

# Naturally Embedded Query Languages

Val Breazu-Tannen, Peter Buneman and Limsoon Wong\*

25 June 1992

Department of Computer and Information Science  
University of Pennsylvania  
Philadelphia, PA 19104-6389, USA

## Abstract

We investigate the properties of a simple programming language whose main computational engine is *structural recursion* on sets. We describe a progression of sublanguages in this paradigm that (1) have increasing expressive power, and (2) illustrate robust conceptual restrictions thus exhibiting interesting additional properties. These properties suggest that we consider our sublanguages as candidates for “query languages”. Viewing query languages as restrictions of our more general programming language has several advantages. First, there is no “impedance mismatch” problem; the query languages are already there, so they share common semantic foundation with the general language. Second, we suggest a uniform characterization of nested relational and complex-object algebras in terms of some surprisingly simple operators; and we can make comparisons of expressiveness in a general framework. Third, we exhibit differences in expressive power that are not always based on complexity arguments, but use the idea that a query in one language may not be *polymorphically* expressible in another. Fourth, ideas of category theory can be profitably used to organize semantics and syntax, in particular our minimal (core) language is a well-understood categorical construction: a cartesian category with a strong monad on it. Finally, we bring out an *algebraic* perspective, that is, our languages come with equational theories, and categorical ideas can be used to derive a number of rather general identities that may serve as optimizations or as techniques for discovering optimizations.

## 1 Introduction

We have recently proposed [4] a programming language for manipulating complex object databases, whose fundamental computational construct is based on *structural recursion on sets*. Its main virtue is that while it maintains most of the flexibility of a general purpose programming language, it can be explained with much economy of concepts [6]. Consequently, the language scales from flat relations up effortlessly, and holds good promise for optimizations.

As part of a larger program for investigating the properties of this programming paradigm, we concentrate here on identifying various sublanguages that arise as conceptually robust restrictions of the general paradigm. Such sublanguages are “query languages” because they may have simpler semantic complexity, because they may be more easily optimizable, because these optimizations may be easier to discover, or because they may have simple and natural syntactic representation *within* the language. But most importantly,

---

\*The authors were partially supported by grants ONR N000-14-88-K-0634, NSF CCR-90-57570, NSF IRI-86-10617 and ARO DAAL03-89-C-0031PRIME. Peter Buneman was also supported by a SERC visiting research fellowship at Imperial College, London.

“embedding” these sublanguages in the general language does not require semantical changes or worrying about compatibility, because these constructs are already definable! They are nothing more than syntactic sugar. Hence “impedance mismatch” is not an issue, and “seamless integration” comes for free.

The idea of the passage from flat relations to nested relations/complex objects is to let product and set constructions combine in every possible way, an *orthogonality* principle. However, in the choice of fundamental operations for manipulating these data structures, most previous work has concentrated on *extensions* of well understood languages such as relational algebra and datalog, even though it is not clear that the concepts that can be justified as *basic* in the case of flat relations, remain basic for nested relations. Moreover, the extensions made, especially in the study of nested relational algebra have a somewhat ad-hoc flavor; it is not clear which operations are really needed and for what.

In contrast, we have found it rewarding to use an orthogonality principle not only in the description of the structure of the data but also in the choice of fundamental operations that manipulate the data. The result is that a small number of semantic ideas are sufficient for explaining and analyzing the expressive power of remarkably rich languages. An equally important principle is that we should think of the various languages as parameterized by a *primitive signature* – a collection of types, constants and functions whose interpretation is external to the languages we shall describe; and many of the considerations we make are independent of this signature.

We have also found it very profitable to use some basic ideas of category theory in organizing the semantics and even the syntax of our languages. In particular, structural recursion comes out of certain adjunctions, and the core of the sublanguages we consider is based on monads. Because of its finitary character the semantics can be often “internalized” and we can use typical categorical ideas to derive useful syntactic properties such as identities that can serve for optimizations.

Thus, to the traditional complexity-theoretical and logical perspectives in query languages we want to add an *algebraic* perspective. The hope is that this perspective will prove to be equally significant, and that it will interact fruitfully with the other two. For example, quantifiers can be expressed in the proposed languages, and these languages can be seen as embodying a simple set theory, with bounded quantification and no axiom of infinity<sup>1</sup>. Moreover, the algebraic perspective emphasizes *equational theories*, and provability in these theories can make both checking optimizations, and the search for optimizations more systematic, with a potential for partial automation. Finally, while we do not discuss operational semantics in this paper, there is clearly an intimate connection here with the equational theories that waits to be explained.

## 1.1 Overview

Our core language (subsection 2.1) combines in an orthogonal fashion constructs for manipulating functions—first-order lambda abstraction and application, tuples—pairing and projection, and sets—the operations of the set monad in “extension form”. Since the last ingredients give most of the flavor, we call it the *monad calculus*. This apparently very restricted language can already express interesting manipulations of nested relations. Our first result is that the monad calculus is equivalent, both semantically and in terms of equational theories, to a *monad algebra* which is the language of cartesian categories with a strong (or “internalizable”) monad (subsection 2.2). In addition to explaining expressive power, this result is technically useful since the monad algebra is indeed an algebra—it has no bound variables, and hence is easier to use in proofs. The equivalence can be parameterized by an arbitrary signature of additional primitive types, constants and functions, and hence applies to most of the languages considered later.

In the next (section 3), we show that it is the same thing add to the monad core enriched with emptyset and union, *any* of the following operations: set intersection, equality, set difference, subset, membership in a set, and relational nesting, since any of these is definable in terms of any other of them. We also add a

---

<sup>1</sup>Representing logical theories in the lambda calculus was the original motivation for inventing the latter, and the first successful example of this kind was Church’s simple theory of types (1941).

conditional operation and we show that the resulting language is polynomially bounded. This complexity bound and its remarkable versatility and expressive power, make this language an excellent candidate for the “right” concept of *nested relational algebra*.

The power of the complex-object algebra introduced by Abiteboul and Beeri is obtained by adding the powerset construct (section 4). The presence of powerset adds a lot to the expressive power, including the ability to compute certain least fixed points, but at the price of suggesting intractable algorithms for certain tractable queries. Intuitively, this is due to a lack of programming flexibility. We succeed in formalizing one aspect of this by proving that this algebra cannot define a *polymorphic* cardinality function.

The most powerful language we consider (section 5) consists of the lambda calculus and products part of the core, singleton set, emptyset, set union, and structural recursion on the insert presentation of sets, as well as equality (equivalently: intersection, or difference, or *etc.* ). We show that all the other previously considered constructs are definable here, and that we can define a polymorphic cardinality function. Moreover, we formalize the intuition that structural recursion is a least fixed point and hence could be simulated in the Abiteboul-Beeri algebra, but this simulation is not polymorphic and cannot accomodate additional primitive functions.

The last section tries to exploit the categorical perspective in discovering identities that can be useful in optimizations. In particular, we show that the meanings of closed polymorphic expressions are actually natural transformations, and that the naturality can be expressed as identities in the language.

In the remainder of the introduction, we review some previous work on nested relational and complex-object languages; in the last subsection we give an introduction to structural recursion on sets.

## 1.2 Nested Relation and Complex-Object Languages

The first two of these languages have been well-studied in the database literature and require little introduction; their intent is to operate on relations that are “non-first-normal-form”; values stored in a relation may themselves be relations. One of the first descriptions of *Nested Relational Algebra* was given by Schek and Sholl [19], who discuss operations for nesting and unnesting and a generalized form of projection that allows projection to be carried out on inner relations. However operations such as join can only be carried out at the top level; and [9, 10], for example, adds a further operation that allows a nested join. However it is not clear that this is all that is needed. As we shall see, it greatly simplifies the description of Nested Relational Algebra if we include a *map* operation together with the ability to define first-order functions. With this, even nesting and unnesting become derived operations.

A *Complex-Object* algebra was described by Abiteboul *et al* [1, 2] which introduces a *powerset* operation. One of the major findings of this work is that a calculus, an algebra and a form of datalog augmented with certain predicates on sets, have equal expressive power. However this augmentation of nested relational algebra takes it out of polynomial time. For example, transitive closure can be expressed in the algebra, but the obvious program requires the construction of the powerset of the values in the input relation. Interestingly, this algebra does include a *map*, but it is presented in conjunction with a “replace specification” which, in a rather complicated fashion, manages to avoid explicit  $\lambda$ -abstraction.

## 1.3 Monads and Comprehensions

The syntax and semantics of some of the sublanguages we shall consider are inspired by the categorical notion of a monad. The idea that monads could be used to organize semantics of programming constructs is due to Moggi [16]. Wadler [23] showed that they are also useful in organizing syntax, in particular they explain the “list-comprehension” syntax of functional programming. Moreover Trinder and Wadler [22] showed that an extension of comprehensions can implement the (flat) relational calculus. Trinder and Watt [21, 24], have

also sought after a uniform algebra for several different bulk types; in particular they have proved a number of optimizations using categorical identities.

## 1.4 Introduction to Structural Recursion on Sets

**Types.** First, we have the *o-types* (for object types) given by the following context-free grammar

$$\sigma ::= b \mid \sigma \times \sigma \mid \text{unit} \mid \{\sigma\}$$

where  $b$  ranges over an unspecified collection of primitive types (such as *int* or *string*).

O-types are syntax, and they denote sets of complex objects, with the obvious interpretation. For instance,  $\{\sigma\}$  denotes the set of all *finite* subsets of the set denoted by  $\sigma$  (which need not be finite). Examples:  $\{\text{int} \times (\text{string} \times \text{int})\}$  – a type of “flat” relations;  $\{\{\text{int}\} \times (\text{unit} \times \{\text{string}\})\}$  – a type of “nested relations”; *unit* is a base type corresponding to the zero-ary cartesian product so only the empty tuple  $()$  has type *unit*. When convenient, we will confuse syntax and semantics, namely o-types  $\sigma$  with the sets of complex objects they denote,  $\llbracket \sigma \rrbracket$ , even writing  $q : \sigma$  when  $q$  is a complex object belonging to the denotation  $\llbracket \sigma \rrbracket$  of  $\sigma$ .

Note that rather than following the common relational database practice of using n-tuples, we use pairs. An n-tuple can be encoded as a nested pair of type  $(\sigma_1 \times (\sigma_2 \times (\dots \sigma_n \dots)))$  and a n-column relation as a set of such tuples. For most of this paper, we will consider languages whose expressions denote either complex objects, or functions mapping complex objects to complex objects. Hence, all the types we will use are either o-types  $\sigma$  or the simplest kind of function types

$$\sigma \rightarrow \tau$$

where  $\sigma$  and  $\tau$  are o-types.

**Universality properties turned into syntax.** The set of all finite subsets has two different algebraic structures that satisfy universality properties. For the first one, let us denote the empty subset by  $\{\}$ , and *inserting* an element  $x$  in a set  $S$  by  $x \uparrow S$ . It is not hard to verify that for any  $\tau$ , any  $e : \tau$  and any  $i : \sigma \times \tau \rightarrow \tau$  satisfying

- (1)  $i(x, i(y, a)) = i(y, i(x, a))$
- (2)  $i(x, i(x, a)) = i(x, a)$

there exists a *unique*  $g : \{\sigma\} \rightarrow \tau$  such that

$$\begin{array}{l} \text{fun} \quad g(\{\}) = e \\ \quad \quad | \quad g(x \uparrow S) = i(x, g(S)) \end{array}$$

For the second universality property, denoting singleton sets by  $\{x\}$ , and union by  $\cup$ , it is again not hard to verify that for any  $\tau$ , any  $e : \tau$ , any  $f : \sigma \rightarrow \tau$  and any  $u : \tau \times \tau \rightarrow \tau$  satisfying

- (3)  $u(a, u(b, c)) = u(u(a, b), c) \quad u(e, a) = u(a, e) = a$
- (4)  $u(a, b) = u(b, a)$
- (5)  $u(a, a) = a$

(in other words,  $(\tau, u, e)$  is an idempotent commutative monoid), there exists a *unique*  $h : \{\sigma\} \rightarrow \tau$  such that

$$\begin{array}{l} \text{fun} \quad h(\{\}) = e \\ \quad \quad | \quad h(\{x\}) = f(x) \\ \quad \quad | \quad h(S_1 \cup S_2) = u(h(S_1), h(S_2)) \end{array}$$

We say that  $g$  and  $h$  above are defined by *structural recursion* on the insert, respectively union, presentation of sets. Each of the two universality properties can be taken as a basis for a programming construct. Here are the corresponding rules for expression formation.

$$\frac{e : \tau \quad i : \sigma \times \tau \rightarrow \tau}{sri(e, i) : \{\sigma\} \rightarrow \tau} \qquad \frac{e : \tau \quad f : \sigma \rightarrow \tau \quad u : \tau \times \tau \rightarrow \tau}{sru(e, f, u) : \{\sigma\} \rightarrow \tau}$$

Actually,  $sru(\cdot)$  is immediately definable in terms of  $sri(\cdot)$  (see section 5) and the converse is also true albeit with the use of some higher-order types that we chose to avoid in this paper (see [6]).

This, as we hope to demonstrate, provides a rather powerful programming language for sets. For example,

$$\begin{aligned} all &= sru(true, \lambda x.x, \wedge) && \text{Are all elements of the input set true?} \\ map\ f &= sru(\{\}, \lambda x.\{f\ x\}, \cup) && \text{Map the function } f \text{ over the input set} \end{aligned}$$

but, in fact, everything in this paper can be expressed with structural recursion, as we explain in section 5 (and see also [4]).

The somewhat delicate aspect of using structural recursion on sets is that in order for, say,  $sru(e, f, u)$  to have a meaning, the semantic conditions (3), (4) and (5) must be satisfied. For example the “bad count” function  $sru(0, \lambda x.1, +) : \{\sigma\} \rightarrow nat$  doesn’t have a meaning because  $+$  is not idempotent (see section 5 on how to count). The situation can be complicated by global variables appearing in  $e$  and  $u$ . In [6] we discuss the precise semantics of programs with such constructs in them, in the presence of lambda abstraction, as well as a logic for proving that a program is well-defined (has a meaning).

As we mentioned before this paper is concerned with various restrictions to the use of structural recursion, in particular restrictions for which the semantic conditions for meaning existence are automatically satisfied. We first investigate the restriction

$$ext(f) = sru(\{\}, f, \cup)$$

where  $f : \sigma \rightarrow \{\tau\}$  hence  $ext(f) : \{\sigma\} \rightarrow \{\tau\}$ . Note that  $ext(f) S$  maps  $f$  over each member of  $S$  and then “flattens” the resulting set of sets into a set. (This is analogous to Lisp’s *flatmap* operation on lists.) It turns out that this is already a rather powerful construct. To illustrate this, suppose that we can also make use of a little lambda abstraction, we can form pairs  $(e_1, e_2)$ , and we can take left and right projections ( $\pi_1$  and  $\pi_2$ ) from pairs. This very simple language already allows us to define a number of familiar functions, for example

- $\mu \stackrel{\text{def}}{=} ext(\lambda S.S)$  has type  $\{\{\sigma\}\} \rightarrow \{\sigma\}$  and “flattens” a set of sets.
- $map(f) \stackrel{\text{def}}{=} ext(\lambda x.\{f\ x\})$  applies  $f$  to each element of its argument. If  $f : \sigma \rightarrow \tau$  then  $map(f) : \{\sigma\} \rightarrow \{\tau\}$ .
- $\Pi_1 \stackrel{\text{def}}{=} map(\lambda x.\pi_1 x)$  and  $\Pi_2 \stackrel{\text{def}}{=} map(\lambda x.\pi_2 x)$  are relational projections. They project out the left and right columns of (binary) relations.
- $cartprod(S_1, S_2) \stackrel{\text{def}}{=} ext(\lambda x_1.map(\lambda x_2.(x_1, x_2)) S_2) S_1$  takes the cartesian product of two sets. If  $S_1 : \{\sigma\}$  and  $S_2 : \{\tau\}$  then  $cartprod(S_1, S_2) : \{\sigma \times \tau\}$ .
- $unnest_2 \stackrel{\text{def}}{=} ext(\rho_2)$  where  $\rho_2(x, S) \stackrel{\text{def}}{=} map(\lambda y.(x, y)) S$  is a “pairwith” function.  $unnest_2$  is the right-unnesting operation of nested relational algebra. It takes a right-nested relation of type  $\{\sigma \times \{\tau\}\}$  and yields a relation of type  $\{\sigma \times \tau\}$ .

This suggests that there may be a relationship between this limited use of structural recursion and relational query languages. In fact [4] shows such a relationship with the (flat) relational algebra by considering some highly restricted syntactic forms. We were therefore drawn to consider the power of the  $ext(\cdot)$  when orthogonally combined with other constructs and this leads inevitably to the notion of a *monad*.

## 2 The monad calculus and algebra

### 2.1 The monad calculus $\mathcal{MC}$

We define a first-order lambda calculus with products, with a singleton set construct  $\{.\}$ , the “flatmap” construct  $ext(.)$  that we have discussed earlier, everything being parameterized with unspecified primitive constants and functions. Note that union and empty set are not in this language.

**Expressions.** We assume given an infinite collection of variables, and, for simplicity, each is assigned once and forever an o-type,  $x : Type(x)$ . (Variables can range only over complex objects—an important restriction that excludes higher-order functions.) The expressions and their types are given by the following rules (here  $e, e_1, e_2$  range over expressions,  $x$  over variables, and  $\sigma, \tau$  over o-types):

$$\begin{array}{c}
 \frac{}{x : Type(x)} \qquad \frac{}{c : Type(c)} \qquad \frac{}{p : DType(p) \rightarrow CType(p)} \\
 \\
 \frac{e : \tau}{\lambda x. e : Type(x) \rightarrow \tau} \qquad \frac{e_1 : \sigma \rightarrow \tau \quad e_2 : \sigma}{e_1 e_2 : \tau} \\
 \\
 \frac{e_1 : \sigma \quad e_2 : \tau}{(e_1, e_2) : \sigma \times \tau} \qquad \frac{e : \sigma \times \tau}{\pi_1 e : \sigma \quad \pi_2 e : \tau} \qquad \frac{}{() : unit} \\
 \\
 \frac{e : \sigma}{\{e\} : \{\sigma\}} \qquad \frac{e : \sigma \rightarrow \{\tau\}}{ext(e) : \{\sigma\} \rightarrow \{\tau\}}
 \end{array}$$

where  $c$  ranges over an unspecified collection of primitive constants, each with an assigned o-type  $Type(c)$ , and  $p$  ranges similarly over primitive functions, each with an assigned function type  $DType(p) \rightarrow CType(p)$ . Each expression has a unique type.

**Notational convention.** We find it convenient sometimes to use a curried notation, as syntactic sugar. For example we may write the “pairwith” function  $\rho_2 : \sigma \times \{\tau\} \rightarrow \{\sigma \times \tau\}$  as  $\lambda x. \lambda S. ext(\lambda y. \{(x, y)\}) S$  rather than the clumsier “official” notation  $\lambda w. ext(\lambda y. \{(\pi_1 w, y)\})(\pi_2 w)$ .

For a given *primitive signature*  $\Sigma = \{b, c, p\}$  of primitive types, constants, and functions, we denote by  $\mathcal{MC}(\Sigma)$  the resulting calculus.

### 2.2 The monad algebra $\mathcal{MA}$

Keeping with our source of inspiration for these languages, category theory, we will call the expressions in  $\mathcal{MA}$ —*morphism expressions* or simply *morphisms*. Their types are function types, *i.e.*, only of the form  $\sigma \rightarrow \tau$  where  $\sigma, \tau$  are o-types. Given a primitive signature  $\Sigma$  for  $\mathcal{MC}$ , we will have a corresponding primitive signature for  $\mathcal{MA}$ , as follows. The primitive functions  $p$  of  $\mathcal{MC}$  are also primitive morphisms in  $\mathcal{MA}$ . The rest of the primitive morphisms of  $\mathcal{MA}$ ,  $Kc$ , correspond one-to-one to the primitive constants  $c$  of  $\mathcal{MC}$ . For simplicity, we denote the primitive signature of  $\mathcal{MA}$  also by  $\Sigma$ . The variable morphisms  $Kx$  of  $\mathcal{MA}$  correspond one-to-one to the variables  $x$  of  $\mathcal{MC}$ .

$$\begin{array}{c}
 \frac{}{Kx : unit \rightarrow Type(x)} \qquad \frac{}{Kc : unit \rightarrow Type(c)} \qquad \frac{}{p : DType(p) \rightarrow CType(p)} \\
 \\
 \frac{f : \sigma \rightarrow \tau \quad g : \tau \rightarrow v}{g \circ f : \sigma \rightarrow v} \qquad \frac{}{id_\sigma : \sigma \rightarrow \sigma}
 \end{array}$$

$$\begin{array}{c}
\frac{f_1 : \sigma \rightarrow \tau_1 \quad f_2 : \sigma \rightarrow \tau_2}{\langle f_1, f_2 \rangle : \sigma \rightarrow (\tau_1 \times \tau_2)} \qquad \frac{}{fst_{\sigma, \tau} : \sigma \times \tau \rightarrow \sigma} \qquad \frac{}{snd_{\sigma, \tau} : \sigma \times \tau \rightarrow \tau} \\
\frac{f : \sigma \rightarrow \tau}{map(f) : \{\sigma\} \rightarrow \{\tau\}} \qquad \frac{}{\eta_\sigma : \sigma \rightarrow \{\sigma\}} \qquad \frac{}{\mu_\sigma : \{\{\sigma\}\} \rightarrow \{\sigma\}} \\
\frac{}{\rho_{\sigma, \tau} : \sigma \times \{\tau\} \rightarrow \{\sigma \times \tau\}} \qquad \frac{}{t_\sigma : \sigma \rightarrow unit}
\end{array}$$

**Notational conventions and syntactic sugar.** We omit type subscripts whenever there is no possibility for confusion. If  $f_1 : \sigma_1 \rightarrow \tau_1$  and  $f_2 : \sigma_2 \rightarrow \tau_2$  we will use the shorthand  $f_1 \times f_2 \stackrel{\text{def}}{=} \langle f_1 \circ fst, f_2 \circ snd \rangle : \sigma_1 \times \sigma_2 \rightarrow \tau_1 \times \tau_2$ .

$map(\cdot)$ ,  $\mu$ , and  $\rho_2$  have been explained earlier.  $\eta$  denotes the singleton set function and  $t$  denotes the constant empty tuple function. As examples of expressions in  $\mathcal{MA}$  consider the functions defined in the introduction:

- $\Pi_1 \stackrel{\text{def}}{=} map(fst)$  and  $\Pi_2 \stackrel{\text{def}}{=} map(snd)$  are relational projections (on sets of tuples.)
- $\rho_1 = map(\langle snd, fst \rangle) \circ \rho_2 \circ \langle snd, fst \rangle$ .  $\rho_1$  is like  $\rho_2$ , but pairs to the left.
- $cartprod \stackrel{\text{def}}{=} \mu \circ map(\rho_1) \circ \rho_2$ .
- $unnest_2 \stackrel{\text{def}}{=} \mu \circ map(\rho_2)$ .

It is interesting to note that FQL [7], a language designed for the pragmatic purpose of communicating with functional/network databases was based roughly on the same set of operators as  $\mathcal{MA}$ .

**Translation from  $\mathcal{MC}(\Sigma)$  to  $\mathcal{MA}(\Sigma)$ .** An  $\mathcal{MC}(\Sigma)$ -expression  $e : \sigma$  is translated into an  $\mathcal{MA}(\Sigma)$ -expression  $\mathcal{A}[e] : unit \rightarrow \sigma$ , while an  $\mathcal{MC}$ -expression  $e : \sigma \rightarrow \tau$  is translated into an  $\mathcal{MA}(\Sigma)$ -expression  $\mathcal{A}[e] : \sigma \rightarrow \tau$ . With the exception of lambda-abstraction, a description of the translation is given below.

$$\begin{array}{c}
\frac{}{\mathcal{A}[x] \stackrel{\text{def}}{=} Kx : unit \rightarrow Type(x)} \qquad \frac{}{\mathcal{A}[c] \stackrel{\text{def}}{=} Kc : unit \rightarrow Type(c)} \\
\frac{}{\mathcal{A}[p] \stackrel{\text{def}}{=} p : DType(p) \rightarrow CType(p)} \qquad \frac{}{\mathcal{A}[()] \stackrel{\text{def}}{=} id : unit \rightarrow unit} \\
\frac{\mathcal{A}[e_1] : \sigma \rightarrow \tau \quad \mathcal{A}[e_2] : unit \rightarrow \sigma}{\mathcal{A}[e_1 e_2] \stackrel{\text{def}}{=} \mathcal{A}[e_1] \circ \mathcal{A}[e_2] : unit \rightarrow \tau} \qquad \frac{\mathcal{A}[e_1] : unit \rightarrow \sigma \quad \mathcal{A}[e_2] : unit \rightarrow \tau}{\mathcal{A}[(e_1, e_2)] \stackrel{\text{def}}{=} \langle \mathcal{A}[e_1], \mathcal{A}[e_2] \rangle : unit \rightarrow \sigma \times \tau} \\
\frac{\mathcal{A}[e] : unit \rightarrow \sigma \times \tau}{\mathcal{A}[\pi_1 e] \stackrel{\text{def}}{=} fst \circ \mathcal{A}[e] : unit \rightarrow \sigma \quad \mathcal{A}[\pi_2 e] \stackrel{\text{def}}{=} snd \circ \mathcal{A}[e] : unit \rightarrow \tau} \\
\frac{\mathcal{A}[e] : unit \rightarrow \sigma}{\mathcal{A}[\{e\}] \stackrel{\text{def}}{=} \eta \circ \mathcal{A}[e] : unit \rightarrow \{\sigma\}} \qquad \frac{\mathcal{A}[e] : \sigma \rightarrow \{\tau\}}{\mathcal{A}[ext(e)] \stackrel{\text{def}}{=} \mu \circ map(\mathcal{A}[e]) : \{\sigma\} \rightarrow \{\tau\}}
\end{array}$$

In order to translate lambda-abstraction we show that  $\mathcal{MA}(\Sigma)$  enjoys a *combinatorial* (or *functional*) *completeness* property [15]. Specifically, one can express “abstraction of variables” as a derived operation as follows. For any morphism expression  $h : \sigma \rightarrow \tau$  and for any  $\mathcal{MA}$ -variable  $Kx : unit \rightarrow \xi$  (equivalently, for any  $\mathcal{MC}$ -variable  $x : \xi$ ), we define a morphism expression  $\kappa x.h : \xi \times \sigma \rightarrow \tau$  by

$$\begin{array}{l}
\kappa x.h \stackrel{\text{def}}{=} h \circ snd \quad \text{if } h \text{ does not contain } Kx \\
\kappa x.Kx \stackrel{\text{def}}{=} fst \\
\kappa x.\langle f_1, f_2 \rangle \stackrel{\text{def}}{=} \langle \kappa x.f_1, \kappa x.f_2 \rangle \\
\kappa x.(g \circ f) \stackrel{\text{def}}{=} (\kappa x.g) \circ \langle fst, \kappa x.f \rangle \\
\kappa x.map(f) \stackrel{\text{def}}{=} map(\kappa x.f) \circ \rho_2
\end{array}$$

This operation satisfies a property that corresponds to the  $\beta$ -conversion rule for lambda-abstraction:  $(\lambda x.e) x = e$ . That is, in the equational theory of  $\mathcal{MA}(\Sigma)$  (see appendix A) we can prove  $(\kappa x.h) \circ \langle Kx \circ t, id \rangle = h$ .

With this, if the translation of  $e : \tau$  is  $\mathcal{A}[e] : unit \rightarrow \tau$ , then  $\lambda x.e$  translates to  $\mathcal{A}[\lambda x.e] \stackrel{\text{def}}{=} (\kappa x.\mathcal{A}[e]) \circ \langle id, t \rangle : Type(x) \rightarrow \tau$ .

**Translation from  $\mathcal{MA}(\Sigma)$  to  $\mathcal{MC}(\Sigma)$ .** An  $\mathcal{MA}$ -expression  $f : \sigma \rightarrow \tau$  is translated into an  $\mathcal{MC}$ -expression  $\mathcal{C}[f] : \sigma \rightarrow \tau$ . A description of the translation is given below.

$$\begin{array}{ll}
\mathcal{C}[Kx] & \stackrel{\text{def}}{=} \lambda u.x \\
\mathcal{C}[p] & \stackrel{\text{def}}{=} p \\
\mathcal{C}[id] & \stackrel{\text{def}}{=} \lambda x.x \\
\mathcal{C}[fst] & \stackrel{\text{def}}{=} \lambda z.\pi_1 z \\
\mathcal{C}[t] & \stackrel{\text{def}}{=} \lambda x.() \\
\mathcal{C}[\eta] & \stackrel{\text{def}}{=} \lambda x.\{x\} \\
\mathcal{C}[\rho_2] & \stackrel{\text{def}}{=} \lambda w.ext(\lambda y.\{(\pi_1 w, y)\})(\pi_2 w) \\
\mathcal{C}[Kc] & \stackrel{\text{def}}{=} \lambda u.c \\
\mathcal{C}[g \circ f] & \stackrel{\text{def}}{=} \lambda x.\mathcal{C}[g](\mathcal{C}[f]x) \\
\mathcal{C}[\langle f_1, f_2 \rangle] & \stackrel{\text{def}}{=} \lambda x.(\mathcal{C}[f_1]x, \mathcal{C}[f_2]x) \\
\mathcal{C}[snd] & \stackrel{\text{def}}{=} \lambda z.\pi_2 z \\
\mathcal{C}[map(f)] & \stackrel{\text{def}}{=} ext(\lambda x.\{\mathcal{C}[f]x\}) \\
\mathcal{C}[\mu] & \stackrel{\text{def}}{=} ext(\lambda S.S)
\end{array}$$

**Theorem 2.1** *For any closed  $f : \sigma \rightarrow \tau$  in  $\mathcal{MA}(\Sigma)$ ,  $\mathcal{C}[f]$  and  $f$  denote the same function. For any closed  $e : \sigma \rightarrow \tau$  in  $\mathcal{MC}(\Sigma)$ ,  $\mathcal{A}[e]$  and  $e$  denote the same function. For any closed  $e : \sigma$  in  $\mathcal{MC}(\Sigma)$ ,  $\mathcal{A}[e]$  and  $\lambda u.e$  denote the same function, where  $u : unit$  is arbitrary. Hence, the translations preserve semantics.*

We conclude that  $\mathcal{MA}(\Sigma)$  and  $\mathcal{MC}(\Sigma)$  are equally expressive, written

$$\mathcal{MC}(\Sigma) \simeq \mathcal{MA}(\Sigma)$$

Actually, there is a deeper result that shows that there exists an intimate connections between the equational theories for the calculus and the algebra (see appendices B and A) namely we can show that the translations preserve and reflect these theories:

**Theorem 2.2**

- (i)  $\mathcal{MA} \vdash \mathcal{A}[\mathcal{C}[f]] = f$
- (ii)  $\mathcal{MC} \vdash \mathcal{C}[\mathcal{A}[e]] = e$  if  $e$  has function type, and  $\mathcal{MC} \vdash \mathcal{C}[\mathcal{A}[e]] = \lambda u.e$  if  $e$  has o-type ( $u$  not free in  $e$ ).
- (iii)  $\mathcal{MC} \vdash e_1 = e_2$  iff  $\mathcal{MA} \vdash \mathcal{A}[e_1] = \mathcal{A}[e_2]$ .
- (iv)  $\mathcal{MA} \vdash f_1 = f_2$  iff  $\mathcal{MC} \vdash \mathcal{C}[f_1] = \mathcal{C}[f_2]$ .

In fact, one can even extend this by adding arbitrary *closed* axioms. Similar results hold for the connection between simply typed lambda calculi and cartesian closed categories [5].

One immediate benefit of the equivalence of the algebra and the calculus via the translations, is that we can write “mixed expressions” that use constructs from both formalisms. For example, we have  $map(f) = ext(\eta \circ f)$  as well as  $\rho_2 = \lambda x.\lambda S.map(\lambda y.(x, y))S$ . The resulting total language

$$\mathcal{M}(\Sigma) \stackrel{\text{def}}{=} \mathcal{MC}(\Sigma) + \mathcal{MA}(\Sigma)$$

can be thought of as an extension by syntactic sugar of either the algebra or the calculus, or it can be thought of as a single formalism, whose equational theory is obtained by joining the theories of the algebra and the calculus and adding the equations that define the two translations. The result is a very rich, semantically sound, equational theory in which optimizing transformations can be validated. It is convenient to use all of  $\mathcal{M}$ , knowing that in order to derive results about  $\mathcal{M}$ , we only need to prove them for one of the equivalent sublanguages  $\mathcal{MC}$  or  $\mathcal{MA}$ .

### 3 Nested Relational Algebra

We now turn to other operations suggested by (nested) relational algebra. We have already seen that the (relational) projection operations  $\Pi_1, \Pi_2$ , *cartprod*, and the unnesting operations *unnest*<sub>1</sub>, *unnest*<sub>2</sub> can be expressed in  $\mathcal{MC}$ .

**Union and emptyset.** Straightforward reasoning about the structural transformations achievable in  $\mathcal{M}()$  shows that neither binary union nor emptyset is expressible in that language. We therefore add these as primitives at all types.

$$\{\}_\sigma : \{\sigma\} \quad \text{union}_\sigma : \{\sigma\} \times \{\sigma\} \rightarrow \{\sigma\} \quad \mathcal{M}_U \stackrel{\text{def}}{=} \mathcal{M}(\{\}, \text{union})$$

We use the shorthand notations  $e_1 \cup e_2 \stackrel{\text{def}}{=} \text{union}(e_1, e_2)$ ,  $e_1 \uplus e_2 \stackrel{\text{def}}{=} \{\{e_1\} \cup e_2\}$  and  $\{e_1, e_2, \dots, e_n\} \stackrel{\text{def}}{=} (\dots(\{e_1\} \cup \{e_2\}) \cup \dots) \cup \{e_n\}$ .

**Selection.** Selection (filtering) normally requires a boolean type. We will stay within the realm of the already introduced complex object types by *simulating* booleans as follows. We represent *true* as  $\{\}$  and *false* as  $\{\}$ , which are the two values of type  $\{\text{unit}\}$ . Note that union provides disjunction on this representation. With this, selection is *definable* in  $\mathcal{M}_U$  as

$$\text{select}(p) = \text{ext}(\lambda x. \Pi_1(\text{cartprod}(\{x\}, p x)))$$

where  $p : \sigma \rightarrow \{\text{unit}\}$  and  $\text{select}(p) : \{\sigma\} \rightarrow \{\sigma\}$ . The trick here is that if  $p x$  is false (*i.e.*, empty), then  $\text{cartprod}(\{x\}, p x)$  is empty, and  $x$  does not contribute to the result.

**Intersection.** Again, this cannot be derived. To see this, define an ordering  $\preceq_\tau$  on complex objects of type  $\tau$ , for all o-types, inductively:

$$\begin{array}{ll} q \preceq_b q & \text{for any } q \in \llbracket b \rrbracket, b \text{ base} \\ (q_1, q_2) \preceq_{\tau_1 \times \tau_2} (r_1, r_2) & \text{if } q_1 \preceq_{\tau_1} r_1 \text{ and } q_2 \preceq_{\tau_2} r_2 \\ Q \preceq_{\{\tau\}} R & \text{if } \forall q \in Q \exists r \in R q \preceq_\tau r \end{array}$$

It can be checked that all the functions definable in  $\mathcal{M}_U$  are monotone with respect to this ordering, while intersection is not. We could also consider various other augmentations of  $\mathcal{M}_U$ . For example  $\mathcal{M}_U(=)$  is  $\mathcal{M}_U$  augmented with an equality functions  $eq_\sigma : \sigma \times \sigma \rightarrow \{\text{unit}\}$ . Similarly, we should consider augmentations with *difference* (set difference), *subset* (a subset predicate), *member* (a membership predicate) and *nest*, a relational nesting operation. The last of these consists of one of two mutually definable operations *nest*<sub>1</sub> and *nest*<sub>2</sub>. For example *nest*<sub>2</sub> is the right-nesting operation of type  $\{\sigma \times \tau\} \rightarrow \{\sigma \times \{\tau\}\}$ . Remarkably, all these are interdefinable. A similar result was proved by Gyssens and Gucht for a nested relational algebra [11]. However, they needed a powerset operator while we do not require anything so drastic.

**Theorem 3.1**  $\mathcal{M}_U(\cap, \Sigma) \simeq \mathcal{M}_U(=, \Sigma) \simeq \mathcal{M}_U(\text{difference}, \Sigma) \simeq \mathcal{M}_U(\text{subset}, \Sigma) \simeq \mathcal{M}_U(\text{member}, \Sigma) \simeq \mathcal{M}_U(\text{nest}, \Sigma)$  with  $\Sigma$  an unspecified additional primitive signature.

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

Given equality, define  $\eta_\cap(x, y) = \Pi_1 \mu(\text{cartprod}(\{x\}, eq(x, y)))$ .  $\eta_\cap(x, y)$  returns the singleton set  $\{x\}$  if  $eq(x, y)$  is true and  $\{\}$  otherwise. Intersection is now obtained by “flat-mapping” this function over the cartesian product.

$$\cap = \text{ext}(\eta_\cap) \circ \text{cartprod}$$

Conversely, equality may be defined from intersection by  $eq(x, y) = \text{map}(\lambda x. ()) (\cap(\{x\}, \{y\}))$  Thus  $\mathcal{M}_U(=, \Sigma) = \mathcal{M}_U(\cap, \Sigma)$

**Negation and quantification.** We can now implement the boolean operators. Union and intersection directly implement *or* and *and*. Negation is a little more complicated. Consider the relation  $NEG \stackrel{\text{def}}{=} \{(\{\}, \{\}), (\{\}, \{\})\}$ , which pairs *false* with *true* and *true* with *false*. From this we may select the tuple whose left component matches the input and project out the right component:

$$\text{not } x = \mu(\Pi_2(\text{select}(\lambda y. \text{eq}(x, \pi_1 y)) \text{ NEG}))$$

We can use these to implement existential and universal quantification:  $\text{exists}(p) = \text{ext}(p)$  and  $\text{forall}(p) = \text{not} \circ (\text{exists}(\text{not} \circ p))$ . Thus, for example, if  $P$  is a predicate with a free variable  $x$ , we can represent the predicate calculus notation  $\exists x \in S.P$  as  $\text{exists}(\lambda x.P)S$

This now brings us to the implementation of *difference*:

$$\text{difference}(S_1, S_2) = \text{select}(\lambda x_1. \text{forall}(\lambda x_2. \text{not}(\text{eq}(x_1, x_2)))) S_2) S_1$$

Noting that  $\cap$  is easily obtained from *difference* we have  $\mathcal{M}_U(\cap, \Sigma) \simeq \mathcal{M}_U(\text{difference}, \Sigma)$ .

Equality can be obtained from membership by  $\text{eq}(x, y) = \text{member}(x, \{y\})$ ; membership is obtained from equality by  $\text{member}(x, S) = \text{exists}(\lambda y. \text{eq}(x, y)) S$ ; and the mutual dependence of *member* and *subset* is immediate; so we have  $\mathcal{M}_U(=, \Sigma) = \mathcal{M}_U(\text{member}, \Sigma) = \mathcal{M}_U(\text{subset}, \Sigma)$ .

Finally, we examine  $\text{nest}_2$ , which can be derived from equality as follows. First consider a function  $f$  of type  $\sigma \times \{\sigma \times \tau\} \rightarrow \{\tau\}$ .  $f(x, S)$  returns the set  $\{y | (x, y) \in S\}$ . It can be written  $f(x, S) = \Pi_2(\text{select}(\lambda y. \text{eq}(x, y)) S)$ .  $\text{nest}_2(R)$  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:  $\text{nest}_2 R = \text{map}(f)(\rho_1(\Pi_1 R, R))$ .

Conversely, we show that *difference* may be derived from  $\text{nest}_2$ . To compute  $\text{difference}(R, S)$ , observe that  $(\text{map}(\langle \langle \pi_1, \lambda x. \{ \} \rangle, \pi_2 \rangle) \circ \text{nest}_1 \circ \text{nest}_2 \circ \text{union})(\rho_1(R, \{ \}), \rho_1(S, \{\}))$  is a set containing possibly the following three pairs and nothing else:

$$\{ ( (R \cap S, \{ \}), \{ \{ \}, \{\} \} ), \\ ( (\text{difference}(R, S), \{ \}), \{ \{ \} \} ), \\ ( (\text{difference}(S, R), \{ \}), \{ \{\} \} ) \}$$

Call this set  $U$ . Now we need a way to select the second pair. To accomplish this, let  $W$  be the set

$$\{ ( (\{ \}, \{ \}), \{ \{ \}, \{\} \} ), \\ ( (\{ \}, \{\}), \{ \{ \} \} ), \\ ( (\{ \}, \{ \}), \{ \{\} \} ) \}$$

Then  $(\Pi_1 \circ \text{nest}_1 \circ \text{union})(U, W)$  gives us a set consisting of three sets:

$$\{ \{ (\text{difference}(R, S), \{ \}), (\{ \}, \{\}) \}, \\ \{ (R \cap S, \{ \}), (\{ \}, \{ \}) \}, \\ \{ (\text{difference}(S, R), \{ \}), (\{ \}, \{ \}) \} \}$$

This set is further manipulated by applying the function  $\mu \circ \text{map}(\text{map}(\pi_1 \times \pi_2)) \circ \text{map}(\text{cartprod} \circ \langle \text{id}, \text{id} \rangle)$  to obtain the set  $V$  consisting of the following pairs:

$$\{ ( \text{difference}(R, S), \{ \} ), \\ ( \text{difference}(R, S), \{\} ), \\ ( R \cap S, \{ \} ), \\ ( \{ \}, \{\} ), \\ ( \{ \}, \{ \} ), \\ ( \text{difference}(S, R), \{ \} ) \}$$

Using the fact that the product of any set with the empty set is empty, we apply *cartprod* to each of these pairs to obtain the desired difference:  $(\Pi_1 \circ \mu \circ \text{map}(\text{cartprod}))(V)$ . This completes the proof.  $\square$

**Conditional.** We also find it useful to have a conditional  $\text{cond}(\cdot, \cdot, \cdot)$  of type  $\{\text{unit}\} \times \tau \times \tau \rightarrow \tau$  which returns the second argument if the first is a nonempty set and the third argument otherwise. This function is not definable in  $\mathcal{M}_U(=)$  at all types  $\tau$ . The techniques used in the proof of theorem 3.1 allow us to define it when  $\tau$  does not contain primitive types, and also when  $\tau = \{\tau'\}$ , but one can show that it cannot be defined when  $\tau$  is a primitive type. We will occasionally use as syntactic sugar *if B then e<sub>1</sub> else e<sub>2</sub>*  $\stackrel{\text{def}}{=} \text{cond}(B, e_1, e_2)$ .

**Discussion.** The nested relational algebra was originally conceived as an extension of the relational algebra with *nest* and *unnest* (e.g., Thomas & Fischer [20] and Paradaens & Gucht [18]).

However, these two operations are not, by themselves, adequate for a number of obvious manipulations. The exception is when the relations are suitably partitioned, or when an indexing function and null values are available. Under these circumstances, it is possible to bring a deeply nested relation to the top level and to nest it back after operating on it. However, it is unrealistic to assume that relations are partitioned. It is also quite complicated to program. To alleviate this difficulty, Schek and Scholl [19] proposed a recursive projection operation for navigation. Colby went further by making all his operators recursive [9, 10]. These methods are *ad hoc* in the sense that they required individual definitions of what recursive means for each operators. Our *map* construct permits all operations to be performed at all levels of nesting; thus completely eliminating the need for restructuring through *nest* and *unnest*. It is arguably more uniform than Colby’s approach.

The authors claim that  $\mathcal{M}_U(=, \text{cond})$  may be profitably considered as the “right” nested relational algebra. The important difference from the approaches cited above is that a *map*(.) construct has been added together with a limited amount of lambda-abstraction, which at once simplifies and extends the power of the language. For example, nested joins and nested projections require no special treatment. Moreover,

**Theorem 3.2** *If the functions denoted by the primitive function symbols are computable in polynomial time with respect to the size of their input then any functions that are definable by morphism expressions in  $\mathcal{M}_U(=, \text{cond})$  are computable in polynomial time with respect to the size of their input. (This, of course, for any reasonable definition of complex object size).*

**Proof.** For any morphism expression  $f$ , a time-bound function  $|f| : \mathbb{N} \rightarrow \mathbb{N}$  is given by

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

$\square$

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

From a practical standpoint, this definition of NRA has two advantages. First, as we have already remarked, the categorical algebra gives us a very good handle on the optimizations that may be performed in this setting. Second, no special additions are needed to deal with the *group-by* operations that are common in practical query languages. Suppose that  $f$  is a function (such as *COUNT*, *AVERAGE*, *SUM* etc.) of type  $\{\sigma\} \rightarrow \tau$ . Then  $\text{group-by}_2(f) : \{\gamma \times \sigma\} \rightarrow \{\gamma \times \tau\}$  is simply defined as

$$\text{group-by}_2(f) \stackrel{\text{def}}{=} \text{map}(\text{id} \times f) \circ \text{nest}_2$$

## 4 The Abiteboul-Beeri Algebra

In view of theorem 3.2, powerset is not definable in  $\mathcal{M}_{\cup}(=, cond)$ . If we add for each o-type  $\sigma$  the primitive  $powerset_{\sigma} : \{\sigma\} \rightarrow \{\{\sigma\}\}$  we obtain a formalism equivalent to the complex object algebra introduced by Abiteboul and Beeri [2]. For the purposes of this paper, let us define

$$A\&B \stackrel{\text{def}}{=} \mathcal{M}_{\cup}(=, cond, powerset)$$

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

$$\bigcap S \stackrel{\text{def}}{=} difference(\mu S, \mu(map(\lambda s. difference(\mu S, s)) S))$$

We remark that a test for equal cardinality can also be expressed in A&B because given sets  $S$  and  $T$  we can construct  $powerset(cartprod(S, T))$  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  $S \setminus T$  have equal cardinality.

We have not discussed operational semantics for the languages we have considered, but clearly such expressions suggest exponential algorithms for these queries (when in fact the queries are obviously polynomial). However, it turns out that cardinality, as a function into a primitive type *nat* of natural numbers, is not definable, no matter what arithmetic functions we take as primitives. Indeed, let A&B&N be the extension of A&B with a primitive type *nat* and an *arbitrary* primitive arithmetic signature

$$c : nat \quad p : nat \times \dots \times nat \rightarrow nat$$

We must be careful in what we mean by cardinality not being definable, because for each o-type  $\tau$  not containing *nat* there is a specific and trivial cardinality function of type  $\{\tau\} \rightarrow nat$ . That is because all the sets of type  $\{\tau\}$  are “known” and definable in the language, so we can just compare the argument with each of them. Of course, these expressions 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*  $\alpha, \beta$ , etc. and consider o-type *expressions*

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

To avoid technical problems with the type variables occurring in the type of usual variables, free or bound, we consider only  $\mathcal{MA}(\Sigma)$ -expressions<sup>2</sup>, since they do not have bound variables, and moreover we are interested only in closed, hence variable-free, such expressions, call them *polymorphic expressions*. Type variables may occur in polymorphic expressions, namely in the subscripts of *id*, *fst*, *snd*, *t*,  $\eta$ ,  $\mu$ ,  $\rho_2$ ,  $K\{\}$ , *union*, *eq* and *powerset*, and we can do substitutions in them, e.g.,

$$(snd_{\{\alpha \times unit, \alpha\}})[\{unit\}/\alpha] \equiv snd_{\{\{unit\} \times unit, \{unit\}\}}$$

We say that cardinality is *polymorphically definable* if there exists a polymorphic expression  $card : \{\alpha\} \rightarrow nat$ , where  $\alpha$  is a type variable, such that for each o-type  $\sigma$ , the expression  $card[\sigma/\alpha] : \{\sigma\} \rightarrow nat$  denotes the cardinality function from  $\{\sigma\}$  to  $\mathbb{N}$ .

**Theorem 4.1** *Cardinality is not polymorphically definable in A&B&N.*

<sup>2</sup> Which we can do without loss of generality in view of Theorem 2.1 whose proof is “polymorphic” hence not affected if we work with type expressions.

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

**Lemma 4.2** *For each polymorphic expression  $f$  of A&B&N there exists an increasing map  $\varphi_f : \mathbb{N} \rightarrow \mathbb{N}$  such that for any instantiation  $\hat{f}$  obtained by substituting o-types for all the type variables in  $f$  we have*

$$\max(\llbracket \hat{f} \rrbracket(q)) \leq \varphi_f(\max q)$$

To prove the theorem now, assume a polymorphic cardinality exists  $card : \{\alpha\} \rightarrow nat$ , and let  $M = \varphi_{card}(0)$ . Let  $\tau$  be an o-type not containing  $nat$  such that  $\llbracket \{\tau\} \rrbracket$  has more than  $M$  elements (for example  $\tau$  can be of the form  $\{\dots\{unit\}\dots\}$ ). Then, by the lemma,  $card[\tau/\alpha]$  cannot denote the cardinality function of type  $\{\tau\}$  to  $\mathbb{N}$ .

In the proof of the lemma, we can take  $\varphi_{id} \stackrel{\text{def}}{=} \varphi_{fst} \stackrel{\text{def}}{=} \dots \stackrel{\text{def}}{=} \varphi_{cond} \stackrel{\text{def}}{=} \varphi_{powerset} \stackrel{\text{def}}{=} \text{the identity on } \mathbb{N}$ , which explains why  $\varphi_f$  gives a bound for all instantiations of  $f$ . For the arithmetic primitives, we take, e.g.,  $\varphi_p(n) \stackrel{\text{def}}{=} \max_{x \leq n, y \leq n} \llbracket p \rrbracket(x, y)$ .  $\square$

As we shall see in the next section, structural recursion allows a polymorphic definition of cardinality. Another interesting undefinability example, that we shall only sketch in this extended abstract, arises when we add to A&B a primitive signature consisting of a type  $sl$ , a constant  $\perp : sl$ , and a binary operation  $\sqcup : sl \times sl \rightarrow sl$ . It turns out that it is not possible to define in  $A\&B(sl, \perp, \sqcup)$  an expression  $\llbracket \cdot \rrbracket : \{sl\} \rightarrow sl$  which (uniformly) denotes the function that computes the join of a finite set of elements in each model in which  $(sl, \perp, \sqcup)$  is interpreted as a join-semilattice with least element. Again, this can be done immediately via structural recursion.

## 5 The Power of Structural Recursion

We now consider our most powerful language, which uses the structural recursion construct  $sri(., .)$  that we have explained in subsection 1.4. Let SR be  $\mathcal{MC}$  without  $ext(., .)$  but with  $sri(., .)$ , as well as  $\{\}$  and  $union$ , and moreover  $eq$ .

We proceed to show that everything else we have mentioned so far is already definable in SR.

$sru(e, f, u) = sri(e, \lambda x. \lambda z. u(f x, z))$ , and, as in subsection 1.4  $ext(f) = sru(\{\}, f, union)$ . The conditional is definable as well:  $cond = \lambda B. \lambda x. \lambda y. sri(y, \lambda u. \lambda z. x) B$ . Hence SR is at least as expressive as  $\mathcal{M}_{\cup}(=, cond)$ , our candidate for nested relational algebra.

Moreover, we can define the powerset operator by

$$\begin{array}{l} fun \quad powerset \{\} = \{\{\}\} \\ | \quad powerset(x \uparrow S) = (powerset S) \cup map(\lambda T. x \uparrow T)(powerset S) \end{array}$$

in other words  $powerset = sri(\{\{\}\}, \lambda x. \lambda W. W \cup map(\lambda T. x \uparrow T) W)$ .

Therefore, SR is at least as expressive as A&B. However, SR offers a more flexible programming style that allows for example expressing efficient algorithms for transitive closure as we have shown in [4]. We illustrate the point here with parity. Define a polymorphic expression  $even : \{\alpha\} \rightarrow \{unit\}$  by

$$\begin{array}{l} fun \quad even \{\} = true \\ | \quad even(x \uparrow S) = if\ member(x, S)\ then\ even\ S\ else\ not(even\ S) \end{array}$$

Notice that this is not quite a use of structural recursion as we have defined it, but a more general form  $gensri(e, j) : \{\sigma\} \rightarrow \tau$  where  $e : \tau$  but  $j : \sigma \times \{\sigma\} \times \tau \rightarrow \tau$ . Using a trick that goes back to Kleene,  $gensri(., .)$

can be obtained from a simple structural recursion  $sri(.,.) : \{\sigma\} \rightarrow \{\sigma\} \times \tau$ . The semantic conditions (1) and (2) are readily verified.<sup>3</sup>

Moreover, cardinality is, as promised, polymorphically definable in SR. Define  $card : \{\alpha\} \rightarrow nat$  by:

$$\begin{array}{l} fun \quad card\{\} = 0 \\ | \quad card(x \uparrow S) = if\ member(x, S)\ then\ card\ S\ else\ 1 + (card\ S) \end{array}$$

This is again a generalized structural recursion and it is justified in the same way.

While we have explained through theorem 4.1 and the remark at the end of section 4 in what sense SR is strictly more powerful than A&B, we still want to explain the intuition that since A&B can do certain least fixed points, in fact enough to simulate a Datalog-like language with predicates on sets [2], it will be able to express the maps 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 SR to A&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 intersections of sets of sets, we need some kind of “universe” that 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  $A\&B(C)$  with only<sup>4</sup> finitely many constants  $C \stackrel{\text{def}}{=} \{c_1, \dots, c_n\}$  of one primitive (base) type  $b$ .

**Theorem 5.1**  $A\&B(C) \simeq SR(C)$

**Proof.** For each o-type  $\sigma$ , we can define in A&B the function  $FORTH_\sigma : \sigma \rightarrow \{\iota\}$  that computes the set of all elements of type  $\iota$  that occur in a complex object of type  $\sigma$ . Namely  $FORTH_\iota \stackrel{\text{def}}{=} \eta$ ,  $FORTH_{unit} \stackrel{\text{def}}{=} K\{\}$ ,  $FORTH_{\sigma \times \tau} \stackrel{\text{def}}{=} union \circ (FORTH_\sigma \times FORTH_\tau)$ ,  $FORTH_{\{\sigma\}} \stackrel{\text{def}}{=} ext(FORTH_\sigma)$ . Next, we can define for each o-type  $\sigma$  a function  $BACK_\sigma : \{\iota\} \rightarrow \{\sigma\}$  which, given a set  $P$  of elements of type  $\iota$ , computes the “universe” of complex objects of type  $\sigma$  that can be constructed using elements from  $P$ :  $BACK_\iota \stackrel{\text{def}}{=} \lambda P.P \cup \{c_1\} \cup \dots \cup \{c_n\}$ ,  $BACK_{unit} \stackrel{\text{def}}{=} \lambda P.\{\}$ ,  $BACK_{\sigma \times \tau} \stackrel{\text{def}}{=} cartprod \circ (BACK_\sigma, BACK_\tau)$ ,  $BACK_{\{\sigma\}} \stackrel{\text{def}}{=} powerset \circ BACK_\sigma$ . We verify these definitions by proving

$$\forall q \in \llbracket \sigma \rrbracket, \quad q \in \llbracket BACK_\sigma \circ FORTH_\sigma \rrbracket(q)$$

Here,  $\llbracket e \rrbracket \rho$  is the meaning of the expression  $e$ , which may have free variables, given an environment  $\rho$ , that is a mapping from variables to values (complex objects) of corresponding type.

**Lemma 5.2** Fix an arbitrary expression  $P : \iota$ , (may have free variables). For any  $e : \sigma$  in SR(C) and any environment  $\rho$

$$(\forall x, \rho(x) \in \llbracket BACK_{Type(x)} P \rrbracket \rho) \implies \llbracket e \rrbracket \rho \in \llbracket BACK_\sigma P \rrbracket \rho$$

For any  $f : \sigma \rightarrow \tau$  in SR(C) and any environment  $\rho$

$$(\forall x, \rho(x) \in \llbracket BACK_{Type(x)} P \rrbracket \rho) \implies (\llbracket f \rrbracket \rho)(\llbracket BACK_\sigma P \rrbracket \rho) \subseteq \llbracket BACK_\tau P \rrbracket \rho$$

<sup>3</sup>With some higher-order lambda calculus we can also express an efficient algorithm for testing equality of cardinality.

<sup>4</sup>For the simplicity of the exposition in this extended abstract.

Now, let  $e : \tau$  and  $i : \sigma \times \tau \rightarrow \tau$  be closed (for simplicity of the exposition) and suppose that for them we have semantically equivalent  $A\&B(C)$  expressions  $e^*$  and  $i^*$ . We will suggest how to construct an  $A\&B(C)$ -expression that is semantically equivalent to  $sri(e, i) : \{\sigma\} \rightarrow \tau$ . Let  $s : \{\sigma\}$  be a variable. Select out of

$$powerset(BACK_{\{\sigma\}}(FORTH_{\{\sigma\}} s) \times BACK_{\tau}(FORTH_{\{\sigma\}} s))$$

only those relations  $R$  that satisfy  $(\{\}, e^*) \in R$  and

$$forall((\lambda x. map((\lambda T. x \uparrow T) \times (\lambda z. i^*(x, z))) R) \subseteq R)s$$

then take the intersection of all the selected relations, call it  $I$ . Using the lemma and the universality property that defines  $sri()$ , we can show that for each value assigned to  $s$ ,  $\llbracket I \rrbracket$  coincides with the restriction of  $\llbracket sri(e, i) \rrbracket$  to  $\llbracket BACK_{\{\sigma\}}(FORTH_{\{\sigma\}} s) \rrbracket$ . Therefore, select from  $I$  all pairs whose left component is  $s$  and project right. The result is an expression  $Q : \{\tau\}$ , and the  $A\&B(C)$ -expression  $\lambda s. Q$  is semantically equivalent to  $sri(e, i)$ , modulo the small unpleasantness that it returns instead of the desired result, a singleton set containing the result. If  $\tau$  is a set type this can be remedied by composing with the flattening function  $\mu$ . The types of the overall translation must be adjusted to take care of this unpleasantness 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  $A\&B(C)$  to  $SR(C)$ .

## 6 Naturality and Optimizations

We can interpret type expressions with  $n$  type variables in them as functors  $SET^n \rightarrow SET$ . Then, taking closed polymorphic expressions in  $A\&B$  *without eq*, 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.

More precisely, for any list of type variables  $(\alpha_1, \dots, \alpha_n)$ , any  $\sigma$ -types  $\sigma_1, \dots, \sigma_n, \tau_1, \dots, \tau_n$ , any expressions  $g_i : \sigma_i \rightarrow \tau_i$ , and any type expression  $\theta$  without primitive types and whose type variables are among  $(\alpha_1, \dots, \alpha_n)$ , define a morphism  $\theta(\vec{g}) : \theta[\vec{\sigma}/\vec{\alpha}] \rightarrow \theta[\vec{\tau}/\vec{\alpha}]$  by induction on  $\theta$  as follows:

$$\begin{aligned} \alpha_i(\vec{g}) &\stackrel{\text{def}}{=} g_i \\ unit(\vec{g}) &\stackrel{\text{def}}{=} id_{unit} \\ \theta_1 \times \theta_2(\vec{g}) &\stackrel{\text{def}}{=} \theta_1(\vec{g}) \times \theta_2(\vec{g}) \\ \{\theta\}(\vec{g}) &\stackrel{\text{def}}{=} map(\theta(\vec{g})) \end{aligned}$$

Now, let  $f : \theta_1 \rightarrow \theta_2$  be a polymorphic expression in  $A\&B$  *without eq*, whose type variables are among  $(\alpha_1, \dots, \alpha_n)$ . We have

### Theorem 6.1 (Naturality)

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

It appears that this generalizes all the identities about “pushing  $map(\cdot)$  through” that we have so far seen to be used in optimizations, including a couple we proposed in [4].

If the meanings of the  $g$ 's are injective functions then this holds for  $eq$  as well. By taking the semantic statement and the  $g$ 's to be bijections we get that all the queries definable in A&B are *generic* or *consistent* [8]. (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.

This is the place to emphasize that the queries definable in our languages are automatically order-independent. This is in contrast with the languages considered in [14] whose semantics is on ordered sets. On the other hand, quite likely not all polynomial-time order independent queries are definable in our languages as we believe that the lower bound suggested in [14] applies.

## 7 Further Work and Conclusions

This is an initial report on a research program to investigate the usefulness of structural recursion on sets as the foundation for database query languages. We have obtained further results that are not included here and which we hope to publish shortly:

- One of us (Limsoon Wong) has shown that  $\mathcal{M}_U(=, cond)$  (our proposal for a nested relational algebra, see section 3) is conservative over the usual (flat) relational algebra. Among other things, this implies that neither parity nor transitive closure are definable in  $\mathcal{M}_U(=, cond)$ .
- Trinder and Wadler [22] have observed that the “comprehension” syntax of functional programming may provide an elegant way of expressing queries, not unlike that of the relational calculus or SQL. We have investigated the precise relations between the ideas developed in this paper and comprehension syntax. However, to be consistent with relational databases and database languages that exploit polymorphic record types [17] we have modified our categorical languages to work with records rather than products.
- The operations we have considered on sets have natural analogues for bags and lists (for structural recursion see [6]). We have investigated them in a more general setting of *collection types*, which include lists, bags and sets, as well as certain trees.

We believe that from a theoretical standpoint our approach provides us a new way of examining relationships among languages; from a practical standpoint it provides us with a better understanding of how to embed query languages in programming languages in that it suggests more general optimization techniques and a general syntactic framework. Moreover – but this is a matter of taste – we believe that the account given here provides a great deal of uniformity to the study of languages for complex objects or nested relations.

**Acknowledgements.** The authors thank Foto Afrati, Leonid Libkin, Shamim Naqvi, Hermann Puhmann, Jon Riecke, and Steve Vickers for helpful discussions and Paul Taylor for his diagram macros.

## References

- [1] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An Introduction to the Completeness of Languages for Complex Objects and Nested Relations. In S. Abiteboul, P. C. Fisher, and H.-J. Schek, editors, *LNCS 361: Nested Relations and Complex Objects in Databases*, pages 117–138. Springer-Verlag, 1987.
- [2] Serge Abiteboul and Catriel 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.

- [3] Catriel Beeri and Yoran Kornatzky. Algebraic Optimisation of Object Oriented Query Languages. In S. Abiteboul and P. C. Kanellakis, editors, *LNCS 470: 3rd International Conference on Database Theory, Paris, France, December 1990*, pages 72–88, Berlin, December 1990. Springer-Verlag.
- [4] V. Breazu-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *Proceedings of 3rd International Workshop on Database Programming Languages*, pages 9–19, Nahplion, Greece, August 1991. Morgan Kaufmann.
- [5] V. Breazu-Tannen and A. R. Meyer. Lambda calculus with constrained types (extended abstract). In R. Parikh, editor, *Proceedings of the Conference on Logics of Programs, Brooklyn, June 1985*, pages 23–40. *Lecture Notes in Computer Science*, Vol. 193, Springer-Verlag, 1985.
- [6] V. Breazu-Tannen and 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, Madrid, Spain, July 1991*, pages 60–75. Springer Verlag, 1991.
- [7] O. P. Buneman, R. Nikhil, and R. E. Frankel. An Implementation Technique for Database Query Languages. *ACM Transactions on Database Systems*, 7(2):164–187, June 1982.
- [8] Ashok Chandra and David Harel. Structure and Complexity of Relational Queries. *Journal of Computer and System Sciences*, 25:99–128, 1982.
- [9] Latha S. Colby. A Recursive Algebra and Query Optimisation for Nested Relations. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 273–283, Portland, Oregon, June 1989.
- [10] Latha S. Colby. A Recursive Algebra for Nested Relations. *Information Systems*, 15(5):567–582, 1990.
- [11] Marc Gyssens and Dirk Van Gucht. A Comparison Between Algebraic Query Languages for Flat and Nested Databases. *Theoretical Computer Science*, 87:263–286, 1991.
- [12] R. Harper, R. Milner, and M. Tofte. The Definition of Standard ML, Version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.
- [13] Richard Hull and Jianwen Su. On the Expressive Power of Database Queries with Intermediate Types. *Journal of Computer and System Sciences*, 43:219–267, 1991.
- [14] Neil Immerman, Sushant Patnaik, and David Stemple. The Expressiveness of a Family of Finite Set Languages. In *Proceedings of 10th ACM Symposium on Principles of Database Systems*, pages 37–52, 1991.
- [15] J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.
- [16] Eugenio Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.
- [17] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database Programming in Machiavelli: A Polymorphic Language with Static Type Inference. In James Clifford, Bruce Lindsay, and David Maier, editors, *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 46–57, Portland, Oregon, June 1989.
- [18] Jan Paredaens and Dirk Van Gucht. Possibilities and Limitations of Using Flat Operators in Nested Algebra Expressions. In *Proceedings of 7th ACM Symposium on Principles of Database Systems, Austin, Texas*, pages 29–38, 1988.
- [19] H.-J. Schek and M. H. Scholl. The Relational Model with Relation-Valued Attributes. *Information Systems*, 11(2):137–147, 1986.

- [20] S. J. Thomas and P. C. Fischer. Nested Relational Structures. In P. C. Kanellakis, editor, *Advances in Computing Research: The Theory of Databases*, pages 269–307. JAI Press, 1986.
- [21] P. W. Trinder. Comprehension: A Query Notation for DBPLs. In *Proceedings of 3rd International Workshop on Database Programming Languages*, pages 49–62, Nahplion, Greece, August 1991. Morgan Kaufmann.
- [22] P. W. Trinder and P. L. Wadler. List Comprehensions and the Relational Calculus. In *Proceedings of 1988 Glasgow Workshop on Functional Programming*, pages 115–123, Rothesay, Scotland, August 1988.
- [23] Philip Wadler. Comprehending Monads. In *Proceedings of ACM Conference on Lisp and Functional Programming*, Nice, June 1990.
- [24] David A. Watt and Phil Trinder. Towards a Theory of Bulk Types. Fide Technical Report 91/26, Glasgow University, Glasgow G12 8QQ, Scotland, July 1991.

# APPENDICES

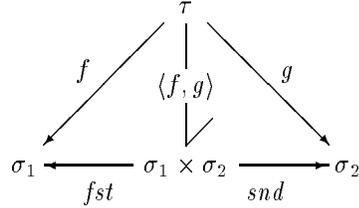
## A Axioms Of $\mathcal{MA}$

The axioms of  $\mathcal{MA}$  are listed below. The reflexivity, symmetry, transitivity, and congruence identities have been omitted.

1. 
$$\frac{f : \tau \rightarrow \tau' \quad g : \tau' \rightarrow \sigma' \quad h : \sigma' \rightarrow \sigma}{h \circ (g \circ f) = (h \circ g) \circ f : \tau \rightarrow \sigma}$$
2. 
$$\frac{f : \tau \rightarrow \sigma}{f \circ id = f : \tau \rightarrow \sigma}$$
3. 
$$\frac{f : \tau \rightarrow \sigma}{id \circ f = f : \tau \rightarrow \sigma}$$
4. 
$$\frac{f \circ Kx = g \circ Kx : unit \rightarrow \tau \quad Kx : unit \rightarrow \sigma \quad x \text{ is fresh}}{f = g : \sigma \rightarrow \tau}$$

That makes  $\mathcal{MA}$  a category with “enough points.”

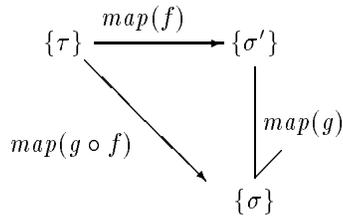
5. 
$$\frac{f : \tau \rightarrow \sigma_1 \times \sigma_2}{\langle fst \circ f, snd \circ f \rangle = f : \tau \rightarrow \sigma_1 \times \sigma_2}$$
6. 
$$\frac{f : \tau \rightarrow \sigma_1 \quad g : \tau \rightarrow \sigma_2}{fst \circ \langle f, g \rangle = f : \tau \rightarrow \sigma_1}$$
7. 
$$\frac{f : \tau \rightarrow \sigma_1 \quad g : \tau \rightarrow \sigma_2}{snd \circ \langle f, g \rangle = g : \tau \rightarrow \sigma_2}$$



8. 
$$\frac{f : \tau \rightarrow unit}{f = t : \tau \rightarrow unit}$$

The category is *cartesian*, i.e., it has been endowed with all finite products.

9. 
$$\frac{f : \tau \rightarrow \sigma' \quad g : \sigma' \rightarrow \sigma}{map(g \circ f) = (map(g)) \circ (map(f)) : \{\tau\} \rightarrow \{\sigma\}}$$



$$10. \frac{}{map(id) = id : \{\sigma\} \rightarrow \{\sigma\}}$$

$$11. \frac{f : \tau \rightarrow \sigma}{(map(f)) \circ \eta = \eta \circ f : \tau \rightarrow \{\sigma\}}$$

$$\begin{array}{ccc} \tau & \xrightarrow{f} & \sigma \\ \eta \downarrow \swarrow & & \downarrow \swarrow \eta \\ \{\tau\} & \xrightarrow{map(f)} & \{\sigma\} \end{array}$$

$$12. \frac{f : \tau \rightarrow \sigma}{(map(f)) \circ \mu = \mu \circ map(map(f)) : \{\{\tau\}\} \rightarrow \{\sigma\}}$$

$$\begin{array}{ccc} \{\{\tau\}\} & \xrightarrow{\mu} & \{\tau\} \\ map(map(f)) \downarrow \swarrow & & \downarrow \swarrow map(f) \\ \{\{\sigma\}\} & \xrightarrow{\mu} & \{\sigma\} \end{array}$$

$$13. \frac{}{\mu \circ \eta = id : \{\sigma\} \rightarrow \{\sigma\}}$$

$$14. \frac{}{\mu \circ map(\eta) = id : \{\sigma\} \rightarrow \{\sigma\}}$$

$$\begin{array}{ccccc} \{\sigma\} & \xrightarrow{\eta} & \{\{\sigma\}\} & \xleftarrow{map(\eta)} & \{\sigma\} \\ & \searrow id & \downarrow \mu & \swarrow id & \\ & & \{\sigma\} & & \end{array}$$

$$15. \frac{}{\mu \circ map(\mu) = \mu \circ \mu : \{\{\{\sigma\}\}\} \rightarrow \{\sigma\}}$$

$$\begin{array}{ccc} \{\{\{\sigma\}\}\} & \xrightarrow{map(\mu)} & \{\{\sigma\}\} \\ \mu \downarrow \swarrow & & \downarrow \swarrow \mu \\ \{\{\sigma\}\} & \xrightarrow{\mu} & \{\sigma\} \end{array}$$

The above are the monad identities, where  $map()$  is the action on morphisms of the set monad functor and  $\eta$  and  $\mu$  are natural transformations.

$$16. \overline{(map(snd)) \circ \rho_2 = snd : unit \times \{\sigma\} \rightarrow \{\sigma\}}$$

$$\begin{array}{ccc} unit \times \{\sigma\} & \xrightarrow{\rho_2} & \{unit \times \sigma\} \\ \text{\scriptsize } snd \downarrow & \swarrow & \uparrow \text{\scriptsize } map(snd) \\ \{\sigma\} & & \end{array}$$

$$17. \overline{\rho_2 \circ (id \times \eta) = \eta : \sigma \times \tau \rightarrow \{\sigma \times \tau\}}$$

$$18. \overline{\rho_2 \circ (id \times \mu) = \mu \circ (map(\rho_2)) \circ \rho_2 : \sigma \times \{\{\tau\}\} \rightarrow \{\sigma \times \tau\}}$$

$$\begin{array}{ccccc} \sigma \times \{\{\tau\}\} & \xrightarrow{\rho_2} & \{\sigma \times \{\tau\}\} & \xrightarrow{map(\rho_2)} & \{\{\sigma \times \tau\}\} \\ \text{\scriptsize } (id \times \mu) \downarrow & & & & \downarrow \text{\scriptsize } \mu \\ \sigma \times \{\tau\} & \xrightarrow{\rho_2} & \{\sigma \times \tau\} & & \\ \text{\scriptsize } (id \times \eta) \swarrow & & \nearrow \text{\scriptsize } \eta & & \\ & \sigma \times \tau & & & \end{array}$$

$$19. \overline{map(i) \circ \rho_2 = \rho_2 \circ (id \times \rho_2) \circ i : (\sigma_1 \times \sigma_2) \times \{\sigma_3\} \rightarrow \{\sigma_1 \times (\sigma_2 \times \sigma_3)\}} \text{ where } i \text{ is } \langle fstofst, \langle sndofst, snd \rangle \rangle.$$

$$\begin{array}{ccccc} (\sigma_1 \times \sigma_2) \times \{\sigma_3\} & \xrightarrow{\rho_2} & \{(\sigma_1 \times \sigma_2) \times \sigma_3\} & & \\ \downarrow i & & & & \downarrow map(i) \\ \sigma_1 \times (\sigma_2 \times \{\sigma_3\}) & \xrightarrow{(id \times \rho_2)} & \sigma_1 \times \{\sigma_2 \times \sigma_3\} & \xrightarrow{\rho_2} & \{\sigma_1 \times (\sigma_2 \times \sigma_3)\} \end{array}$$

$$20. \frac{f : \sigma \rightarrow \sigma' \quad g : \tau \rightarrow \tau'}{\text{map}(f \times g) \circ \rho_2 = \rho_2 \circ (f \times \text{map}(g)) : \sigma \times \{\tau\} \rightarrow \{\sigma' \times \tau'\}}$$

$$\begin{array}{ccc}
\sigma \times \{\tau\} & \xrightarrow{\rho_2} & \{\sigma \times \tau\} \\
\downarrow (f \times \text{map}(g)) & & \downarrow \text{map}(f \times g) \\
\sigma' \times \{\tau'\} & \xrightarrow{\rho_2} & \{\sigma' \times \tau'\}
\end{array}$$

The monad is made into a strong monad with a natural transformation  $\rho_2$  using the above identities.

## B Axioms Of $\mathcal{MC}$

The axioms for  $\mathcal{MC}$  are listed below. The reflexivity, symmetry, transitivity, and congruence identities have been omitted.

1.  $\frac{\lambda x.e : \sigma \rightarrow \tau \quad e' : \sigma}{(\lambda x.e)e' = e[e'/x] : \tau}$
2.  $\frac{e = e' : \tau}{\lambda x.e = \lambda x.e' : \sigma \rightarrow \tau}$
3.  $\frac{e : \sigma \rightarrow \tau \quad x \notin FV(e)}{\lambda x.ex = e : \sigma \rightarrow \tau}$

The above are the three usual rules for lambda calculus.

4.  $\frac{e_1 : \sigma \quad e_2 : \tau}{\pi_1(e_1, e_2) = e_1 : \sigma}$
5.  $\frac{e_1 : \sigma \quad e_2 : \tau}{\pi_2(e_1, e_2) = e_2 : \tau}$
6.  $\frac{e : \sigma \times \tau}{(\pi_1 e, \pi_2 e) = e : \sigma \times \tau}$
7.  $\frac{e : \text{unit}}{e = () : \text{unit}}$

The above are rules for finite products.

8.  $\frac{f : \sigma \rightarrow \{\tau\} \quad x : \sigma}{\text{ext}(f)\{x\} = fx : \{\tau\}}$
9.  $\frac{S : \{\sigma\}}{\text{ext}(\lambda x.\{x\})S = S : \{\sigma\}}$
10.  $\frac{S : \{\sigma\} \quad f : \sigma \rightarrow \{\sigma'\} \quad g : \sigma' \rightarrow \{\tau\}}{\text{ext}(g)(\text{ext}(f)S) = \text{ext}(\lambda x.\text{ext}(g)(fx))S : \{\sigma\} \rightarrow \{\tau\}}$

These last identities are that of monad in “extension” form.

We leave the equational axiomatization of  $\mathcal{M}_U(=, \text{cond})$ ,  $A\&B$ , and  $SR$  for a future paper.