf{sng}(x)) & = & f(x) \\ h(\mathsf{comb}(C_1, C_2)) & = & \mathsf{comb}(h(C_1), h(C_2)) \end{array} \qquad \text{Typing:} \quad \frac{f : \sigma \to \mathsf{coll}(\tau)}{\mathsf{ext}(f) : \mathsf{coll}(\sigma) \to \mathsf{coll}(\tau)}$$

Notation: $h = \mathsf{ext}(f)$.

We will also use the specializations $\mathsf{ext}_{set}(f)$, $\mathsf{ext}_{bag}(f)$, $\mathsf{ext}_{list}(f)$. The meaning of $\mathsf{ext}(f)(S)$ is to apply f to each member of S and then to "flatten" the resulting collection of collections. That is,

$$\begin{array}{rcl} \mathsf{ext}_{set}(f)(\{x_1,\ldots,x_n\}) & = & f(x_1) \cup \cdots \cup f(x_n) \\ \mathsf{ext}_{bag}(f)(\{\!|x_1,\ldots,x_n|\!\}) & = & f(x_1) \uplus \cdots \uplus f(x_n) \\ \mathsf{ext}_{list}(f)([x_1,\ldots,x_n]) & = & f(x_1) @ \cdots @ f(x_n) \end{array}$$

Expressiveness of $\mathsf{ext}(\cdot)$ The following examples of the expressive power of $\mathsf{ext}(\cdot)$ suggest its interest for database programming.

- $\mathsf{map}(f) := \mathsf{ext}(g)$ where $g(x) := \mathsf{sng}(f(x))$.
- $\mathsf{flatten} : \mathsf{coll}(\mathsf{coll}(\sigma)) \to \mathsf{coll}(\sigma)$ which "flattens" a collection of collections by combining its members is given by $\mathsf{flatten} := \mathsf{ext}(\mathsf{id})$.
- $\Pi_1 : \mathsf{coll}(\sigma \times \tau) \to \mathsf{coll}(\sigma)$ which projects the first component is given by $\Pi_1 := \mathsf{map}(f)$ where $f(x, y) := x$. Similarly $\Pi_2 : \mathsf{coll}(\sigma \times \tau) \to \mathsf{coll}(\tau)$. For sets, these are the relational projections.
- The function $\mathsf{pairwith}_2 : \sigma \times \mathsf{coll}(\tau) \to \mathsf{coll}(\sigma \times \tau)$ that pairs an element with each element of a collection is given by $\mathsf{pairwith}_2(e, C) := \mathsf{map}(f)(C)$ where $f(y) := (e, y)$. Similarly, pairing on the right $\mathsf{pairwith}_1 : \mathsf{coll}(\sigma) \times \tau \to \mathsf{coll}(\sigma \times \tau)$.
- Depending on which argument we traverse first, we have two generalizations of the cartesian product, both of type $\mathsf{coll}(\sigma) \times \mathsf{coll}(\tau) \to \mathsf{coll}(\sigma \times \tau)$. The first one is $\mathsf{cartprod}_1(S_1, S_2) := \mathsf{ext}(f)(S_1)$ where $f(x) := \mathsf{pairwith}_2(x, S_2)$ while the second one is $\mathsf{cartprod}_2(S_1, S_2) := \mathsf{ext}(g)(S_2)$ where $g(y) := \mathsf{pairwith}_1(S_1, y)$. For bags and sets the two coincide. For sets this is the usual cartesian product.
- The function $\mathsf{unnest}_2 : \mathsf{coll}(\sigma \times \mathsf{coll}(\tau)) \to \mathsf{coll}(\sigma \times \tau)$ that unnests a nested collection is given by $\mathsf{unnest}_2 := \mathsf{ext}(\mathsf{pairwith}_2)$.

We see therefore that this simple and always well-defined instance of structural recursion can already express important operations which, when interpreted for sets, are operations of relational or nested relational algebra.

Monads $\mathsf{ext}(\cdot)$ is interesting not only because it is expressive, but also because it is an instance of a mathematically ubiquitous and hence well-studied concept, that of *monad* (see [39] or [42] where monads are called algebraic theories). An introduction to category theory and to monads is beyond the scope of this paper. While the main body of the paper makes several references to monads, notably in section 3, these references are mainly about terminology. For the reader interested in more information about monads, we present in appendix A the category-theoretic version of the monad axioms, preceded by a short discussion of the relationship between structural recursion and monads.

3 A core language built around monad constructs

We now want to develop a "core" language starting from the the expressive power of $\mathsf{ext}(\cdot)$. Only sets are treated in what follows. Bags and lists can be treated similarly. We shall take *complex object types* to be those types that can be constructed using the set and product onstructors. These types are given by:

$$\tau ::= b \mid unit \mid \tau \times \tau \mid \{\tau\}$$

where b ranges over base types. For example, b can be instantiated by a basic data type such as integers, strings, etc. The type *unit* contains just one element $()$. This can be taken as the type of "0-ary" tuples (there can only be one such tuple), The product and set types have already been described in subsection 2.1.

In this section we present two equivalent formulations for such a monad-based core language for complex objects. The first one (subsection 3.1) is calculus-like because it makes heavy use of bound variables. The second one (subsection 3.2)is algebraic since it is variable-free. They are equivalent in the sense that they describe the same class of functions that map complex objects to complex objects (subsection 3.3). In appendix B we give a deeper equivalence result, one that relates the equational theories of the two formulations.

3.1 A core calculus of complex objects

We assume given an infinite collection of variables, and, for simplicity, each is assigned once and forever a complex object type, x^σ. Hence variables can range only over complex objects — an important restriction that precludes variables being bound to functions. expressions and their types are given by the rules below. Within these rules, e, e_1, e_2 range over expressions, x over variables, and σ, τ over complex object types.

In order to make the additional point that function definitions can be avoided in the core language and that everything can be done only with expressions that denote data objects, we shall use the syntax

$$\Phi x^\sigma \in S.\, T$$

to denote

$$\mathsf{ext}(f)(S) \quad \text{where} \quad f(x:\sigma) = T$$

In lambda notation, $\Phi x^\sigma \in S.\, T := \mathsf{ext}(\lambda x^\sigma .\, T)(S)$.

The typing of $\Phi x^\sigma \in S.\, T$ is

$$\frac{T : \{\tau\} \qquad S : \{\sigma\}}{\Phi x^\sigma \in S.\, T : \{\tau\}}$$

The expression $\Phi x^\sigma \in S.\, T$ is a little special, because x is a *bound variable* here, similar to variables bound by lambda abstractions; and the purpose of introducing this special notation is specifically to avoid introducing into our language the more general construct of lambda abstraction. The scope of x^σ is the subexpression T. Note that S is *not* part of the scope. This is easier to see if we recall that $\Phi x^\sigma \in S.\, T$ was suggested as an alternative notation for $\mathsf{ext}(\lambda x^\sigma . T)(S)$. As is customary, we identify those expressions that differ only in the name of the bound variables, and we adopt the bound variable convention [6] which says that in any given mathematical context, we can assume that all the bound variables are distinct among themselves and distinct from the free variables occurring in that context.

In addition to Φ we add to our language the expected operations associated with products, with the type *unit* and singleton set formation.

Variables: $\quad \dfrac{}{x^\sigma : \sigma}$

Products: $\dfrac{e_1 : \sigma \qquad e_2 : \tau}{(e_1, e_2) : \sigma \times \tau}$ $\qquad \dfrac{e : \sigma \times \tau}{\pi_1(e) : \sigma \qquad \pi_2(e) : \tau}$ $\qquad \dfrac{}{() : \mathit{unit}}$

Sets: $\dfrac{e : \tau}{\{e\} : \{\tau\}}$ $\qquad \dfrac{S : \{\sigma\} \qquad T : \{\tau\}}{\Phi x^\sigma \in S.\, T \ : \ \{\tau\}}$

Note that each well-typed expression e has a unique type, which we sometimes denote $\mathit{type}(e)$.

Note that many operations on sets are absent, notably emptyset and union. We made this choice in order to present a language that corresponds closely to the monad operations, and thus obtain an exact correspondence with an equivalent formulation based on category-theoretic operations on functions. (see subsection 3.2 theorem 3.1). Emptyset, union, and other operations can be added to either of the two formalisms (this calculus of complex objects or the equivalent algebra of functions presented next), and we consider such extensions in section 4.

While the informal meaning of these expressions is quite clear, the theorems that follow will benefit from a (concise) formal definition of the standard meaning. Since expressions have free variables, we introduce valuations, or *environments*, which are type-preserving functions ρ from variables to complex objects. Since ρ assigns a value to each free variable of e, we can define the meaning of expression e in the environment ρ, denoted $[\![e]\!]\rho$, as follows

$[\![x^\sigma]\!]\rho := \rho(x^\sigma)$

$[\![(e_1, e_2)]\!]\rho := ([\![e_1]\!]\rho, [\![e_2]\!]\rho)$

$[\![\pi_1(e)]\!]\rho := a$ and $[\![\pi_2(e)]\!]\rho := b$ where $[\![e]\!]\rho = (a, b)$

$[\![()]\!]\rho :=$ the unique element of *unit* (can also be viewed as the "tuple with 0 components" since *unit* can be viewed as a 0-ary cartesian product)

$[\![\{e\}]\!]\rho := \{[\![e]\!]\rho\}$

$[\![\Phi x^\sigma \in S.\, T]\!]\rho := [\![T]\!]\rho[a_1/x^\sigma] \cup \cdots \cup [\![T]\!]\rho[a_n/x^\sigma]$ where $[\![S]\!]\rho = \{a_1, \ldots, a_n\}$ and where

$$\rho[b/y](z) := \begin{cases} b & \text{if } z = y \\ \rho(z) & \text{otherwise} \end{cases}$$

If $[\![S]\!]\rho = \emptyset$ then $[\![\Phi x^\sigma \in S.\, T]\!]\rho := \emptyset$.

It is easy to see that $[\![e]\!]\rho$ depends only on the values that ρ takes on the free variables of e. Hence, the meaning of expressions with free variables can be also understood as a function. For example, let $e : \tau$ be an expression with free variables $x_1^{\sigma_1}$ and $x_2^{\sigma_2}$. We can associate with it the function $\sigma_1 \times \sigma_2 \to \tau$ which takes (a_1, a_2) to $[\![e]\!]\rho[a_1/x_1][a_2/x_2]$. Remarkably, there is an elegant alternative description of the functions that are thus definable by complex-object expressions in the core calculus, which will be presented in the next subsection.

3.2 An equivalent (variable-free) core algebra of functions

Here we present an algebraic (no variables!) alternative to the complex object calculus introduced in subsection 3.1. There is an important distinction however: while the operations of the calculus manipulate complex objects, the operations of the algebra manipulate *functions* (from complex objects to complex objects). It is natural to look to category theory for inspiration with such operations, and we borrow general category-theoretic terminology and notation, such as terminators and functorial strength, for our algebra. Nonetheless, we are only talking about sets and set-theoretic functions here, so the meaning of the operations can be easily explained in elementary terms, and we do so right away (the interested reader can consult appendix A for the category-theoretic axiomatization of these operations).

The algebra is given as a many-sorted language. As usual in the language of category theory, the sorts have the form $\sigma \to \tau$ where σ denotes a *source* or *domain* object and τ denotes a *target* or *codomain* object. In our case σ and τ are complex object types. The operations of our algebraic presentation are the following:

Category: $\dfrac{f : \sigma \to \tau \qquad g : \tau \to \upsilon}{g \circ f : \sigma \to \upsilon}$ $\qquad \dfrac{}{\mathsf{id}_\sigma : \sigma \to \sigma}$

Binary products: $\dfrac{f_1 : \sigma \to \tau_1 \qquad f_2 : \sigma \to \tau_2}{\langle f_1, f_2 \rangle : \sigma \to \tau_1 \times \tau_2}$ $\qquad \dfrac{}{\mathsf{fst}_{\sigma,\tau} : \sigma \times \tau \to \sigma}$ $\qquad \dfrac{}{\mathsf{snd}_{\sigma,\tau} : \sigma \times \tau \to \tau}$

Terminator: $\dfrac{}{\mathsf{ter}_\tau : \tau \to \mathit{unit}}$

Monad: $\dfrac{}{\mathsf{sng}_\tau : \tau \to \{\tau\}}$ $\qquad \dfrac{f : \sigma \to \tau}{\mathsf{map}(f) : \{\sigma\} \to \{\tau\}}$ $\qquad \dfrac{}{\mathsf{flatten}_\tau : \{\{\tau\}\} \to \{\tau\}}$

Functorial strength: $\dfrac{}{\mathsf{pairwith}_{2\ \sigma,\tau} : \sigma \times \{\tau\} \to \{\sigma \times \tau\}}$

We omit type subscripts whenever there is no possibility for confusion. The standard model for this presentation is that of functions over complex objects. The meaning of the operations in this model is the following: $\circ$ is function composition, id are the identity functions, $\langle f_1, f_2 \rangle(a) := (f_1(a), f_2(a))$, fst and snd are the first and second projection functions, ter maps everything to the unique element of *unit*, sng produces singleton sets from single elements, $\mathsf{map}(f)(\{a_1, \ldots, a_n\}) := \{f(a_1), \ldots, f(a_n)\}$, $\mathsf{flatten}(\{S_1, \ldots, S_m\}) := S_1 \cup \cdots \cup S_m$, and $\mathsf{pairwith}_2(b, \{a_1, \ldots, a_k\}) := \{(b, a_1), \ldots, (b, a_k)\}$.

A useful abbreviation when dealing with products is the following. For $f_1 : \sigma_1 \to \tau_1,\ f_2 : \sigma_2 \to \tau_2$, write

$$f_1 \times f_2 := \langle f_1 \circ \mathsf{fst}, f_2 \circ \mathsf{snd} \rangle : \sigma_1 \times \sigma_2 \to \tau_1 \times \tau_2.$$

The choice of sng, $\mathsf{map}(\cdot)$ and $\mathsf{flatten}$ to describe monads is only one of several [42]. For example, monads can be equivalently described with sng and $\mathsf{ext}(\cdot)$ in view of the identities:

$$\begin{aligned} \mathsf{map}(f) &= \mathsf{ext}(\mathsf{sng} \circ f) \\ \mathsf{flatten}_\tau &= \mathsf{ext}(\mathsf{id}_{\{\tau\}}) \\ \mathsf{ext}(f) &= \mathsf{flatten} \circ \mathsf{map}(f) \end{aligned}$$

As examples of functions definable in this algebra consider:

- $\Pi_1 := \mathsf{map}(\mathsf{fst})$ and $\Pi_2 := \mathsf{map}(\mathsf{snd})$ are relational projections on sets of pairs.
- $\mathsf{pairwith}_1 := \mathsf{map}(\langle \mathsf{snd}, \mathsf{fst} \rangle) \circ \mathsf{pairwith}_2 \circ \langle \mathsf{snd}, \mathsf{fst} \rangle$. $\mathsf{pairwith}_1$ is like $\mathsf{pairwith}_2$, but pairs to the left.
- $\mathsf{cartprod} := \mathsf{flatten} \circ \mathsf{map}(\mathsf{pairwith}_1) \circ \mathsf{pairwith}_2$
- $\mathsf{unnest}_2 := \mathsf{flatten} \circ \mathsf{map}(\mathsf{pairwith}_2)$.

It is interesting to note that FQL [11], a language designed for the pragmatic purpose of communicating with functional/network databases was based roughly on the same set of functional operators.

3.3 Translations between the calculus and the algebra

As we mentioned before, expressions in the calculus define functions when we consider their free variables as arguments. More generally, we want to consider finite sequence of distinct variables $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ that include the free variables of the expression. We will use Γ to range over such sequences. To deal with closed expressions (without free variables) we also want to consider the empty sequence, $\emptyset$. Γ, x^σ denotes the sequence obtained by adding the variable x^σ at the end of Γ, provided x^σ is not already in Γ. To each such sequence we associate a type which is the cartesian product of the types of the variables in the sequence:

$$\begin{aligned} \mathit{type}(\emptyset) &:= \mathit{unit} \\ \mathit{type}(\Gamma, x^\sigma) &:= \mathit{type}(\Gamma) \times \sigma \end{aligned}$$

We will use the notation $\Gamma \triangleright e$ for a pair consisting of a well-typed calculus expression e and a sequence of variables Γ containing all the free variables in e. To each such pair we associate a function $\mathcal{A}[\Gamma \triangleright e] : \mathit{type}(\Gamma) \to \mathit{type}(e)$ (recall that $\mathit{type}(e)$ is the unique type of e) as follows:

$$\begin{aligned} \mathcal{A}[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \triangleright x_i^{\sigma_i}] &:= \mathsf{proj}_i^n \\ \mathcal{A}[\Gamma \triangleright (e_1, e_2)] &:= \langle \mathcal{A}[\Gamma \triangleright e_1], \mathcal{A}[\Gamma \triangleright e_2] \rangle \\ \mathcal{A}[\Gamma \triangleright \pi_1(e)] &:= \mathsf{fst} \circ \mathcal{A}[\Gamma \triangleright e] \\ \mathcal{A}[\Gamma \triangleright \pi_2(e)] &:= \mathsf{snd} \circ \mathcal{A}[\Gamma \triangleright e] \\ \mathcal{A}[\Gamma \triangleright ()] &:= \mathsf{ter}_{\mathit{type}(\Gamma)} \\ \mathcal{A}[\Gamma \triangleright \{e\}] &:= \mathsf{sng} \circ \mathcal{A}[\Gamma \triangleright e] \\ \mathcal{A}[\Gamma \triangleright \Phi x^\sigma \in S.\, T] &:= \mathsf{flatten} \circ \mathsf{map}(\mathcal{A}[\Gamma, x^\sigma \triangleright T]) \circ \\ &\quad\ \mathsf{pairwith}_2\,{}_{\mathit{type}(\Gamma), \sigma} \circ \langle \mathsf{id}_{\mathit{type}(\Gamma)}, \mathcal{A}[\Gamma \triangleright S] \rangle \end{aligned}$$

Where proj_i^n is defined by $\mathsf{proj}_1^1 = \mathsf{id}$, $\mathsf{proj}_i^n := \mathsf{snd}$ for for $1 < i = n$, and $\mathsf{proj}_i^n := \mathsf{proj}_i^{n-1} \circ \mathsf{fst}$ for $1 \le i < n$.

In the translation of $\Gamma \triangleright \Phi x^\sigma \in S.\, T$, we assume, according to the bound variable convention, that x^σ is not in Γ. Note also that this translation makes essential use of the functorial strength $\mathsf{pairwith}_2$.

To translate from the algebra of functions to the calculus of complex objects, we associate to each function $f : \sigma \to \tau$ a pair $\mathcal{C}[f] = x^\sigma \triangleright e$ where e is a well-typed calculus expression with exactly one free variable, namely x^σ. The translation goes as follows:

$\mathcal{C}[g \circ f] := x^\sigma \triangleright e'[e/y]$ where $\mathcal{C}[f] = x^\sigma \triangleright e$ and $\mathcal{C}[g] = y^\tau \triangleright e'$. Here $e'[e/y]$ is the result of substituting e for y in e'.

$\mathcal{C}[\mathsf{id}_\sigma] := x^\sigma \triangleright x^\sigma$.

$\mathcal{C}[\langle f_1, f_2\rangle] := x^\sigma \triangleright (e_1, e_2)$ where $\mathcal{C}[f_1] = x^\sigma \triangleright e_1$ and $\mathcal{C}[f_2] = x^\sigma \triangleright e_2$.

$\mathcal{C}[\mathsf{fst}_{\sigma,\tau}] := z^{\sigma\times\tau} \triangleright \pi_1(z)$.

$\mathcal{C}[\mathsf{snd}_{\sigma,\tau}] := z^{\sigma\times\tau} \triangleright \pi_2(z)$.

$\mathcal{C}[\mathsf{ter}_\tau] := x^\tau \triangleright ()$.

$\mathcal{C}[\mathsf{sng}_\tau] := x^\tau \triangleright \{x\}$.

$\mathcal{C}[\mathsf{map}(f)] := s^{\{\sigma\}} \triangleright \Phi x^\sigma \in s.\, \{e\}$ where $\mathcal{C}[f] = x^\sigma \triangleright e$.

$\mathcal{C}[\mathsf{flatten}_\tau] := S^{\{\{\tau\}\}} \triangleright \Phi s^{\{\tau\}} \in S.\, s$.

$\mathcal{C}[\mathsf{pairwith}_{2\ \sigma,\tau}] := w^{\sigma\times\{\tau\}} \triangleright \Phi y^\tau \in \pi_2(w).\, \{(\pi_1(w), y)\}$.

Both translations preserve meaning. Formally

Theorem 3.1

1. *For any pair $\Gamma \triangleright e$ in the calculus, denoting $\mathcal{A}[\Gamma \triangleright e]$ by f, we have:*
   - *if $\Gamma = \emptyset$ then $f(u) = [\![e]\!]\rho$ where u is the unique element of type unit and ρ is arbitrary;*
   - *if $\Gamma = x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ then for any $a_1, \ldots, a_n$ we have $f(\ldots(u, a_1), \ldots, a_n) = [\![e]\!]\rho[a_1/x_1]\cdots[a_n/x_n]$ where u is the unique element of type unit and ρ is arbitrary.*
2. *For any function $f : \sigma \to \tau$ in the algebra, denoting $\mathcal{C}[f]$ by $x^\sigma \triangleright e$, we have $[\![e]\!]\rho = f(\rho(x^\sigma))$ for any ρ.*

The proof of this theorem is by straightforward inductions on calculus expressions (part 1) and on algebra expressions (part 2) and is omitted.

This semantic relationship between the calculus and the algebra is sufficient when we concern ourselves with query language expressiveness. Even though the standard model of functions over complex objects is the only one we are considering in this paper, it is worthwhile considering the associated equational theories that may have other possible models. As we explain in section 6 the equational axiomatization of these standard mathematical properties seems to play an important role in validating and discovering optimizations.

For the algebra the equational theory is axiomatized by "commutative diagrams" in standard category theory style (appendix A). For the calculus one can find a corresponding axiomatization, in the style of lambda calculus (appendix B, and see [46]). If we think of the equational theories as semantics, it turns out that a more profound relationship exists between the calculus and the algebra: we show in appendix B that the translations given above "preserve and reflect" these equational theories.

Notation. Since the calculus and the algebra are equivalent, we can speak conceptually of a core language $\mathcal{M}$ whose constructs are those associated with a monad. We can then choose either one of the two formalisms when we need to prove something about this core language.

Since the equivalence between the calculus and the algebra is given by effective translations we can use syntactic sugar that mixes the two formalisms when we wish to show that something is expressible in $\mathcal{M}$. For example, if $f : \sigma \to \tau$ is a function in the algebra, and $e : \sigma$ is an expression in the calculus, we can "apply" f to e, writing $f\,e$ instead of the calculus expression $e'[e/x^\sigma]$ of type τ that is obtained by translating f into the calculus: $x^\sigma \triangleright e' := \mathcal{C}[f]$.

Also, if $e : \tau$ is an expression in the calculus, and $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}, n \geq 1$ is a sequence of distinct variables containing the free variables of e, we can obtain a function in the algebra by "abstract" these variables over e. We would then add the auxiliary function definition

$$\mathit{fname}(x_1, \ldots, x_n) = e$$

and then we would use $\mathit{fname} : \sigma_1 \times \cdots \times \sigma_n \to \tau$ as a function in the algebra, knowing that fname stands for the algebra expression $f \circ (\cdots(\langle \mathsf{ter}, \mathsf{id}\rangle \times \mathsf{id}) \times \cdots \times \mathsf{id})$ that is obtained by translating e into the algebra: $f := \mathcal{A}[x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \triangleright e]$.

In the following sections we will extend $\mathcal{M}$ with other primitives. It is important to keep in mind that the extensions are done differently in the two formalisms. In the complex object calculus we would add an expression construct of the form

$$\frac{e_1 : \mathit{Arg}_1\mathit{type}(\mathsf{C}) \quad \cdots \quad e_n : \mathit{Arg}_n\mathit{type}(\mathsf{C})}{\mathsf{C}(e_1, \ldots, e_n) : \mathit{Type}(\mathsf{C})}$$

while in the algebra of functions we would add a functional constant:

$$\frac{}{c : \mathit{DType}(c) \to \mathit{CType}(c)}$$

The equivalence between the calculus and the algebra would be preserved because typically $\mathcal{A}[\Gamma \triangleright \mathsf{C}(e_1, \ldots, e_n)]$ is straightforwardly expressed in terms of c, and so is $\mathcal{C}[c]$ in terms of C.

Notation. If Σ is a signature of additional primitives, we will denote the extension of $\mathcal{M}$ with these primitives by $\mathcal{M}(\Sigma)$.

4 Nested relational algebra

We now proceed to enrich $\mathcal{M}$ in order to express the operations of the relational algebra and, of particular interest, those of the *nested relational algebra*. There are several equivalent formulations of the nested relational algebra and one of the goals of this section is a rational reconstruction for it. We mention two specific operations of the nested relational algebra $\mathsf{unnest}_2 : \{\sigma \times \{\tau\}\} \to \{\sigma \times \tau\}$ which, as we saw in Section 2.3, is already expressible in $\mathcal{M}$, and $\mathsf{nest}_2 : \{\sigma \times \tau\} \to \{\sigma \times \{\tau\}\}$ which is considered below.

As for the operations that are already in the relational algebra, we have seen in Section 2.3 that relational projections Π_1 and Π_2 and cartesian product $\mathsf{cartprod}$ can be expressed in $\mathcal{M}$. However, union cannot be expressed in $\mathcal{M}$. In this section we extend $\mathcal{M}$ parsimoniously, while staying within polynomial-time computability. We reach a formulation that is equivalent to those given to the nested relational algebra, then we argue for the utility of a conditional and we conclude the section with a discussion of the advantages of our formulation.

4.1 Adding union and the empty set

Theorem 4.1 *Neither binary union nor emptyset can be expressed in $\mathcal{M}$.*

Proof. Assume a domain of complex objects over some base type with at least two distinct elements c_1 and c_2. Consider the sub-domain in which every set — of whatever type — has exactly one member. This sub-domain is closed under the operations of the functional algebra of $\mathcal{M}$ and hence under $\mathcal{M}$ by Theorem B.1. It does not contain empty set; nor is it closed under union because the union of $\{c_1\}$ and $\{c_2\}$ has two members. Hence neither the set union operation nor empty set are definable in $\mathcal{M}$. $\square$

We therefore add these as primitives at all types. In the complex object calculus we have

$$\frac{}{\{\}_\tau : \{\tau\}} \qquad \frac{S_1 : \{\tau\} \quad S_2 : \{\tau\}}{S_1 \cup S_2 : \{\tau\}}$$

while in the algebra of functions we have

$$\frac{}{\mathsf{emptyset}_\tau : unit \to \{\tau\}} \qquad \frac{}{\mathsf{union}_\tau : \{\tau\} \times \{\tau\} \to \{\tau\}}$$

Notations. We shall use $\mathcal{R}$ for $\mathcal{M}(\{\}, \cup)$. Wadler has already noted the usefulness of this extension to a monad and termed it a *ringad*, hence the $\mathcal{R}$; see [58]. We shall also use the shorthand notations $e_1 \upharpoonleft e_2 := \{e_1\} \cup e_2$ and $\{e_1, \ldots, e_n\} := \{e_1\} \cup \ldots \cup \{e_n\}$.

The relational algebra operation of selection normally requires a boolean type. Some conceptual economy is possible if we simulate the booleans by representing *true* as $\{()\}$ and *false* as $\{\}$, which are the two values of type $\{unit\}$ [4]. Note that $b_1 \cup b_2$ is the disjunction of b_1 and b_2.

[4] An interesting remark is that $\{\!|unit|\!\}$ can simulate the type of natural numbers. $\mathsf{sr\_add}_{bag}$ becomes the usual recursion on natural numbers for this type.

With this representation, selection is definable in $\mathcal{R}$ as

$$\mathsf{select}\ x\ \mathsf{from}\ S\ \mathsf{where}\ P := \Phi x{\in}S.\,\Pi_1(\mathsf{cartprod}(\{x\},P))$$

where $x:\sigma$, $S:\{\sigma\}$, and $P:\{unit\}$ is a predicate with free variable x. The trick here is that if P is false, then it is empty. Consequently, $\mathsf{cartprod}(\{x\},P)$ is empty. So x does not contribute to the result.

4.2 Adding non-monotonic operations

The remaining operator of the flat relational algebra is the set difference operator. The weaker operation of the positive relational algebra is the set intersection operator.

Theorem 4.2 *Intersection cannot be expressed in $\mathcal{R}$.*

Proof. Define an ordering $\preceq^\sigma$ on complex objects of type σ as follows:

- $q \preceq^b q$, for any object q of base type b.
- $(q_1,q_2) \preceq^{\sigma\times\tau} (r_1,r_2)$, if $q_1 \preceq^\sigma r_1$ and $q_2 \preceq^\tau r_2$.
- $Q \preceq^{\{\sigma\}} R$, if for each $q \in Q$, there is $r \in R$, and $q \preceq^\sigma r$.

It can be checked that all functions definable in $\mathcal{R}$ are monotone with respect to this ordering, while set intersection is not. □

It follows that set difference cannot be defined in $\mathcal{R}$. A similar argument shows that equality cannot be defined in $\mathcal{R}$ nor can membership or subset predicates. Also, the nesting operation of the nested relational algebra cannot be defined in $\mathcal{R}$. This is one of two mutually-definable operations nest_1 and nest_2. For example, nest_2 is the right-nesting operation of type $\{\sigma\times\tau\}\to\{\sigma\times\{\tau\}\}$.

Thus a further extension is still necessary. It turns out that we can make this extension in several equivalent ways. We shall use the notation $\mathcal{R}(\Sigma)$ for $\mathcal{R}$ augmented with additional primitive operations Σ. For example $\mathcal{R}(=)$ is $\mathcal{R}$ augmented calculus-style with

$$\frac{e_1:\sigma \quad e_2:\sigma}{e_1=e_2:\{unit\}}$$

or a corresponding function $\mathsf{eq}_\sigma:\sigma\times\sigma\to\{unit\}$ in the functional algebra.

Theorem 4.3 *Let Σ be any additional primitive signature. The following languages are equivalent: $\mathcal{R}(\mathsf{intersection},\Sigma)$, $\mathcal{R}(=,\Sigma)$, $\mathcal{R}(\mathsf{difference},\Sigma)$, $\mathcal{R}(\mathsf{subset},\Sigma)$, $\mathcal{R}(\mathsf{member},\Sigma)$, and $\mathcal{R}(\mathsf{nest},\Sigma)$.*

Proof. To show these equivalences we have to exhibit translations between these functions. In the course of this, we also provide the usual complement of boolean functions.

Given equality, define $\eta_\cap(x,y) := \Pi_1(\mathsf{cartprod}(\{x\}, \mathsf{eq}(x,y)))$. Then $\eta_\cap(x,y)$ returns the singleton set $\{x\}$ if $\mathsf{eq}(x,y)$ is true and $\{\}$ otherwise. Set intersection is now obtained by flat-mapping this function over the cartesian product: $\mathsf{intersection}(x,y) := \Phi z \in \mathsf{cartprod}(x,y).\, \eta_\cap(z)$. Conversely, equality may be defined from intersection by $\mathsf{eq}(x,y) := \mathsf{map}(\mathsf{ter})(\mathsf{intersection}(\{x\},\{y\}))$. Thus $\mathcal{R}(=,\Sigma) = \mathcal{R}(\mathsf{intersection},\Sigma)$.

Recall that we have implemented the booleans by the two values of type $\{unit\}$, using $\{\}$ for false and $\{()\}$ for true. Disjunction and conjunction are then directly implemented by union and intersection. To implement negation, consider the relation $\{(\{\},\{()\}),\ (\{()\},\{\})\}$ which pairs false with true and true with false. We can select from this relation the tuple whose left component matches the input and project the right component:

$$\mathsf{not}(x) := \mathsf{flatten}(\Pi_2(\mathsf{select}\ y\ \mathsf{from}\ \{(\{\},\{()\}),\ (\{()\},\{\})\}\ \mathsf{where}\ \mathsf{eq}(x,\ \pi_1(y))))$$

We can use these to implement existential and universal quantification. Define:

$$\begin{aligned} \mathsf{some}\ x\ \mathsf{in}\ S\ \mathsf{satisfies}\ P &:= \Phi x{\in} S.\, P \\ \mathsf{every}\ x\ \mathsf{in}\ S\ \mathsf{satisfies}\ P &:= \mathsf{not}(\mathsf{some}\ x\ \mathsf{in}\ S\ \mathsf{satisfies}\ \mathsf{not}(P)) \end{aligned}$$

Thus, if $P : \{unit\}$ is a predicate expression with a free variable x, we can represent the predicate calculus notation $\exists x \in S\,.\,P$ as $\mathsf{some}\ x\ \mathsf{in}\ S\ \mathsf{satisfies}\ P$ and the predicate calculus notation $\forall x \in S\,.\,P$ as $\mathsf{every}\ x\ \mathsf{in}\ S\ \mathsf{satisfies}\ P$.

This brings us to the implementation of set difference in terms of equality:

$$\mathsf{difference}(R,S) := \mathsf{select}\ x\ \mathsf{from}\ R\ \mathsf{where}\ (\mathsf{every}\ y\ \mathsf{in}\ S\ \mathsf{satisfies}\ \mathsf{not}(\mathsf{eq}(x,y))).$$

Noting that intersection is easily obtained from difference, we now have $\mathcal{R}(=,\Sigma) = \mathcal{R}(\mathsf{intersection},\Sigma) = \mathcal{R}(\mathsf{difference},\Sigma)$.

Equality can be obtained from membership by: $\mathsf{eq}(x,y) := \mathsf{member}(x,\{y\})$. Membership is obtained from equality by: $\mathsf{member}(x,S) := \mathsf{some}\ y\ \mathsf{in}\ S\ \mathsf{satisfies}\ \mathsf{eq}(x,y)$. The mutual dependence of member and subset is immediate. So we have $\mathcal{R}(=,\Sigma) = \mathcal{R}(\mathsf{member},\Sigma) = \mathcal{R}(\mathsf{subset},\Sigma)$.

Finally, we examine nest_2, which can be derived from equality as follows. First consider a function $f : \sigma \times \{\sigma \times \tau\} \to \sigma \times \{\tau\}$ such that $f(x,S)$ returns the pair $(x, \{y \mid (x,y) \in S\})$. It can be defined as $f(x,S) := (x, \Pi_2(\mathsf{select}\ y\ \mathsf{from}\ S\ \mathsf{where}\ \mathsf{eq}(x,\ \pi_1(y))))$. Now $\mathsf{nest}_2(R)$ is obtained by pairing each member of the left column of R with the whole of R and mapping f over the relation so formed: $\mathsf{nest}_2(R) := \mathsf{map}(f)(\mathsf{pairwith}_1(\Pi_1\ R,\ R))$.

Conversely, we show that difference can be derived from nest_1 and nest_2. First, we observe that negation can be obtained from nest_1 as follows. Writing T for $\{()\}$ and F for $\{\}$, consider a boolean variable x, and form the set $\{((F,F),T),((T,F),F),((F,T),x)\}$ and apply nest_1 and then Π_1. This yields $\{\{(F,F),(F,T)\},\{(T,F)\}\}$ if x is true and $\{\{(T,F)\},\{(T,F),(F,T)\}\}$ if x is false. Now apply

$$\mathsf{flatten} \circ \mathsf{map}(\mathsf{map}(\mathsf{fst} \times \mathsf{snd}) \circ \mathsf{cartprod} \circ \langle \mathsf{id}, \mathsf{id} \rangle)$$

to obtain $\{(F,F),(F,T),(T,F)\}$ and $\{(T,F),(T,T),(F,F),(T,T)\}$ respectively. Since F is an empty set its cartesian product with anything is an empty set, so applying $\mathsf{flatten} \circ \mathsf{map}(\mathsf{fst} \circ \mathsf{cartprod})$ gives us F and T respectively.

To obtain the difference of two sets R and S, pair each member of R with F and each member of S with T and nest on the left column. That is, compute $(\mathsf{nest}_2 \circ \mathsf{union})(\mathsf{pairwith}_1(R, F),\ \mathsf{pairwith}_1(S, T))$. Tuples in this relation are of one of the three form $(a, \{F, T\})$, $(b, \{F\})$, and $(c, \{T\})$, where $a \in R \cup S$, $b \in \mathsf{difference}(R, S)$, and $c \in \mathsf{difference}(S, R)$. If we now apply $\mathsf{flatten}$ to the right-hand column and apply sng to the left, these tuples are of the form $(\{a\}, T)$, $(\{b\}, F)$, and $(\{c\}, T)$. Negate the second column (we have just shown that this can be done with nest_1) and apply cartesian product to each tuple to obtain only those elements in $\mathsf{difference}(R, S)$. This completes the proof. □

A related result was proved by Gyssens and Van Gucht [21] who showed that these non-monotonic operators are inter-definable in the language of Schek and Scholl [51] when the $\mathsf{powerset}$ operator was made available as an additional primitive. In view of two results that follow, Theorem 4.4 (which indicates can our language is of polynomial time complexity) and Theorem 4.5 (which shows that our language is equivalent to that of Schek and Scholl), Theorem 4.3 is a significant improvement.

Given this equivalence result, we choose one of the non-monotonic operations, namely equality, and add it to $\mathcal{R}$.

Theorem 4.4 (PTIME computability) *Assuming that the functions denoted by the primitive function symbols to be computable polynomial time with respect to the size of their input. Then any functions that are definable in $\mathcal{R}(=, \Sigma)$ are computable in polynomial time with respect to the size of their input, for any reasonable definition of complex object size.*

Proof. We consider the presentation of $\mathcal{R}(=, \Sigma)$ as an algebra of functions. For each function f definable this way a polynomial time-bound function $|f| : \mathbb{N} \to \mathbb{N}$ is given by

$$|f|(n) = \begin{cases} |g|(n) + |h|(n) & \text{if } f \text{ is } \langle g, h\rangle \\ |g|(|h|(n)) & \text{if } f \text{ is } g \circ h \\ n \cdot |g|(n) & \text{if } f \text{ is } \mathsf{map}(g) \\ O(n^{k_p}) & \text{if } f \text{ is a primitive function } p\text{, bound is by assumption} \\ O(n) & \text{otherwise} \end{cases}$$

□

In fact this result can be strengthened by showing that the implementation suggested by the operational semantics of structural recursion is also polynomial.

4.3 Relating $\mathcal{R}(=)$ to other nested relational algebras.

The language of Thomas and Fischer [57] is the most widely known of nested relational algebras. The language of Schek and Scholl [51] is an extension of Thomas and Fischer's with a recursive projection operator. The language of Colby [16] is in turn an extension of Schek and Scholl's that makes all operators recursive. It is a theorem of Colby [16] that her algebra is expressible in the algebra of Thomas and Fischer.

This result can be strengthened by showing that $\mathcal{R}(=)$ coincides in expressive power with these three nested relational languages. Hence it can be argued that $\mathcal{R}(=)$ possesses just the right amount of expressive power for manipulating nested relations.

A detailed description of Thomas and Fischer's language is required for proving this result.

- Union of sets. $\mathsf{union}_\sigma : \{\sigma\} \times \{\sigma\} \to \{\sigma\}$. This is already present in $\mathcal{R}(=)$.
- Intersection of sets. $\mathsf{intersection}_\sigma : \{\sigma\} \times \{\sigma\} \to \{\sigma\}$. This is definable in $\mathcal{R}(=)$ by Theorem 4.3.
- Set difference. $\mathsf{difference}_\sigma : \{\sigma\} \times \{\sigma\} \to \{\sigma\}$. This is definable in $\mathcal{R}(=)$ by Theorem 4.3.
- Relational nesting. $\mathsf{nest2}_{\sigma,\tau} : \{\sigma \times \tau\} \to \{\sigma \times \{\tau\}\}$. This is definable in $\mathcal{R}(=)$ by Theorem 4.3.
- Relational unnesting. $\mathsf{unnest2}_{\sigma,\tau} : \{\sigma \times \{\tau\}\} \to \{\sigma \times \tau\}$. It can be defined in $\mathcal{R}(=)$ as given in Section 2.3.
- Cartesian product. $\mathsf{cartprod}_{\sigma,\tau} : \{\sigma\} \times \{\tau\} \to \{\sigma \times \tau\}$. It can be defined in $\mathcal{R}(=)$ as given in Section 3.2.
- Their projection operator is the relational projection. General relational projection works on multiple columns and has the form $\mathsf{projection}(f)$. This can be interpreted in $\mathcal{R}(=)$ as $\mathsf{map}(f)$, where a restriction is placed on the form of f: it must be built entirely from fst, snd, $\langle \cdot, \cdot \rangle$, $\cdot \circ \cdot$, and id.
- Their selection operator is the relational selection and has the form $\mathsf{selection}(f, g)$. It can be interpreted in $\mathcal{R}(=)$ as $\mathsf{select}\ x\ \mathsf{from}\ R\ \mathsf{where}\ \mathsf{eq}(f\ x,\ g\ x)$, where R stands for the input relation. However, the same restriction given in $\mathsf{projection}$ is placed on f and g.
- As in the traditional relational algebra, Thomas and Fischer used letters to represent input relations. Without loss of generality, only one input relation is considered. We reserve the letter R for this purpose and it is assumed to be distinct from all other variables. Finally, constant relations are written down directly. For example, $\{\{\}\}$ is the constant relation whose only element is the empty set.

A query is just an expression e of complex object type such that R is its only free variable. We view such a *Thomas&Fischer* query as the function f such that $f(R) = e$.

Theorem 4.5 *$\mathcal{R}(=) = Thomas\&Fischer = Schek\&Scholl = Colby$.*

Proof. It is sufficent to show that every function in $\mathcal{R}(=)$ is also in *Thomas&Fischer*. Since $=$ is clearly definable in terms of $=_b$ and not, we can instead prove that every function definable in $\mathcal{R}(=_b, \mathsf{not})$ is also definable in *Thomas&Fischer*.

Let $encode^\sigma : \sigma \to \{unit \times \sigma\}$ be the function $encode^\sigma(o) = \{((),\ o)\}$. Let $decode^\tau : \{unit \times \tau\} \to \tau$ be the partial function $decode^\tau\{((),\ o)\} = o$. Note that both $encode^\sigma$ and $decode^\tau$ are definable in *Thomas&Fischer* when σ and τ are both products of set types. Suppose

Claim. For every closed morphism $f : \sigma \to \tau$ in $\mathcal{R}(=_b, \mathsf{not})$, for every complex object type δ, there is a query $f' : \{\delta \times \sigma\} \to \{\delta \times \tau\}$ in *Thomas&Fischer* such that f' denotes the same function as $\mathsf{map}(\mathsf{id} \times f)$.

Then calculate as below:

•	$decode \circ f' \circ encode$	
=	$decode \circ \mathsf{map}(\mathsf{id} \times f) \circ encode$	By the above claim.
=	$decode \circ \mathsf{sng} \circ \langle \mathsf{ter}, f \rangle$	Because $encode = \mathsf{sng} \circ \langle \mathsf{ter}, \mathsf{id} \rangle$.
=	f	

It remains to provide a proof of the claim. This is not difficult if one define f' by induction on the structure of f. The complete proof can be found in Wong [64]. We provide the case when f has the form $\mathsf{map}(g)$ for illustration.

To define $\mathsf{map}(g)' : \{\delta \times \{\sigma\}\} \to \{\delta \times \{\tau\}\}$, assume by hypothesis that $g' : \{\delta \times \sigma\} \to \{\delta \times \tau\}$ exists. Define $A(R) := \mathsf{projection}\langle \mathsf{fst} \circ \mathsf{fst},\ \mathsf{id} \circ \mathsf{snd}\rangle(\mathsf{nest}_2(g'(\mathsf{unnest}_2(\mathsf{map}(\langle \mathsf{id}, \mathsf{snd}\rangle)(R)))))$. Then $A(R) = \mathsf{map}(\mathsf{id} \times g)(R_1)$, where R_1 contains exactly those pairs in R whose right component is nonempty. Define $B(R) := \mathsf{cartprod}(\mathsf{projection}(\mathsf{fst} \circ \mathsf{fst})(\mathsf{selection}(\mathsf{snd} \circ \mathsf{fst},\ \mathsf{snd})(\mathsf{cartprod}(R, \{\{\}\}))), \{\{\}\})$. Then $B(R) = \mathsf{map}(\mathsf{id} \times g)(R_2)$, where R_2 contains exactly those pairs in R whose right component is empty. Finally, we set $\mathsf{map}(g)'(R) := \mathsf{union}(A(R), B(R))$.

As can be seen, the definition for $\mathsf{map}(g)'$ is not simple. There are two reasons for this. The first reason is that g may not satisfy the severe restriction that Thomas and Fischer put on their projection operation. The second reason is that the only way to implement the emptiness test in the language of Thomas and Fischer is via their selection operation. □

It is an immediate corollary of this theorem that

Corollary 4.6 *Every function from flat relations to flat relations expressible in* $\mathcal{R}(=)$ *is also expressible in flat relational algebra.*

Proof. It is known from Paredaens and Van Gucht [49] that every function from flat relations to flat relations expressible in *Thomas&Fischer* is also expressible in flat relational algebra. The corollary thus follows from Theorem 4.5. □

In fact, elsewhere we are able to strengthen the theorem of Paredaens and Van Gucht to a general theorem on the conservative extension property of $\mathcal{R}(=)$ and its various extensions. That is, we can show that the definability of a function in $\mathcal{R}(=)$ is independent of any restriction that can be imposed on the depth of set nesting in intermediate data [34, 35, 63].

4.4 Conditionals

An if-then-else construct is often needed in programming. Consider the function $\mathsf{cond}_\sigma : \{unit\} \times (\sigma \times \sigma) \to \sigma$ such that $\mathsf{cond}(B, q, r)$ returns q if B is nonempty and r otherwise. This function is not definable in $\mathcal{R}(=)$ at all types σ. The techniques used in the proof of Theorem 4.3 allow us to define it when σ has the form $\{\sigma_1\} \times \cdots \times \{\sigma_n\}$. However, cond_σ is not definable in $\mathcal{R}(=)$ when σ is a base type, because any function in $\mathcal{R}(=)$ whose output type is a base type must either be a constant function or is a chain of projection operations. We find it useful to have a conditional in the calculus-style

$$\frac{e : \{unit\} \quad e_1 : \sigma \quad e_2 : \sigma}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \sigma}$$

or in the algebra-style

$$\frac{}{\mathsf{cond}_\sigma : \{unit\} \times \sigma \times \sigma \to \sigma}$$

It should be noted [64] that the addition of the conditional is a convenience; it does not greatly affect the expressive power of $\mathcal{R}(=)$.

4.5 Discussion

Nested relational algebras were introduced to relax the first normal form restriction originally imposed by Codd [15] and considered unacceptable in many modern applications [41, 40, 26, 29]. The earliest definition was that of Jaeschke and Schek [29] who allowed the components of tuples to be sets of atomic values. That is, nesting of relations was restricted to two levels. This restriction was relaxed by Thomas and Fischer [57] who allowed relations to be nested to arbitrary depth. Their algebraic query language consisted of the operators of flat relational algebra generalized to nested relations together with two operators for nesting and unnesting relations. However, their operators can only be applied to the outermost level of nested relations. Before a deeply nested relation could be manipulated, it is necessary to bring it up to the outermost level by a sequence of unnest operations; and after the top-level manipulation, to push the result back down to the right level by nesting. However, *nest* and *unnest* are not mutual inverses, and some care has to be taken during restructuring, as can be gauged from the full proof of Theorem 4.5 [64].

This constant need for restructuring was eliminated by Schek and Scholl [51] who introduced a recursive projection operator for navigation and later by Colby [16] who made all her operators recursive. Their method is *ad hoc* in the sense that individual definitions are required for each recursive operator. For example, the semantics given by Schek and Scholl [51] for the recursive projection operator has over 10 cases.

The $\mathsf{map}(\cdot)$ construct of $\mathcal{R}(=, \mathsf{cond})$ allows all operations to be performed at all levels of nesting; thus completely eliminating the need for restructuring through nest and unnest, as in Thomas and Fischer's algebra. The recognition that any function can be passed to $\mathsf{map}(\cdot)$ at once simplifies the language; thus eliminating the need for *ad hoc* operations and complicated semantics. Every expression construct in $\mathcal{R}(=, \mathsf{cond})$ enjoys the same status and can be freely mixed as long as typing rules are not violated; thus eliminating the need for special syntax for the parameters to different operators. In addition, it provides a framework with which to extend nested relational algebra to other collection types, and allows us to reason about languages with external functions such as the aggregate operations of SQL [28]. We therefore believe that $\mathcal{R}(=, \mathsf{cond})$ may be profitably considered as the "right" nested relational algebra.

5 The power of structural recursion and languages with powerset

We have shown that the nested relational algebra admits an elegant formulation using operations on complex objects suggested by the concept of monad. At the same time the nested relational algebra has severe limitations on expressiveness. Indeed, in view of corollary 4.6 and [4, 14], there are also polynomial time operations such as transitive closure and parity that cannot be defined in $\mathcal{R}(=, \mathsf{cond})$. In this section we consider constructs that extend the expressive power of $\mathcal{R}(=, \mathsf{cond})$. In subsection 5.1 we discuss Abiteboul and Beeri's complex object algebra which essentially adds the finite powerset operations to $\mathcal{R}(=, \mathsf{cond})$. This increases the expressive power, but seems to suggest an inflexible programming style. In subsection 5.2 we show that structural recursion can express efficient polymorphic algorithms for some of the functions that are beyond the reach of the nested relational algebra. Finally, in subsection 5.3 we show that in the absence of external functions the powerset operation can express the functions defined by structural recursion, albeit in an inefficient and non-polymorphic manner.

5.1 Abiteboul and Beeri's complex object algebra

In view of theorem 4.4, $\mathsf{powerset}$ is not definable in $\mathcal{R}(=, \mathsf{cond})$. In [1], Abiteboul and Beeri introduce three languages that can all express powerset, and they show them to be equivalent: a "complex object" algebra and calculus, and an extension to datalog with certain higher-order predicates such as subset and membership. Gyssens and Van Gucht [22] show that several augmentations of the nested relational algebra with recursive and iterative constructs are equivalent to the augmentation with powerset.

If we add for each complex object type σ the primitive $\mathsf{powerset}^{\sigma} : \{\sigma\} \to \{\{\sigma\}\}$, we obtain a formalism equivalent to the complex object algebra in [1]. For the purpose of this paper, let us define Abiteboul and Beeri's algebra as

$$\mathcal{A}\&\mathcal{B} := \mathcal{R}(=, \mathsf{cond}, \mathsf{powerset})$$

Abiteboul and Beeri show how to express transitive closure of a relation R in $\mathcal{A}\&\mathcal{B}$, by selecting from $\mathsf{powerset}(\mathsf{cartprod}(\Pi_1(R), \Pi_2(R)))$ those relations which are transitive and contain R and then taking their intersection. The intersection of a set of sets, $S : \{\{\tau\}\}$ is readily defined, even in $\mathcal{R}(=, \mathsf{cond})$, via complements:

$$\bigcap S := \mathsf{difference}(\mathsf{flatten}(S),\ \Phi s^{\{\tau\}} \in S.\, \mathsf{difference}(\mathsf{flatten}(S), s))$$

We remark that a test for equal cardinality can also be expressed in $\mathcal{A}\&\mathcal{B}$: given sets S and T we can construct $\mathsf{powerset}(\mathsf{cartprod}(S, T))$ and then test whether it contains a bijection between S and T. Then we can test for parity of the cardinality of a set S by testing whether for some subset $T \subseteq S$, the sets T and $\mathsf{difference}(S, T)$ have equal cardinality.

We have not discussed operational semantics for the languages we have considered, but clearly these expressions of transitive closure and parity using powerset suggest exponential time algorithms (obvious implementaions are even be exponential space) when in fact the queries themselves are polynomial. In fact, by corollary 4.6, it is clear that queries such as transitive closure, equal cardinality, and parity, are not definable in $\mathcal{A}\&\mathcal{B}$ without a potentially costly excursion through a powerset. This observation, made in [9], begs the question: is there an "efficient" way of programming these queries in $\mathcal{A}\&\mathcal{B}$? This is a delicate question since it depends on accepting a "reasonable" notion of operational semantics for $\mathcal{A}\&\mathcal{B}$. Suciu and Paredaens [54] show that if we adopt the usual, eager, evaluation strategy for queries, then any $\mathcal{A}\&\mathcal{B}$ expression for transitive closure must construct an intermediate result of exponential size, hence obtaining an EXPSPACE lower bound. Abiteboul and Hillebrand [2] show that an operational semantics with pipelining optimizations yields a PSPACE (but still EXPTIME) algorithm for the $\mathcal{A}\&\mathcal{B}$ expression of transitive closure mentioned above. One is strongly inclined to think that $\mathcal{A}\&\mathcal{B}$ does not offer a flexible enough programming style to be able to code transitive closure, or parity, efficiently.

As we shall see in subsection 5.2, structural recursion can express efficient algorithms for transitive closure and parity in a rather straightforward manner.

We can make another aspect of this inflexibility precise by considering cardinality. It turns out that cardinality, as a function into a primitive type $\mathbb{N}$ of natural numbers, is not definable, no matter what arithmetic functions we take as primitives. Indeed, let $\mathcal{A}\&\mathcal{B}(\mathbb{N})$ be the extension of $\mathcal{A}\&\mathcal{B}$ with a primitive type $\mathbb{N}$ and an *arbitrary* primitive arithmetic signature of constants and functions of the form

$$c : \mathbb{N} \qquad\qquad p : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$$

We must be careful in what we mean by cardinality not being definable, because for each finite complex object type σ not containing $\mathbb{N}$, there is a specific and trivial expression c_σ for the cardinality function of type $\{\sigma\} \to \mathbb{N}$. That is because all the sets of type $\{\sigma\}$ are "known" and definable in the language (recall that it is assumed that σ doesn't contain $\mathbb{N}$), so we can just compare the argument of c_σ with each of them and build the answer into c_σ. Of course, the expressions c_σ do not depend uniformly on the type. It is precisely such a uniform, "parametric," or "polymorphic" definition that does not exist.

To describe precisely polymorphic definitions, we introduce type variables α, β, etc., and consider complex object type expressions

$$\theta ::= \alpha \mid unit \mid b \mid \theta \times \theta \mid \{\theta\}$$

To avoid technical problems with type variables occurring in the type of usual variables, free or bound, we consider only expressions in the functional algebra formalism, since they do not have bound variables. Moreover we are interested only in closed variable-free expressions; call them polymorphic expressions. Type variables may now occur in polymorphic expressions, namely in the subscripts of id, fst, sng, ter, snd, $\mathsf{flatten}$, $\mathsf{pairwith}_2$, $\mathsf{emptyset}$, union, eq, and $\mathsf{powerset}$; and we can substitute for them. For example:

$$(\mathsf{snd}_{\{\alpha \times unit\},\alpha})[\{unit\}/\alpha] \equiv \mathsf{snd}_{\{\{unit\} \times unit\},\{unit\}}$$

We say that cardinality is *polymorphically definable* if there exists a polymorphic expression $count_{set} : \{\alpha\} \to \mathbb{N}$, where α is a type variable, such that for each complex object type σ, the expression $count_{set}[\sigma/\alpha] : \{\sigma\} \to \mathbb{N}$ denotes the cardinality function from $\{\sigma\} \to \mathbb{N}$.

Theorem 5.1 *Cardinality is not polymorphically definable in $\mathcal{A}\&\mathcal{B}(\mathbb{N})$.*

Proof. For any complex object q, let $\max(q)$ be the largest natural number that occurs in q (0 if none occurs). We show

Claim. For each polymorphic expression f of $\mathcal{A}\&\mathcal{B}(\mathbb{N})$, there exists an increasing map $\varphi_f : \mathbb{N} \to \mathbb{N}$ such that for any instantiation $\hat{f}$ obtained by substituting complex object types for all the type variables in f we have $\max(\hat{f}(q)) \leq \varphi_f(\max(q))$.

Proof of claim. Take $\varphi_{id} := \varphi_{fst} := \cdots := \varphi_{cond} := \varphi_{powerset} :=$ the identity on $\mathbb{N}$, which explains why φ_f gives a bound for all instantiations of f. For any arithmetic primitive p, we take $\varphi_p(n) := \max_{x \leq n, y \leq n}(p(x, y))$. The claim follows by induction on f.

To prove the theorem, assume a polymorphic cardinality function $count_{set} : \{\alpha\} \to \mathbb{N}$ exists, and let $m = \varphi_{count}(0)$. Let σ be a complex object type not containing $\mathbb{N}$ such that $\{\sigma\}$ has more than m elements; for example, σ can be of the form $\{\cdots\{unit\}\cdots\}$. Then, by the claim, $count_{set}[\sigma/\alpha]$ cannot denote the cardinality function of type $\{\sigma\} \to \mathbb{N}$. □

As we shall see next, structural recursion allows a straightforward polymorphic definition of cardinality.

5.2 The power of structural recursion

The most powerful and at the same time flexible language on sets that we consider in this paper uses structural recursion, specifically on the following construct first introduced in section 2:

$$\begin{array}{lcl} g(\{\}) & = & e \\ g(x \upharpoonleft S) & = & i(x, g(S)) \end{array} \qquad\qquad \frac{e : \tau \qquad i : \sigma \times \tau \to \tau}{g : \{\sigma\} \to \tau}$$

We have denoted g by $\mathsf{sr\_add}_{set}(i, e)$. As we have seen in subsection 2.2, we can express $\mathsf{sr\_comb}_{set}$ in terms of $\mathsf{sr\_add}_{set}$. Hence $\mathsf{ext}(\cdot)$ and Φ can also be expressed with $\mathsf{sr\_add}_{set}$. A direct formulation of this is $\Phi x \in S.\, T := f\, S$ where

$$\begin{array}{lcl} f(\{\}) & = & \{\} \\ f(x \upharpoonleft Z) & = & T \cup Z \end{array}$$

The conditional can be expressed as $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 := h\, e$ where

$$\begin{array}{lcl} h(\{\}) & = & e_2 \\ h(u \upharpoonleft Z) & = & e_1 \end{array}$$

Moreover, as we have seen in section 2, structural recursion can express powerset, $\mathsf{powerset}(S) := p\, S$, where

$$\begin{array}{lcl} p(\{\}) & = & \{\{\}\} \\ p(x \upharpoonleft S) & = & p(S) \cup \mathsf{map}(h)(p\, S) \end{array}$$

and where $h(Z) := x \upharpoonleft Z$.

Hence, structural recursion together with those primitives in $\mathcal{R}(=)$ which are distinct from Φ is at least as powerful as $\mathcal{A}\&\mathcal{B}$. Given the problems that arise for $\mathcal{A}\&\mathcal{B}$, some proven and some conjectured, it will be interesting to show that structural recursion offers a flexible programming style which allows for polymorphically expressing efficient algorithms for parity test, set cardinality, and transitive closure.

Parity test Notice that the predicate $even : \{\alpha\} \to \{unit\}$ satisfies:

$$\begin{array}{lcl} even(\{\}) & = & true \\ even(x \upharpoonleft S) & = & \mathsf{if}\ \mathsf{member}(x, S)\ \mathsf{then}\ even\ S\ \mathsf{else}\ \mathsf{not}((even\ S)) \end{array}$$

This is not quite a use of structural recursion on collections as we have defined it. It is a more general form:

$$\begin{array}{lcl} g'(\mathsf{empty}) & = & e \\ g'(\mathsf{add}(x, C)) & = & j(x, C, g'(C)) \end{array} \qquad \frac{e : \tau \qquad j : \sigma \times \{\sigma\} \times \tau \to \tau}{g' : \mathsf{coll}(\sigma) \to \tau}$$

Recalling an idea of Kleene (who used it to code the predecessor function in the lambda calculus), this apparently stronger form can be obtained from a simple structural recursion $g' = \mathsf{snd} \circ \mathsf{sr\_add}(i', e')$ where $i' : \sigma \times (\mathsf{coll}(\sigma) \times \tau) \to \mathsf{coll}(\sigma \times \tau)$ is defined by $i'(x, C, a) := (\mathsf{add}(x, C), j(x, C, a))$ and $e' := (\mathsf{empty}, e)$.

Suitably generalized forms of left-commutativity and left-idempotence

$$\begin{array}{lcl} j(x, y \upharpoonleft S, j(y, S, a)) & = & j(y, x \upharpoonleft S, j(x, S, a)) \\ j(x, x \upharpoonleft S, j(x, S, a)) & = & j(x, S, a) \end{array}$$

constitute sufficient conditions for g' to be well-defined on sets. They are easily verified for the definition of $even$. In the case of bags we only require generalized left-commutativity; no conditions are needed for lists. [5]

Set cardinality As promised earlier, $count_{set} : \{\alpha\} \to \mathbb{N}$ is polymorphically definable with structural recursion, in a similar manner as $even$:

$$\begin{array}{lcl} count_{set}(\{\}) & = & 0 \\ count_{set}(x \upharpoonleft S) & = & \mathsf{if}\ \mathsf{member}(x, S)\ \mathsf{then}\ count_{set}(S)\ \mathsf{else}\ 1 + (count_{set} S) \end{array}$$

Transitive closure It is clear that the implementation of transitive closure as given in section 5.1 is severely inefficient. We now show that a much better algorithm for transitive closure can be expressed with structural recursion. First, we need binary relation composition, which is expressible as

$$R\#S := \mathsf{map}(f)(\mathsf{select}\ w\ \mathsf{from}\ \mathsf{cartprod}(R, S)\ \mathsf{where}\ \pi_2(\pi_1(w)) = \pi_1(\pi_2(w)))$$

[5] In fact, with some higher-order lambda calculus, we can also express an efficient algorithm for testing equality of cardinality.

where

$$f(v) := (\pi_1(\pi_1(v)), \pi_2(\pi_2(v)))$$

Now consider $i : (\alpha \times \alpha) \times \{\alpha \times \alpha\} \to \{\alpha \times \alpha\}$ defined by

$$i(r, T) = \{r\} \cup T \cup (\{r\}\#T) \cup (T\#\{r\}) \cup (T\#\{r\}\#T)$$

and then transitive closure, $TC : \{\alpha \times \alpha\} \to \{\alpha \times \alpha\}$, is given by

$$\begin{array}{rcl} TC(\{\}) & = & \{\} \\ TC(s \upharpoonleft R) & = & i(s,\ TC(R)) \end{array}$$

We have to verify that TC is well-defined. That is, that the semantics of i satisfiy the commutativity and idempotence conditions on the right set of values, and that the meaning of TC is in fact the transitive closure operator. In what follows, we will perpetrate a slight abuse of notation by writing semantic proofs of semantic facts in programming syntax. (In fact, the proofs for the next lemma can all be formalized in syntax too, by using one of the logics described in [8].) We still need one more notation: the *semantic* transitive closure of R is denoted by R^+.

Proposition 5.2

1. *$\{\}$ is transitive. If T is transitive then $i(r, T)$ is also transitive.*
2. *Let T be transitive. Then $i(r, i(s, T)) = i(s, i(r, T))$ and $i(r, i(r, T)) = i(r, T)$.*
3. *If T is transitive then $i(r, T) = (r \upharpoonleft T)^+$.*
4. *$i(r, R^+) = (r \upharpoonleft R)^+$.* □

The key observation in proving part 1 is the following simple fact: for any R, $\{s\}\#R\#\{s\} \subseteq \{s\}$. Part 2, which implies that TC is correctly defined (working with a range consisting only of transitive relations), is shown using part 1. Part 3 follows immediately from part 1, and part 4 from part 3. Part 4 of the lemma is the essential step in showing by structural induction on the insert presentation of sets, that for any R, $TC(R) = R^+$.

This algorithm for transitive closure resembles Warshall's algorithm, except that we are doing edge insertion rather than node insertion. To obtain Warshall's algorithm, suppose we are given a set of nodes $V : \{\alpha\}$ and a set of edges $E : \{\alpha \times \alpha\}$ among these nodes. Then, the transitive closure of E is given by $W(V)$ where W is defined by

$$\begin{array}{rcl} W(\{\}) & = & E \\ W(v \upharpoonleft A) & = & W(A) \cup W(A)\#\{(v, v)\}\#W(A) \end{array}$$

Indeed, one can show that W is well-defined and that for any $A \subseteq V$, $W(A)$ is the set of pairs of nodes which are connected by paths whose intermediate nodes all belong to A.

Warshall's algorithm runs in $O(n^3)$ time while the edge insertion algorithm runs in $e \cdot n^2$ time, where n is the number of nodes and e is the number of edges. In any case, these are efficient algorithms for transitive closure, in comparison to the $\mathcal{A}\&\mathcal{B}$ query mentioned earlier. In the spirit of Warshall's algorithm, one can also represent Floyd's shortest paths algorithm.

5.3 $\mathcal{A}\&\mathcal{B}$ is equivalent to structural recursion when external functions are absent

While we have explained through theorem 5.1 in what sense structural recursion is strictly more powerful than $\mathcal{A}\&\mathcal{B}$, we still want to explain the intuition that since $\mathcal{A}\&\mathcal{B}$ can do certain least fixed points, in fact enough to simulate a Datalog-like language with predicates on sets [1], it will be able to express the functions defined by structural recursion, which are also least relations given appropriate properties. It will turn out that we can justify this intuition formally, but our reduction from structural recursion to $\mathcal{A}\&\mathcal{B}$ will *not* be polymorphic.

The difficulty in formalizing this intuition comes from the fact that in order to express such least relations with powerset and intersection of set of sets, we need some kind of "universe" which collects all the elements that could be involved in the computation of the least fixed point. This was simple to get in the case of transitive closure, it was simply all the elements occurring in the relation. Our situation is more general and we quickly realize that nothing can be done in the presence of primitive functions.

Thus we consider $\mathcal{A}\&\mathcal{B}(C)$ with only finite many constants $C := \{c_1, \ldots, c_n\}$ of one base type ι.

Theorem 5.3 *The class of functions that are expressible in $\mathcal{A}\&\mathcal{B}(C)$ is (essentially) closed under structural recursion.*

Proof. First we define in $\mathcal{A}\&\mathcal{B}$ for each type σ two functions $FORTH^\sigma$ and $BACK^\sigma$:

$FORTH^\sigma : \sigma \to \{\iota\}$ computes the set of all elements of type ι that occur in a complex object of type σ:

$$\begin{array}{rcl} FORTH^\iota & := & \mathsf{sng} \\ FORTH^{unit} & := & \mathsf{emptyset} \\ FORTH^{\sigma\times\tau} & := & \mathsf{union} \circ FORTH^\sigma \times FORTH^\tau \\ FORTH^{\{\sigma\}} & := & \mathsf{ext}(FORTH^\sigma) \end{array}$$

$BACK^\sigma : \{\iota\} \to \{\sigma\}$ which, given a "basic" set of constants $B : \{\iota\}$, computes a set which is the finite "universe" of all complex objects of type σ that can be constructed using elements from B:

$$\begin{array}{rcl} BACK^\iota(B) & = & B \cup \{c_1, \ldots, c_n\} \\ BACK^{unit}(B) & = & \{()\} \\ BACK^{\sigma\times\tau} & := & \mathsf{cartprod} \circ \langle BACK^\sigma, BACK^\tau \rangle \\ BACK^{\{\sigma\}} & := & \mathsf{powerset} \circ BACK^\sigma \end{array}$$

We can verify these definitions by proving that

Claim I. For any complex object q of type σ, we have $q \in BACK^\sigma(FORTH^\sigma(q))$.

Claim II. Let $f : \sigma \to \tau$ be a function that can be expressed in $\mathcal{A}\&\mathcal{B}(C)$ plus structural recursion and let $B : \{\iota\}$ be a basic set of constants. For any complex object q of type σ, if $q \in BACK^\sigma(B)$ then $f(q) \in BACK^\tau(B)$.

The claims are proved by induction on complex objects and on function expressions in $\mathcal{A}\&\mathcal{B}(C)$. Details omitted.

Next, we show that $\mathcal{A}\&\mathcal{B}(C)$ is closed under definitions by structural recursion. Let $e : \tau$ and $i(x^\sigma, z^\tau) = e_i$ where $i : \sigma \times \tau \to \tau$ be expressible in $\mathcal{A}\&\mathcal{B}(C)$. Let $s^{\{\sigma\}}$ be a variable. We need an $\mathcal{A}\&\mathcal{B}(C)$ expression G such that $g := \mathsf{sr\_add}_{set}(i, e)$ can be expressed as $g(s^{\{\sigma\}}) = G$. In other words, G expresses the result of applying g to $s^{\{\sigma\}}$.

Think of $s^{\{\sigma\}}$ as expressing a complex object that g takes as argument. Then,

$$B := \mathit{FORTH}^{\{\sigma\}} s^{\{\sigma\}}$$

expresses the basic set of constants that occur in this argument. $\mathit{BACK}^\tau B$ then expresses the set of all complex objects of type τ that can be built with these constants. In view of Claim II, the result of applying g to $s^{\{\sigma\}}$ must be among these complex objects.

The next step is to express the binary relation of type $\{\{\sigma\} \times \tau\}$ that is the graph of g restricted to arguments that can be built out of the constants in B. The set

$$\mathsf{powerset}(\mathsf{cartprod}(\mathit{BACK}^{\{\sigma\}} B, \mathit{BACK}^\tau B))$$

consists of all binary relations between complex objects built from these constants. Out of this set, select only those relations R that contain $(\{\}, e)$ and which are such that

$$\mathsf{every}\ x^\sigma\ \mathsf{in}\ s^{\{\sigma\}}\ \mathsf{satisfies}\ \mathsf{subset}(R', R)$$

where $R' := \mathsf{map}(f)(R)$ and where $f(t^{\{\sigma\}}, z^\tau) := (x^\sigma \uparrow t^{\{\sigma\}}, e_i)$.

Clearly all this can be expressed in $\mathcal{A}\&\mathcal{B}$. We want the smallest relation among those selected, and this is achieved by taking their intersection (see subsection 5.1). Let I be the resulting expression. From the universality property that defines g it follows that I expresses the desired graph of g restricted to arguments that can be built out of the constants in B. Therefore, we select from I all pairs whose left component is $s^{\{\sigma\}}$ (there will be only one such pair since g is a function) and then take the second relational projection. The result is an expression of type $\{\tau\}$ which is semantically equivalent to $gs^{\{\sigma\}}$, modulo the small unpleasantness that instead of the desired result, it returns a singleton set containing the result. When τ is a set type or a product of set types, this can be remedied by further composing with relational projections and flattenings. The types of the overall translation must be adjusted to take care of this unpleasantness (this is the reason for the qualifier "essentially" in the theorem's statement,) but this is straightforward. $\square$

The point of this result is not a practical one, since the transformations it suggests are neither polymorphic nor efficient. In addition to formalizing certain intuitions about the flavor of these languages, we hope that we might be able in the future to use it to transfer theoretical results, for example complexity lower bounds, from $\mathcal{A}\&\mathcal{B}(C)$ to languages with structural recursion.

6 Optimization and equational theories

6.1 An equationally provable optimization

The equational theory that we exhibit for our query languages can be used to validate algebraic optimizations. Trinder [58] has already studied optimizations for languages like the ones we present in this paper. Using comprehension syntax, which is equivalent to Φ, he identifies *qualifier interchange* as an identity on comprehensions that generalizes the important optimization known as *selection promotion*. To illustrate the power of the equational theories in the appendices, we will show, similarly, that selection promotion is *provable* in these theories.

Let $R : \{\sigma\}$ and $S : \{\tau\}$ be two sets, and let $p : \sigma \times \tau \to \{unit\}$ be a predicate. We wish to show that when p really only tests the first component of its argument, that is, when $p(x, y)$ is equivalent to $p'x$ for an appropriate $p' : \sigma \to \{unit\}$ (for example we can take $\mathsf{fun}\ \ p(z) = p'\pi_1(z)$), then the selection

$$\mathsf{select}\ z\ \mathsf{from}\ \mathsf{cartprod}(R, S)\ \mathsf{where}\ pz$$

is provably equal to

$$\mathsf{cartprod}((\mathsf{select}\ x\ \mathsf{from}\ R\ \mathsf{where}\ p'x), S)$$

(this may reduce the size of one of the sets involved in the cartesian product, before the product is computed).

To make the calculations less obscure, we will "redefine" selection as

$$\mathsf{select}\ x\ \mathsf{from}\ S\ \mathsf{where}\ P := \Phi x{\in} S.\, \mathsf{if}\ P\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \{\}.$$

This is only a small variation, since we have pointed out in section 4 that at such types the conditional is in fact definable, and its definition would get us back to the original definition of selection.

Recall also that

$$\mathsf{cartprod}(R, S) := \Phi x{\in} R.\, \mathsf{pairwith}_2(x, S)$$

and

$$\mathsf{pairwith}_2(x, S) := \Phi y{\in} S.\, \{(x, y)\}$$

Expanding this syntactic sugar and applying some axioms for $\mathcal{M}$ (axiom 8 and axiom 7, appendix B) $\mathsf{select}\ z\ \mathsf{from}\ \mathsf{cartprod}(R, S)\ \mathsf{where}\ pz$ becomes

$$\Phi x{\in} R.\, (\Phi y{\in} S.\, (\mathsf{if}\ p(x, y)\ \mathsf{then}\ \{(x, y)\}\ \mathsf{else}\ \{\}))$$

Now replace $p(x, y)$ with $p'x$. To make further progress in simplifying the expression we need axioms for the extension $\mathcal{R}(=, \mathsf{cond})$ (namely axiom 6 and axiom 5, appendix C). Applying these, we get

$$\Phi x{\in} R.\, (\mathsf{if}\ p'x\ \mathsf{then}\ (\Phi y{\in} S.\, \{(x, y)\})\ \mathsf{else}\ \{\})$$

This is a sort of normal form[6], so we will try to reach the same by transforming equationally the other expression, namely $\mathsf{cartprod}((\mathsf{select}\ x\ \mathsf{from}\ R\ \mathsf{where}\ p'x), S)$.

[6] In fact, Wong [63, 64] organizes these equational theories as confluent and terminating rewrite systems.

Expanding the syntactic sugar we get

$$\Phi x'\in(\Phi x\in R.\,(\mathsf{if}\ p'x\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \{\}))\,.\,\mathsf{pairwith}_2(x',S)$$

Applying axiom 8, appendix B

$$\Phi x\in R.\,(\Phi x'\in(\mathsf{if}\ p'x\ \mathsf{then}\ \{x\}\ \mathsf{else}\ \{\})\,.\,\mathsf{pairwith}_2(x',S))$$

Again, to make further progress we need more axioms that are specific to $\mathcal{R}(=,\mathsf{cond})$ (axiom 3 and axiom 2, appendix C). Applying these as well as axiom 7 of appendix B we get the normal form that was also reached above, which completes the proof.

6.2 Naturality.

Beyond using the equational theory as a validation tool for optimizations, we hope that the category-theoretic foundations on which it is based could be used to "discover" useful optimizations. For example many algebraic optimizations take the form of commutations between constructs. One property known from category theory has typically the form of a commutation—naturality.

To see this, consider again complex object type expressions with type variables. We can interpret such type expressions with n type variables in them as functors $SET^n \to SET$. Then, taking closed polymorphic expressions in $\mathcal{A}\&\mathcal{B}$ (without eq, it turns out) we can show that their meanings for various sets assigned to the type variables are natural transformations. Moreover, the action on morphisms of the functors is expressible in the language. Hence the naturality can be expressed as a family of equations that hold between expressions. Finally, since the equational theory can prove the naturality of each construct separately, we know by general category-theoretic considerations that it will be able to prove the naturality equations for any expression in $\mathcal{A}\&\mathcal{B}$ without eq.

More precisely, consider a type expression θ without primitive types, and a list of distinct type variables α_1, ..., α_n that includes all the type variables in θ. We define a functor $\theta : SET^n \to SET$ associated to this type expression as follows. The action of θ on objects is given by

$$(\sigma_1,\ldots,\sigma_n) \mapsto \theta[\vec{\sigma}/\vec{\alpha}]$$

where $[\vec{\sigma}/\vec{\alpha}]$ is an abbreviation for $[\sigma_1/\alpha_1]\cdots[\sigma_n/\alpha_n]$. The action on morphisms can be defined entirely inside our language $\mathcal{M}$, as follows. Given $g_1 : \sigma_1 \to \tau_1,\ldots,g_n : \sigma_n \to \tau_n$, define a morphism $\theta(g_1,\ldots,g_n) : \theta[\vec{\sigma}/\vec{\alpha}] \to \theta[\vec{\tau}/\vec{\alpha}]$ by the following induction on θ:

$$\begin{array}{rcl}
\alpha_i(g_1,\ldots,g_n) & := & g_i \\
unit(\vec{g}) & := & \mathsf{id}_{unit} \\
(\theta_1 \times \theta_2)(\vec{g}) & := & \theta_1(\vec{g}) \times \theta_2(\vec{g}) \\
\{\theta\}(\vec{g}) & := & \mathsf{map}(\theta(\vec{g}))
\end{array}$$

where $\vec{g}$ is an abbreviation for $g_1,\ldots,g_n$. Now we have

Theorem 6.1 (Naturality) *Let $f : \theta_1 \to \theta_2$ be a polymorphic expression in $\mathcal{A}\&\mathcal{B}$ without* eq *such that the type variables of f, θ_1 and θ_2 are in the list α_1, ..., α_n For any $g_1 : \sigma_1 \to \tau_1, \ldots, g_n : \sigma_n \to \tau_n$, the equation*

$$f[\vec{\tau}/\vec{\alpha}] \circ \theta_1(\vec{g}) = \theta_2(\vec{g}) \circ f[\vec{\sigma}/\vec{\alpha}]$$

is true, and is in fact provable from the equational theory in appendix A, enriched with axioms that state the naturality of emptyset, union, cond, *and* powerset.

As mentioned before, once we notice that the equational theory of $\mathcal{M}$ proves the naturality of each of the language's constructs, this theorem holds on general category-theoretical principles.

It appears that this generalizes several identities used in algebraic optimizations, especially regarding commutations with projections. Indeed, recall the definition $\Pi_{1\,\alpha_1,\alpha_2} := \mathsf{map}(\mathsf{fst}_{\alpha_1,\alpha_2})$ and note that the action of type expression functors on morphisms similarly combines map and tuple manipulation.

A simple application of the theorem yields that for any $g_1 : \sigma_1 \to \tau_1$ and any $g_2 : \sigma_2 \to \tau_2$ we have

$$\Pi_{1\,\tau_1,\tau_2} \circ \mathsf{map}(g_1 \times g_2) = \mathsf{map}(g_1) \circ \Pi_{1\,\sigma_1,\sigma_2}.$$

In a more interesting application, let θ and $f : \theta \to \{\alpha\}$ have (for simplicity) just the type variable α. We can take $\mathsf{fst}_{\sigma,\tau} : \sigma \times \tau \to \sigma$ in the role of g and the theorem gives

$$\Pi_{1\,\sigma,\tau} \circ f[\sigma \times \tau/\alpha] = f[\sigma/\alpha] \circ \theta(\mathsf{fst}_{\sigma,\tau}).$$

It turns out that if the meanings of the g's are injective functions the we can deal with non-monotonic primitives such as equality and the theorem holds (semantically) for all of $\mathcal{A}\&\mathcal{B}$.

By taking the semantic statement and the g's to be bijections, we get that all the queries definable in $\mathcal{A}\&\mathcal{B}$ are generic or consistent [13]. Genericity with respect to additional primitive operations can also be shown by working with bijections that are homomorphisms for these operations. These results extend to structural recursion.

7 Recent Developments and Related Work

This paper is about the semantics of languages that derive from structural recursion over collection types. Considerable effort is needed to realize this semantics in the syntax of a practical programming language. Since the inception of this work in [7, 9], there have been two practical developments. The first is the implementation of practical languages: Shaharazade by Naqvi and his colleagues at Bellcore [50, 60]. Shahrazade has been used to model telecommunications operations support systems. One such prototype system allows planners and designers to manipulate models of Digital Loop Carrier systems with multiple choices of subcomponents, i.e., a parts explosion with functionally equivalent subparts. In another experimental prototype system Shahrazade and VEIL have been used to design a design manager for telecommunications equipment.

The second practical development is the the Collection Programming Language CPL, together with its programming environment Kleisli, implemented by Wong and Hart at the University of Pennsylvania.

This implementation uses optimizing transformations that derive directly from those presented in this paper. In addition there has been a substantial body of research on the expressive power of various forms of structural recursion, the complexity of languages based on structural recursion, and the investigation of structural recursion on other collection types (in this paper we have focussed on sets.)

7.1 The collection programming language CPL

The observation that the monad operations we have used in this paper can be used to interpret the syntax of *comprehensions* used in functional programming languages was first made by Wadler [61], and the connection with database query languages was shown by Trinder, Wadler and Watt [58, 59, 62]. Comprehension syntax, as it is realized in CPL, superficially resembles Zermelo-Fraenkel set notation, but there are important differences. For example the composition of binary relations R and S is expressed in CPL as

$$\{(x, z) \mid (\backslash x, \backslash y_1) \leftarrow R, (\backslash y_2, \backslash z) \leftarrow S, y_1 = y_2\}$$

The right hand part, or body, of the comprehension contains two syntactic forms: generators such as $(\backslash x, \backslash y_1) \leftarrow R$ and conditions such as $y_1 = y_2$. The general form of a generator is $p \leftarrow e$ where p is a *pattern* and e is an expression denoting some collection. Patterns serve to introduce variables. By matching a pattern to successive components of a collection, variables are bound to values. The explicit marking $\backslash x$ of a variable when it is introduced is important if one is to have a general form of pattern matching. For example the expression above is equivalent to $\{(x, z) \mid (\backslash x, \backslash y) \leftarrow R, (y, \backslash z) \leftarrow S\}$. Wadler observed that a comprehension of the form

$$\{e \mid \backslash x \leftarrow e', c_2, \ldots c_n\}$$

in which $c_2, \ldots, c_n$ are the remaining components – generators or conditions – of the body of the comprehension, may be expressed as

$$\Phi x \in e'. \{e \mid c_2, \ldots c_n\}$$

An extension of this idea to account for general patterns and conditions can be used to interpret comprehension syntax.

The language CPL exploits this to obtain a query that is based on the semantics given in this paper and that manipulates the collection types lists, bags and sets together with records and variants. A description of CPL is given in [10], from which the following example of an Employee type is taken.

```
{[Name : [FirstName : string,LastName : string],
  DNum : int,
  Status : <Regular : [Salary : int Extension : string],
            Consultant : [Day_Rate : int, Phone : string]>,
  Projects : { string }]}
```

In which $\mathsf{[FirstName}: \mathit{string}, \mathsf{LastName}: \mathit{string}\mathsf{]}$ is a record type and $\mathsf{<Regular} : \ldots, \mathsf{Consultant} : \ldots\mathsf{>}$ is a variant or "tagged union" type. Variants are well known in programming languages, but are often overlooked in data models, where their absence creates needless fragmentation of the database and confusion over null values. Variants can be conveniently used in pattern matching:

$$
\begin{array}{l}
\{\, \mathsf{[Name} = n,\ \mathsf{Phone} = t\mathsf{]} \mid \\
\quad \mathsf{[Name} = \backslash n, \\
\quad\ \ \mathsf{Status} = \mathsf{<Consultant} = \\
\quad\quad\quad \mathsf{[Phone} = \backslash t, \ldots\mathsf{]>}, \ldots\mathsf{]} <\!\!- \mathsf{Emp}\}
\end{array}
$$

$$
\begin{array}{l}
\{\, \mathsf{[Name} = n, \mathsf{NoProj} = \mathit{count}(\mathit{SetToBag}(p))\mathsf{]} \mid \\
\quad \mathsf{[Name} = \backslash n, \\
\quad\ \ \mathsf{Status} = \mathsf{<Regular} = \backslash r\mathsf{>}, \\
\quad\ \ \mathsf{Projects} = \backslash p, \ldots\mathsf{]} <\!\!- \mathsf{Emp}\}
\end{array}
$$

The first query finds the names and telephone numbers of all consultants, because the pattern in this comprehension only matches consultants. The second query returns, for each regular employee, the name and number of projects to which that employee is assigned. CPL allows the expression of such queries. It also allows function definitions and the use of the more general forms of structural recursion we have described in this paper.

There is an obvious similarity between comprehension syntax and well-known languages such as SQL with some form of select ... from ... where. The authors believe that the ideas in this paper may be a better starting point than relational algebra for the practical implementation of such query languages for a number of reasons.

1. The need, that we have already mentioned, to extend query languages to new collection types and to allow their use on nested collections.
2. The ability to incorporate variants and to give a clean interpretation of pattern matching.
3. The ability to *construct* data structures that are as complicated as those being analyzed.
4. The ability to implement functions or incorporate external functions in a systematic fashion. Few implement query languages allow function definition. We believe a functional account of database query languages is important here.

At the university of Pennsylvania interfaces have been constructed between CPL/Kleislli and several biological databases that are part of the Human Genome Project [64, 25, 24]. This language has provided biological researchers with a simple language for querying and integrating a number of biological data sources, something that could not be performed by existing query languages. These sources not only include standard (relational) databases, but also include data in a number of data exchange formats. One of the data sources is expressed in ASN.1 a format that can describe sets, lists, variants and records, and arbitrary combinations of these types (points 1 and 2 above.) A frequent requirement is for data to be restructured to a complex format that makes it suitable for input to, say, a user interface (point 3). Also, much data is contained in special-purpose software such as sequence-matching programs, that implement external functions (point 4).

7.2 Further results on structural recursion and collection types

Since the appearance of the papers on which this work was based [7, 9], a substantial body of related research has appeared. Following the conservative extension result of [63], [52] shows that by adding a bounded fixed-point construct to $\mathcal{R}(=, \mathsf{cond})$ gives us, at relational types, inflationary datalog. In [34, 35] it is shown that nesting at intermediate types does not add expressiveness in presence of aggregate functions and certain generic queries. Other results on expressive power are to be found in [34, 36, 35]. Our approach can be used for different collections: languages for or-sets were studied in [33, 23, 38] and bag languages in [37]. As mentioned before, [54] shows that transitive closure, which is efficiently expressible using structural recursion, has a necessarily exponential implementation in complex-object algebra [1]. [30] show how to encode related database languages in the simply-typed lambda-calculus. The possibility of treating arrays as collection types is suggested in [12]. Connections with parallel complexity classes are studied in [53]. [56] shows that, in the presence of suitable external functions, $\mathsf{sr\_add}_{set}$ is strictly more expressive that $\mathsf{sr\_comb}_{set}$. [55] studies foundational issues concerning complex objects with queries over external functions.

Acknowledgements. The authors thank Foto Afrati, Dirk Van Gucht, Leonid Libkin, Hermann Puhlmann, Jon Riecke, Dan Suciu, and Steve Vickers for helpful discussions and Paul Taylor for his diagram macros. Tannen was partially supported by grants ONR NOOO-14-88-K-0634, NSF CCR-90-57570, and ARO DAALO3-89-C-0031-PRIME. Buneman was partially supported by grants ONR NOOO-14-88-K-0634 and NSF IRI-86-10617, and by a UK SERC visiting fellowship at Imperial College, London. Wong was partially supported by grants NSF IRI-90-04137 and ARO DAALO3-89-C-0031-PRIME.

References

[1] S. Abiteboul, C. Beeri, On the power of languages for the manipulation of complex objects, *in* "Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects," Darmstadt, 1988.

[2] S. Abiteboul and G. Hillebrand Space usage in functional query languages, *in* "LNCS 893: Proceedings of 5th International Conference on Database Theory," 439–454, Prague, Czech Republic, January 1995.

[3] S. Abiteboul, V. Vianu, Fixpoint extensions of first-order logic and Datalog-like languages, *in* "Proceedings of 4th IEEE Symposium on Logic in Computer Science," 71–79, Pacific Grove, California, June 1989.

[4] A. V. Aho, J. D. Ullman, Universality of data retrieval languages, *in* "Proceedings 6th Symposium on Principles of Programming Languages," 110-120, Texas, January 1979.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, P. Valduriez, A powerful and simple database language, *in* "Proceedings of 14th International Conference on Very Large Data Bases," 97–105, 1988.

[6] H. Barendregt, "The Lambda Calculus: Its Syntax and Semantics," Elsevier, 1984.

[7] V. Breazu-Tannen, P. Buneman, S. Naqvi, Structural recursion as a query language, *in* "Proceedings of 3rd International Workshop on Database Programming Languages," 9–19, Naphlion, Greece, August 1991.

[8] V. Breazu-Tannen, R. Subrahmanyam, Logical and computational aspects of programming with Sets/Bags/Lists, *in* "LNCS 510: Proceedings of 18th International Colloquium on Automata, Languages, and Programming," 60–75, Madrid, Spain, July 1991.

[9] V. Breazu-Tannen, P. Buneman, L. Wong, Naturally embedded query languages, *in* "LNCS 646: Proceedings of 4th International Conference on Database Theory," 140–154, Berlin, Germany, October 1992.

[10] P. Buneman, L. Libkin, D. Suciu, V. Tannen, L. Wong, Comprehension syntax, *SIGMOD Record* **23** (1994), 87–96.

[11] P. Buneman, R. E. Frankel, R. Nikhil, An implementation technique for database query languages, *ACM Transactions on Database Systems* **7**, No. 2 (1982), 164–187.

[12] P. Buneman, The Fast Fourier Transform as a Database Query. Technical Report MS-CIS-93-37/L&C 60, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, March 1993.

[13] A. Chandra, D. Harel, Computable queries for relational databases, *Journal of Computer and System Sciences* **21**, No. 2 (1980), 156–178.

[14] A. Chandra, D. Harel, Structure and complexity of relational queries, *Journal of Computer and System Sciences* **25** (1982), 99–128.

[15] E. F. Codd, A relational model for large shared databanks, *Communications of the ACM* **13**, No. 6 (1970), 377–387.

[16] L. S. Colby, A recursive algebra for nested relations, *Information Systems* **15**, No. 5 (1990), 567–582.

[17] L. Fegaras, Efficient optimization of iterative queries, *in* "Proceedings of 4th International Workshop on Database Programming Languages," 200–225, New York, August 1993.

[18] L. Fegaras, D. Maier, Towards an effective calculus for object query languages, *in* "Proceedings of ACM SIGMOD Conference on Management of Data," San Jose, May 1995. To appear.

[19] J.A. Goguen, J.W.Thatcher, E.G. Wagner, An initial algebra approach to the specification, correctness and implementation of abstract data types, *in* "Current Trends in Programming Methodology," 80-149, Prentice Hall, 1978.

[20] J. Goguen, J. Meseguer, Completeness of many-sorted equational logic, *Houston Journal of Mathematics* **11**, No. 3 (1985), 307–334.

[21] M. Gyssens, D. Van Gucht, A comparison between algebraic query languages for flat and nested databases, *Theoretical Computer Science* **87**, 263–286, 1991.

[22] M. GYSSENS AND D. VAN GUCHT, The Powerset Algebra as a natural tool to handle nested database relations, *Journal of Computer and System Sciences*, **45**, 76–103, 1992

[23] E. GUNTER AND L. LIBKIN, OR-SML: A functional database programming language for disjunctive information, *in* "Proceedings of Conference on Database and Expert Systems Applications," Athens, Greece, September 1994.

[24] K. HART, L. WONG, A query interface for heterogeneous biological data sources, Manuscript, February 1994. Available on WWW via `ftp://www.cis.upenn.edu/ pub/ papers/ db-research/ kleisli.ps.Z`.

[25] K. HART, L. WONG, C. OVERTON, P. BUNEMAN, Using a query language to integrate biological data, *in* "Abstracts of Meeting on the Interconnection of Molecular Biology Databases," Stanford, August 1994. Available on WWW via `http://www.cis.upenn.edu/ ~cbil/ mimbd94/ mimbd94CPL.html`.

[26] T. IMIELINSKI, S. NAQVI, K. VADAPARTY, Querying design and planning databases, *in* "LNCS 566: Proceedings of 2nd International Conference on Deductive and Object Oriented Databases," 524–545, Munich, Germany, December 1991.

[27] N. IMMERMAN, S. PATNAIK, D. STEMPLE, The expressiveness of a family of finite set languages, *in* "Proceedings of 10th ACM Symposium on Principles of Database Systems," 37–52, May 1991.

[28] ISO, Standard 9075, Information Processing Systems, Database Language SQL, 1987.

[29] G. JAESCHKE, H. J. SCHEK, Remarks on the algebra of non-first-normal-form relations, *in* "Proceedings ACM Symposium on Principles of Database Systems," 124–138, Los Angeles, California, March 1982.

[30] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson. Database query languages embedded in the typed lambda calculus. *in* "Proceedings of 8th IEEE Symposium on Logic in Computer Science," 332–343, Montreal, Canada, June 1993.

[31] J. LAMBEK, P. J. SCOTT, "Introduction to Higher Order Categorical Logic," Cambridge University Press, London, 1986.

[32] L. LIBKIN, "Aspects of Partial Information in Databases," PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994.

[33] L. LIBKIN, L. WONG, Semantic representations and query languages for or-sets, *in* "Proceedings of 12th ACM Symposium on Principles of Database Systems," 37–48, Washington, D. C., May 1993.

[34] L. LIBKIN, L. WONG, Aggregate functions, conservative extension, and linear orders, *in* "Proceedings of 4th International Workshop on Database Programming Languages," 282–294, Manhattan, New York, August 1993.

[35] L. LIBKIN, L. WONG, Conservativity of nested relational calculi with internal generic functions, *Information Processing Letters* **49** (1994), 273–280.

[36] L. Libkin, L. Wong, New techniques for studying set languages, bag languages, and aggregate functions, *in* "Proceedings of 13th ACM Symposium on Principles of Database Systems," 155–166, Minneapolis, Minnesota, May 1994.

[37] L. Libkin, L. Wong, Some properties of query languages for bags, *in* "Proceedings of 4th International Workshop on Database Programming Languages," 97–114, Manhattan, New York, August 1993.

[38] L. Libkin, Approximation in databases, *in* "LNCS 893: Proceedings of 5th International Conference on Database Theory," 411–424, Prague, Czech Republic, January 1995.

[39] S. MacLane, "Categories for the Working Mathematician," Springer-Verlag, Berlin, 1971.

[40] I. A. Macleod, A database management system for document retrieval applications, *Information Systems* **6**, No. 2 (1981).

[41] A. Makinouchi, A consideration on normal form of not necessarily normalised relation in the relational data model, *in* "Proceedings of 3rd International Conference on Very Large Databases," 447–453, Tokyo, Japan, October 1977.

[42] E. G. Manes, "Algebraic Theories," Springer-Verlag, Berlin, 1976.

[43] L. Meertens, Algorithmics—towards programming as a mathematical activity, *in* "Proceedings of CWI Symposium on Mathematics and Computer Science," 289–334, North-Holland, 1986.

[44] A. R. Meyer, J. C. Mitchell, E. Moggi, R. Statman, Empty types in polymorphic λ-calculus, *in* "Proceedings of the 14th Symposium on Principles of Programming Languages," 253–262. January 1987.

[45] R. Milner, M. Tofte, R. Harper, "The Definition of Standard ML," MIT Press, 1990.

[46] E. Moggi, Notions of computation and monads, *Information and Computation* **93**(1991), 55–92.

[47] S. Naqvi, S. Tsur, "A Logical Language for Data and Knowledge Bases," Computer Science Press, 1989.

[48] A. Ohori, P. Buneman, V. Breazu-Tannen, Database programming in Machiavelli, a polymorphic language with static type inference, *in* "Proceedings of ACM-SIGMOD International Conference on Management of Data," 46–57, Portland, Oregon, June 1989.

[49] J. Paredaens, D. Van Gucht, Converting nested relational algebra expressions into flat algebra expressions, *ACM Transaction on Database Systems* **17**, No. 1 (1992), 65–93.

[50] Y. Saraiya, Optimizing functional query languages, *in* "Proceedings of Conference on Information and Knowledge Management," Washington, D. C., 1993.

[51] H.-J. Schek, M. H. Scholl, The relational model with relation-valued attributes, *Information Systems* **11**, No. 2 (1989), 137–147.

[52] D. Suciu, Fixpoints and bounded fixpoints for complex objects, *in* "Proceedings of 4th International Workshop on Database Programming Languages," 263–281, Manhattan, New York, August 1993.

[53] D. Suciu, V. Tannen, A query language for NC, *in* "Proceedings of 13th ACM Symposium on Principles of Database Systems," 167–178, Minneapolis, Minnesota, May 1994.

[54] D. Suciu, J. Paredaens, Any algorithm in the complex object algebra with powerset needs exponential space to compute transitive closure, *in* "Proceedings of 13th ACM Symposium on Principles of Database Systems," 201–209, Minneapolis, May 1994.

[55] D. Suciu, Domain-independent queries on databases with external functions, *in* "LNCS 893: Proceedings of 5th International Conference on Database Theory," 177–190, Prague, Czech Republic, January 1995.

[56] D. Suciu, L. Wong, On two forms of structural recursion, *in* "LNCS 893: Proceedings of 5th International Conference on Database Theory," 111–124, Prague, Czech Republic, January 1995.

[57] S. J. Thomas, P. C. Fischer, Nested relational structures, *in* "Advances in Computing Research: The Theory of Databases," 269–307, JAI Press, 1986.

[58] P. W. Trinder, Comprehensions, a query notation for DBPLs, *in* "Proceedings of 3rd International Workshop on Database Programming Languages," 49–62, Nahplion, Greece, August 1991.

[59] P. W. Trinder, P. L. Wadler, List comprehensions and the relational calculus, *in* "Proceedings of 1988 Glasgow Workshop on Functional Programming," 115–123, Rothesay, Scotland, August 1988.

[60] G. Venkatesh, M. Kohli, Shahrazade: A database query language for design and planning applications, Bellcore technical memorandum. Bellcore, Box 1910, Morristown, NJ 07960

[61] P. L. Wadler, Comprehending monads, *Mathematical Structures in Computer Science* **2**(1992), 461–493.

[62] D. A. Watt, P. W. Trinder, Towards a Theory of Bulk Types, Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

[63] L. Wong, Normal forms and conservative properties for query languages over collection types, *in* "Proceedings of 12th ACM Symposium on Principles of Database Systems," 26–36, Washington, D. C., May 1993.

[64] L. Wong, "Querying Nested Collections," PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994.

APPENDICES

A Monads and equational axiomatization for the algebra of functions

Saying that for each x $(\mathsf{coll}(\sigma), \mathsf{comb}(\cdot,\cdot), \mathsf{sng}(x), \mathsf{empty})$ is initial in the appropriate class of algebras is the same as saying that $(\mathsf{coll}(\sigma), \mathsf{comb}(\cdot,\cdot), \mathsf{empty})$ is the free (commutative(-idempotent)) monoid on (generated by) σ via $\mathsf{sng}(\cdot)$. The category-theoretic terminology for this is that the forgetful functor from the category of (commutative(-idempotent)) monoids to the category of sets and functions has a left adjoint, and $\mathsf{sng}(\cdot)$ is the unit of the adjunction.

As with any adjunction, by composing the functor that gives the free (commutative(-idempotent)) monoid with the forgetful functor we get a monad on the category of sets and functions. This is how the (finite) list, bag and set monads arise out of the adjunctions that are the basis for structural recursion. There are several equivalent formulations for monads (see [42] where monads are called algebraic theories) and one of these has $\mathsf{ext}(\cdot)$ as its salient operation (together with singleton). (monads in "extension" form). Another formulation [39] is based on $\mathsf{map}(\mathrm{cot})$, $\mathsf{flatten}$, and sng. We give below the commutative diagrams that describe this last formulation, and hence we give an equational axiomatization for our algebra of functions. From this perspective, only the essential axioms are listed below. The reflexivity axiom, as well as the symmetry, transitivity, and congruence inference rules have been omitted,

First, the axioms that make the algebra of functions a category.

1. $\dfrac{f : \sigma \to \sigma' \quad g : \sigma' \to \tau' \quad h : \tau' \to \tau}{h \circ (g \circ f) = (h \circ g) \circ f : \sigma \to \tau}$

2. $\dfrac{f : \sigma \to \tau}{\mathsf{id} \circ f = f : \sigma \to \tau}$

3. $\dfrac{f : \sigma \to \tau}{f \circ \mathsf{id} = f : \sigma \to \tau}$

Next, the axioms that make $\times$ a binary products and *unit* a terminal object (hence we have all finite products).

4. $$\frac{f : \nu \to \sigma \times \tau}{\langle \mathsf{fst} \circ f,\ \mathsf{snd} \circ f \rangle = f : \nu \to \sigma \times \tau}$$

5. $$\frac{f : \nu \to \sigma \quad g : \nu \to \tau}{\mathsf{fst} \circ \langle f, g \rangle = f : \nu \to \tau}$$

6. $$\frac{f : \nu \to \sigma \quad g : \nu \to \tau}{\mathsf{snd} \circ \langle f, g \rangle = g : \nu \to \tau}$$

7. $$\frac{f : \sigma \to \mathit{unit}}{f = \ \mathsf{ter} : \sigma \to \mathit{unit}}$$

$$\begin{tikzcd}
 & \nu \arrow[dl, "f"'] \arrow[dr, "g"] \arrow[dd, "\langle f, g \rangle"] & \\
\sigma & & \tau \\
 & \sigma \times \tau \arrow[ul, "\mathsf{fst}"] \arrow[ur, "\mathsf{snd}"'] &
\end{tikzcd}$$

Then, the axioms that say that we have a functor whose action on objects is $s \mapsto \{s\}$ and on morphisms, $f \mapsto \mathsf{map}(f)$. Moreover, the axioms that say that sng and $\mathsf{flatten}$ are natural transformations, and that the functor $s \mapsto \{s\}$, together with these natural transformations is a monad.

8. $$\frac{f : \nu \to \sigma \quad g : \sigma \to \tau}{\mathsf{map}(g \circ f) = \mathsf{map}(g) \circ \mathsf{map}(f) : \{\nu\} \to \{\tau\}}$$

$$\begin{tikzcd}
\{\nu\} \arrow[r, "map(f)"] \arrow[dr, "map(g \circ f)"'] & \{\sigma\} \arrow[d, "map(g)"] \\
 & \{\tau\}
\end{tikzcd}$$

9. $$\frac{}{\mathsf{map}(\mathsf{id}) = \mathsf{id} : \{\sigma\} \to \{\sigma\}}$$

10. $$\frac{f : \sigma \to \tau}{\mathsf{map}(f) \circ \mathsf{sng} = \mathsf{sng} \circ f : \sigma \to \{\tau\}}$$

$$\begin{tikzcd}
\sigma \arrow[r, "f"] \arrow[d, "\mathsf{sng}"'] & \tau \arrow[d, "\mathsf{sng}"] \\
\{\sigma\} \arrow[r, "\mathsf{map}(f)"'] & \{\tau\}
\end{tikzcd}$$

11. $$\frac{f : \sigma \to \tau}{\mathsf{map}(f) \circ \mathsf{flatten} = \mathsf{flatten} \circ \mathsf{map}(\mathsf{map}(f)) : \{\{\sigma\}\} \to \{\tau\}}$$

$$\begin{tikzcd}
\{\{\sigma\}\} \arrow[r, "\mathsf{flatten}"] \arrow[d, "\mathsf{map}(\mathsf{map}(f))"'] & \{\sigma\} \arrow[d, "\mathsf{map}(f)"] \\
\{\{\tau\}\} \arrow[r, "\mathsf{flatten}"'] & \{\tau\}
\end{tikzcd}$$

12. $\dfrac{}{\mathsf{flatten} \circ \mathsf{sng} = \mathsf{id} : \{\sigma\} \to \{\sigma\}}$

13. $\dfrac{}{\mathsf{flatten} \circ \mathsf{map}(\mathsf{sng}) = \mathsf{id} : \{\sigma\} \to \{\sigma\}}$

$$\begin{tikzcd}
& \{\{\sigma\}\} \arrow[dd, "\mathsf{flatten}"] & \\
\{\sigma\} \arrow[ur, "\mathsf{sng}"] \arrow[dr, "\mathsf{id}"'] & & \{\sigma\} \arrow[ul, "\mathsf{map}(\mathsf{sng})"'] \arrow[dl, "\mathsf{id}"] \\
& \{\sigma\} &
\end{tikzcd}$$

14. $\dfrac{}{\mathsf{flatten} \circ \mathsf{map}(\mathsf{flatten}) = \mathsf{flatten} \circ \mathsf{flatten} : \{\{\{\sigma\}\}\} \to \{\sigma\}}$

$$\begin{tikzcd}
\{\{\{\sigma\}\}\} \arrow[r, "\mathsf{map}(\mathsf{flatten})"] \arrow[d, "\mathsf{flatten}"'] & \{\{\sigma\}\} \arrow[d, "\mathsf{flatten}"] \\
\{\{\sigma\}\} \arrow[r, "\mathsf{flatten}"'] & \{\sigma\}
\end{tikzcd}$$

Finally, the axioms that make the monad into a strong monad via the natural transformation $\mathsf{pairwith}_2$ [46].

15. $\dfrac{}{\mathsf{map}(\mathsf{snd}) \circ \mathsf{pairwith}_2 = \mathsf{snd} : \mathit{unit} \times \{\sigma\} \to \{\sigma\}}$

$$\begin{tikzcd}
\mathit{unit} \times \{\sigma\} \arrow[r, "\mathsf{pairwith}_2"] \arrow[d, "\mathsf{snd}"'] & \{\mathit{unit} \times \sigma\} \arrow[dl, "\mathsf{map}(\mathsf{snd})"] \\
\{\sigma\} &
\end{tikzcd}$$

16. $\dfrac{}{\mathsf{pairwith}_2 \circ (\mathsf{id} \times \mathsf{sng}) = \mathsf{sng} : \sigma \times \tau \to \{\sigma \times \tau\}}$

17. $\dfrac{}{\mathsf{pairwith}_2 \circ (\mathsf{id} \times \mathsf{flatten}) = \mathsf{flatten} \circ \mathsf{map}(\mathsf{pairwith}_2) \circ \mathsf{pairwith}_2 : \sigma \times \{\{\tau\}\} \to \{\sigma \times \tau\}}$

$$\begin{tikzcd}
\{\sigma \times \{\tau\}\} \arrow[r, "\mathsf{map}(\mathsf{pairwith}_2)"] & \{\{\sigma \times \tau\}\} \arrow[r, "\mathsf{flatten}"] & \{\sigma \times \tau\} \\
\sigma \times \{\{\tau\}\} \arrow[u, "\mathsf{pairwith}_2"] \arrow[r, "(\mathsf{id} \times \mathsf{flatten})"'] & \sigma \times \{\tau\} \arrow[ur, "\mathsf{pairwith}_2"] & \sigma \times \tau \arrow[l, "(\mathsf{id} \times \mathsf{sng})"] \arrow[u, "\mathsf{sng}"']
\end{tikzcd}$$

18. $\dfrac{}{\mathsf{map}(i) \circ \mathsf{pairwith}_2 = \mathsf{pairwith}_2 \circ (\mathsf{id} \times \mathsf{pairwith}_2) \circ i : (\nu \times \sigma) \times \{\tau\} \to \{\nu \times (\sigma \times \tau)\}}$

where i is $\langle \mathsf{fst} \circ \mathsf{fst},\ \langle \mathsf{snd} \circ \mathsf{fst},\ \mathsf{snd} \rangle \rangle$.

$$\begin{tikzcd}
(\nu \times \sigma) \times \{\tau\} \arrow[rr, "\mathsf{pairwith}_2"] \arrow[d, "i"'] & & \{(\nu \times \sigma) \times \tau\} \arrow[d, "\mathsf{map}(i)"] \\
\nu \times (\sigma \times \{\tau\}) \arrow[r, "(\mathsf{id} \times \mathsf{pairwith}_2)"'] & \nu \times \{\sigma \times \tau\} \arrow[r, "\mathsf{pairwith}_2"'] & \{\nu \times (\sigma \times \tau)\}
\end{tikzcd}$$

19. $$\frac{f : \sigma \to \sigma' \quad g : \tau \to \tau'}{\mathsf{map}(f \times g) \circ \mathsf{pairwith}_2 = \mathsf{pairwith}_2 \circ (f \times \mathsf{map}(g)) : \sigma \times \{\tau\} \to \{\sigma' \times \tau'\}}$$

$$\begin{tikzcd}
\sigma \times \{\tau\} \arrow[r, "\mathsf{pairwith}_2"] \arrow[d, "(f \times \mathsf{map}(g))"'] & \{\sigma \times \tau\} \arrow[d, "\mathsf{map}(f \times g)"] \\
\sigma' \times \{\tau'\} \arrow[r, "\mathsf{pairwith}_2"'] & \{\sigma' \times \tau'\}
\end{tikzcd}$$

B Axioms for the complex object calculus

We present here an equational axiomatization for the core complex object calculus which follows immediately from Manes' axioms for monads in extension form [42]. A similar axiomatization is used by Moggi [46].

In order to allow reasoning that is sound in models with empty types (in our case, this would occur iff some base type is empty) we tag equations with sequences of distinct variables [20, 31, 44]: $e_1 =_\Gamma e_2$ where all the free variables of e_1 and e_2 are included in Γ.

The reflexivity axiom, as well as the symmetry, transitivity, and congruence inference rules have been omitted, except for congruence with respect to the Φ construct, which is analogous to the ξ rule in the lambda calculus [6]:

1. $$\frac{S_1 =_\Gamma S_2 : \{\sigma\} \quad T_1 =_{\Gamma, x^\sigma} T_2 : \{\tau\}}{\Phi x^\sigma \!\in\! S_1 . T_1 =_\Gamma \Phi x^\sigma \!\in\! S_2 . T_2 : \{\tau\}}$$

The axioms for binary and 0-ary products:

2. $$\frac{e_1 : \sigma \quad e_2 : \tau}{\pi_1(e_1, e_2) =_\Gamma e_1 : \sigma}$$

3. $$\frac{e_1 : \sigma \quad e_2 : \tau}{\pi_2(e_1, e_2) =_\Gamma e_2 : \tau}$$

4. $$\frac{e : \sigma \times \tau}{(\pi_1(e),\ \pi_2(e)) =_\Gamma e : \sigma \times \tau}$$

5. $$\frac{e : \mathit{unit}}{e =_\Gamma () : \mathit{unit}}$$

The axioms for monads in "extension" form:

6. $$\frac{S : \{\sigma\}}{\Phi x^\sigma{\in} S.\, \{x^\sigma\} =_\Gamma S : \{\sigma\}}$$

7. $$\frac{e : \sigma \quad T : \{\tau\}}{\Phi x^\sigma{\in}\{e\}.\, T =_\Gamma T[e/x^\sigma] : \{\tau\}}$$

8. $$\frac{R : \{\nu\} \quad S : \{\sigma\} \quad T : \{\tau\}}{\begin{array}{l}\Phi x^\nu{\in}(\Phi y^\sigma{\in} S.\, R).\, T \\ =_\Gamma \Phi y^\sigma{\in} S.\, (\Phi x^\nu{\in} R.\, T) : \{\tau\}\end{array}}$$

We can now state a result that relates the translations in section 3 with this equational theory and the one in appendix A. We need one more notation; define a "tupling" transformation that constructs from each each sequence of distinct variables Γ an expression in the calculus:

$$\begin{array}{rcl} \mathit{tuple}(\emptyset) & := & () \\ \mathit{tuple}(\Gamma, x^\sigma) & := & (\mathit{tuple}(\Gamma), x^\sigma) \end{array}$$

Theorem B.1 *1. For any pair $\Gamma \triangleright e$, letting $z^{\mathit{type}(\Gamma)} \triangleright e' := \mathcal{C}[\mathcal{A}[\Gamma \triangleright e]]$, the equation $e =_\Gamma e'[\mathit{tuple}(\Gamma)/z]$ is provable from the axioms and rules above.*

2. For any function $f : \sigma \to \tau$ in the algebra, letting $f' := \mathcal{A}[\mathcal{C}[f]] : \mathit{unit} \times \sigma \to \tau$, the equation $f = f' \circ \langle \mathsf{ter}_\sigma, \mathsf{id}_\sigma \rangle$ is provable from the axioms and rules in appendix A.

3. $e_1 =_\Gamma e_2$ is provable from the axioms and rules above iff $\mathcal{A}[\Gamma \triangleright e_1] = \mathcal{A}[\Gamma \triangleright e_2]$ is provable from the axioms and rules in appendix A.

4. $f_1 = f_2 : \sigma \to \tau$ is provable from the axioms and rules in appendix A iff $e_1 =_{x^\sigma} e_2$ is provable from the axioms and rules above, where $x^\sigma \triangleright e_i := \mathcal{C}[f_i],\ i = 1, 2$.

The proofs of parts 1 and 2 are straightforward inductions on expressions. The "only if" sides of parts 3 and 4 are proved straightforwardly by induction on equational proof trees. Using parts 1 and 2, the "if" sides of parts 3 and 4 then follow. The details are omitted.

C Some additional axioms for extensions of $\mathcal{M}$

We offer some equalities relating Φ with emptyset, union and the conditionals whose main merit is to be ... true. The first two axioms have been given by Wadler for his ringads [58], and they seem to express fundamental properties. The third axiom would probably follow from the cleaner representation of booleans as $unit + unit$ (but we have deliberately ignored sums in this paper). The status of the last three axioms is unclear. It is quite possible that they are derivable from the rest. Finally, we do not have a reasonable axiomatization of equality.

1. $\Phi x \in S_1 \cup S_2 . T \;=\; \Phi x \in S_1 . T \;\cup\; \Phi x \in S_2 . T$.
2. $\Phi x \in \{\} . T \;=\; \{\}$.
3. $\Phi x \in (\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2) . T = \mathsf{if}\ B\ \mathsf{then}\ (\Phi x \in S_1 . T)\ \mathsf{else}\ (\Phi x \in S_2 . T)$.
4. $\Phi x \in S . T_1 \cup T_2 \;=\; \Phi x \in S . T_1 \;\cup\; \Phi x \in S . T_2$.
5. $\Phi x \in S . \{\} \;=\; \{\}$.
6. $\Phi x \in S . (\mathsf{if}\ B\ \mathsf{then}\ T_1\ \mathsf{else}\ T_2) = \mathsf{if}\ B\ \mathsf{then}\ (\Phi x \in S . T_1)\ \mathsf{else}\ (\Phi x \in S . T_2)$ where x is not free in B.