Typed Query Languages for Databases Containing Queries*

(extended abstract)

Frank Neven†
Limburgs Universitair Centrum
fneven@luc.ac.be

Jan Van den Bussche
Limburgs Universitair Centrum
vdbuss@luc.ac.be

Dirk Van Gucht
Indiana University
vgucht@cs.indiana.edu

Gottfried Vossen
University of Münster
vossen@uni-muenster.de

Abstract

This paper introduces and studies the *relational meta algebra*, a statically typed extension of the relational algebra to allow for meta programming in databases. In this meta algebra one can manipulate database relations involving not only stored data values (as in classical relational databases) but also stored relational algebra expressions. Topics discussed include modeling of advanced database applications involving "procedural data"; desirability as well as limitations of a strict typing discipline in this context; equivalence with a first-order calculus; and global expressive power and non-redundancy of the proposed formalism.

1 Introduction

Various advanced database systems, such as active and object-oriented systems, as well as the data dictionaries of standard relational database systems, provide the functionality of "stored procedures". The potential functionality of such systems was already envisaged by Stonebraker and his collaborators in the '80s [20, 21]. However, little work has been done on formal models providing logical foundations for such systems. Indeed, current systems approaches treat stored procedures simply as string values. Only the special case of "schema querying" has received a significant amount of attention (e.g., [7, 13]).

The purpose of the present paper is to contribute towards these needed logical foundations, by proposing and studying an extension of the relational algebra to allow for meta programming. The proposed *relational meta algebra*, denoted by $\mathcal{MA}$, extends the relational algebra with three new operators for computing with relations in which not only ordinary data values, but also relational algebra expressions can be stored. The first new operator is `extract`, used to extract subexpressions from stored expressions. The second is `rewrite`, used to rewrite subexpressions according to certain patterns (as is familiar from algebraic query optimization).

The third and most important new operator of $\mathcal{MA}$ is `eval`, used to dynamically evaluate stored expressions. A fundamental property one wants to achieve is *type safety* of `eval`, in the sense that this dynamic evaluation never results in a run-time error. To guarantee type safety, the operators `extract` and `rewrite` are carefully calibrated so that they preserve syntactical correctness and so that the type of the expressions resulting from their manipulations is determined statically.

The type system we put on $\mathcal{MA}$ is an adaptation of the simple two-level type system discussed by Sheard and Hook in the context of Meta-ML [19]. We type ordinary relations by their width, type relational algebra expressions by the type of their result relations, and type relations containing relational algebra expressions by typing the columns as containing either ordinary data values or expressions of a designated type. Expressions of $\mathcal{MA}$, finally, are again typed by the type of their result relations (which may contain expressions).

The contents of this paper are summarized as follows. We begin by recalling the necessary definitions concerning relational databases and relational algebra, and introduce our extension of the relational database model to allow for stored relational algebra expressions in relations (Section 2). Then we introduce the operators of $\mathcal{MA}$ and give examples of interesting queries definable in $\mathcal{MA}$ (Section 3).

After that, we investigate the expressive power of our formalism (Section 4). Specifically, we establish the following results:

1. We present a many-sorted first-order calculus whose "safe" fragment is equivalent to $\mathcal{MA}$, thus extending Codd's classical theorem on the equivalence of relational algebra and calculus [8].[1] This result is a generalization of Ross' [17], who worked in a model allowing only relation names, not general algebra expressions, to be stored in relations.

2. We illustrate an interesting limitation on the expressive power of $\mathcal{MA}$, due to its inherently typed nature: there are computationally extremely simple queries, well-typed at the input and output sides, which are nevertheless not definable in $\mathcal{MA}$, intuitively because

*Work supported by NATO Collaborative Research Grant 960954.
†Research Assistant of the Fund for Scientific Research, Flanders.

Proceedings 17th ACM Symposium on Principles of Database Systems, ACM Press, 1998.

[1] Generalizations of Codd's theorem to extensions of the relation model have always been a popular research topic (e.g., [12, 14, 1, 10, 4]).

their computation requires untyped intermediate results (which cannot be represented by an $\mathcal{MA}$ computation). The equivalence with the calculus allows an elegant model-theoretic proof of this observation.

3. We show that $\mathcal{MA}$ is a conservative extension of the relational algebra, in the sense that as far as queries over ordinary relations (not containing stored expressions) are concerned, $\mathcal{MA}$ is no more expressive than the relational algebra.[2]

4. We give a rigorous proof of the intuitively clear fact that `eval` is a primitive operator in $\mathcal{MA}$: it cannot be simulated using the other operators. This stands in contrast to the situation in a complete programming language such as Lisp, where `eval` is clearly definable in Lisp without `eval` and thus not primitive. (Also the other operators of $\mathcal{MA}$ can be shown to be primitive.)

The paper concludes with a discussion in Section 5.

The present paper is a follow-up on an earlier paper by three of us [24]. There, we studied the expressive power of evaluating stored relational algebra programs in a completely untyped setting. Relational algebra programs were encoded in data relations, and the standard operators of the relational algebra were used to manipulate these "program relations". This approach resulted in a powerful, but difficult to use, query language called the *reflective relational algebra ($\mathcal{RA}$)*. Our main result was that by adding `eval` to the relational algebra much more queries on classical relational databases become definable. This stands in contrast to the conservative extension property of $\mathcal{MA}$ with respect to $\mathcal{A}$ we prove here. In fact, our motivation for the work reported in this paper was the desire *(i)* to understand the situation where typing and type safety are mandatory, and *(ii)* to design a formalism that is more programmer-friendly than $\mathcal{RA}$.

2 Relations, expressions, and meta relations

2.1 Relational databases and relational algebra

Assume a sufficiently large supply of *relation names* is given, where each relation name has an associated *arity* (a natural number). To denote that relation name R has arity n we write $R : n$. A *database schema* is a finite set of relation names.

Assume further a universe $\mathbf{V}$ of data values is given. A *relation of arity n* is a finite subset of $\mathbf{V}^n$. An *instance* of a database schema $\mathcal{S}$ is a mapping $\mathcal{I}$ on $\mathcal{S}$ which assigns to each relation name $R : n \in \mathcal{S}$ a relation $\mathcal{I}(R)$ of arity n.

Fix a schema $\mathcal{S}$. We denote the set of relational algebra expressions over $\mathcal{S}$ by $\mathcal{A}$. Each expression has an arity; as for relation names, to denote that expression e has arity n we write $e : n$. Formally:

- Each $R : n \in \mathcal{S}$ is in $\mathcal{A}$.
- If $e_1 : n$ and $e_2 : n$ are in $\mathcal{A}$, then so are $(e_1 \cup e_2) : n$ and $(e_1 - e_2) : n$.
- If $e_1 : n_1$ and $e_2 : n_2$ are in $\mathcal{A}$, then so is $(e_1 \times e_2) : n_1 + n_2$.
- If $e : n$ is in $\mathcal{A}$, then so are
  - $\sigma_{i=j}(e) : n$, where $i, j \in \{1, \ldots, n\}$; and
  - $\pi_{i_1,\ldots,i_p}(e) : p$, where $i_1, \ldots, i_p \in \{1, \ldots, n\}$.

Given an instance $\mathcal{I}$ of $\mathcal{S}$, an $\mathcal{A}$-expression $e : n$ over $\mathcal{S}$ evaluates to a relation of arity n, which we denote by $[\![e]\!]^{\mathcal{I}}$, in the well-known manner [23].

Example 2.1 Suppose $\mathcal{S} = \{R : 2, S : 2\}$. Consider the $\mathcal{A}$-expression $e : 2 = \pi_{1,4}\sigma_{2=3}(R \times S)$ over $\mathcal{S}$. For any instance $\mathcal{I}$ of $\mathcal{S}$, which assigns concrete binary relations $\mathcal{I}(R)$ and $\mathcal{I}(S)$ to R and S, the binary relation $[\![e]\!]^{\mathcal{I}}$ equals the composition of $\mathcal{I}(R)$ and $\mathcal{I}(S)$.

2.2 Extending the model

We want to extend the basic relational database model to allow not only data values, but also $\mathcal{A}$-expressions to be stored in relations. Thereto, the simple type system based on arities has to be extended first:

Definition 2.2 A *type* is a tuple $\tau = [\tau_1, \ldots, \tau_n]$, where each τ_i is either the symbol 0, or of the form $\langle m \rangle$, where m is a natural number. In the first case, we say that i is a *data column of τ*; in the second case, we say that i is an *expression column of τ*.

We can now define typed tuples, and relations, containing expressions as follows:

Definition 2.3 Let $\mathcal{S}$ be a schema, and let $\tau = [\tau_1, \ldots, \tau_n]$ be a type. A *tuple of type τ over $\mathcal{S}$* is a tuple $(x_1, \ldots, x_n)$, such that for each $i = 1, \ldots, n$:

- if τ_i is 0 then x_i is a data value (i.e., an element of $\mathbf{V}$).
- if τ_i is $\langle m \rangle$ then x_i is an $\mathcal{A}$-expression over $\mathcal{S}$, of arity m.

A *relation of type τ over $\mathcal{S}$* is a finite set of tuples of type τ over $\mathcal{S}$.

Note that a relation of type $[0, \ldots, 0]$ (n zeros) is an ordinary relation of arity n.

In the kind of systems we intend to model, there will be two kinds of relations. First, we have ordinary relations containing only data values; the schema consisting of the names of these relations is called the *object-level schema*. Second, we have relations containing both data values and $\mathcal{A}$-expressions over the object-level schema; the schema consisting of the names of these relations is then called the *meta-level schema*. Formally:

Definition 2.4
- A *meta-level schema* is a finite set of relation names, where each relation name has an associated type. To denote that a relation name R has type τ we write $R : \tau$.
- Let $\mathcal{M}$ be a meta-level schema, and let $\mathcal{S}$ be a schema disjoint from $\mathcal{M}$ (i.e., having no relation names in common). An *instance of $\mathcal{M}$ over $\mathcal{S}$* is a mapping $\mathcal{J}$ on $\mathcal{M}$ which assigns to each relation name $R : \tau \in \mathcal{M}$ a relation of type τ over $\mathcal{S}$. The pair $(\mathcal{S}, \mathcal{M})$ is called a *combined schema*, in which $\mathcal{S}$ is referred to as the *object-level schema*.
- Finally, an *instance* of a combined schema $(\mathcal{S}, \mathcal{M})$ is simply the union of an instance of $\mathcal{S}$ and an instance of $\mathcal{M}$ over $\mathcal{S}$. We refer to such unions as *combined instances*.

[2] Analogous conservative extension properties are known for complex object databases [16, 25, 22] and spatial databases [15].

Example 2.5 Let $\mathcal{S}$ be the schema of some database which is queried by several users, such as that of a bookstore on the Internet. Queries are represented as $\mathcal{A}$-expressions over $\mathcal{S}$. Suppose we want to monitor the usage made of the database by the users. Then we may want to maintain a meta-level relation *Log* of type $[0, \langle 1\rangle]$, containing pairs (u, q), where u is a username and q is a query u has posed. The expression column $\langle 1\rangle$ indicates that we focus on queries of arity 1; such queries return unary relations (i.e., sets of data values; in an Internet bookstore this will be sets of book records). In this simple example, the object-level schema is $\mathcal{S}$; an instance of $\mathcal{S}$ gives the concrete contents of the relations named in $\mathcal{S}$. The meta-level schema $\mathcal{M}$ contains *Log* (and possibly other meta-level relation names); an instance of $\mathcal{M}$ over $\mathcal{S}$ gives the concrete contents of the relation *Log* (and possibly of others).

3 The relational meta algebra

The relational algebra is a core language for defining queries on ordinary instances. We now want to have a similar formalism for defining queries on combined instances.

First, note that the five operators of the relational algebra can be canonically extended to work on meta-level relations as well as on ordinary, object-level relations. For instance, if $R : [\langle 3\rangle, \langle 3\rangle]$ is the name of a relation storing pairs of expressions of arity 3, we can write $\sigma_{1=2}(R)$ to retrieve those pairs from R with identical first and second components. However, the relational algebra operators do not recognize stored expressions as such; they are treated as abstract data values.

Hence, the five relational algebra operators are a good start, but additional operators are needed. We propose three new operators: `extract`, to extract subexpressions out of stored expressions; `rewrite`, to rewrite (subexpressions of) stored expressions; and `eval`, to dynamically evaluate stored expressions. Adding these three operators to the relational algebra yields what we believe is the functionality one should expect from a core meta query language.

Syntax. We now formally define the expressions of the *relational meta algebra*. Each expression has a type, derived from that of its subexpressions; to denote that expression e has type τ we write $e : \tau$.

Definition 3.1 Fix a combined schema $(\mathcal{S}, \mathcal{M})$. The set $\mathcal{MA}$ of *relational meta algebra expressions over* $(\mathcal{S}, \mathcal{M})$ is the smallest set satisfying:

1. Each relation name $S : n \in \mathcal{S}$ is in $\mathcal{MA}$, and is of type $[0, \ldots, 0]$ (n zeros).
2. Each relation name $R : \tau \in \mathcal{M}$ is in $\mathcal{MA}$.
3. If $e_1 : \tau$ and $e_2 : \tau$ are in $\mathcal{MA}$, then so are $(e_1 \cup e_2) : \tau$ and $(e_1 - e_2) : \tau$.
4. If $e_1 : \tau$ and $e_2 : \omega$ are in $\mathcal{MA}$ with $\tau = [\tau_1, \ldots, \tau_n]$ and $\omega = [\omega_1, \ldots, \omega_m]$, then so is $(e_1 \times e_2) : [\tau_1, \ldots, \tau_n, \omega_1, \ldots, \omega_m]$.
5. If $e : \tau$ is in $\mathcal{MA}$ with $\tau = [\tau_1, \ldots, \tau_n]$, then so are
   - $\sigma_{i=j}(e) : \tau$, where $i, j \in \{1, \ldots, n\}$ such that $\tau_i = \tau_j$; and
   - $\pi_{i_1,\ldots,i_p}(e) : [\tau_{i_1}, \ldots, \tau_{i_p}]$, where $i_1, \ldots, i_p \in \{1, \ldots, n\}$.
6. For each $\mathcal{A}$-expression $e : n$ over $\mathcal{S}$, $\langle e\rangle : [\langle n\rangle]$ is in $\mathcal{MA}$.
7. If $e : \tau$ is in $\mathcal{MA}$ with $\tau = [\tau_1, \ldots, \tau_n]$ and i is an expression column[3] of τ, then the following expressions are also in $\mathcal{MA}$:
   - $\texttt{extract}_{i:m}(e) : [\tau_1, \ldots, \tau_n, \langle m\rangle]$, where m is a natural number;
   - $\texttt{rewrite-one}_{i:\alpha\to\beta}(e)$ and $\texttt{rewrite-all}_{i:\alpha\to\beta}(e)$, both of type $[\tau_1, \ldots, \tau_n, \tau_i]$, where $\alpha \to \beta$ is a *rewrite rule over* $\mathcal{S}$ *with respect to* τ (to be defined shortly); and
   - $\texttt{eval}_i(e) : [\tau_1, \ldots, \tau_n, 0, \ldots, 0]$ (ℓ zeros), where ℓ is given by $\tau_i = \langle \ell\rangle$.

Rewrite rules. To finish the above definition we need to define the system of rewrite rules on which the `rewrite` operators are based. Thereto the classical notion of a term rewrite rule [11] must be adapted to our setting.

Let $\mathcal{S}$ be a schema and let $\tau = [\tau_1, \ldots, \tau_n]$ be a type. Let $C \subseteq \{1, \ldots, n\}$ be the set of expression columns of τ, and for $j \in C$ let ℓ_j be given by $\tau_j = \langle \ell_j\rangle$.

Definition 3.2 A *rewrite rule over* $\mathcal{S}$ *with respect to* τ is a rule of the form $\alpha \to \beta$, where α and β are $\mathcal{A}$-expressions of the same arity, over the augmented schema $\mathcal{S} \cup \{\Box_j \mid j \in C\}$. Here, each $\Box_j$ is an *expression variable of arity* ℓ_j.

An expression variable is formally nothing but a specially reserved relation name of arity ℓ_j; intuitively it should be thought of as a placeholder for subexpressions of arity ℓ_j.

Semantics of $\mathcal{MA}$. In the context of a given combined instance $\mathcal{K}$ of $(\mathcal{S}, \mathcal{M})$, an $\mathcal{MA}$-expression $e : \tau$ over $(\mathcal{S}, \mathcal{M})$ evaluates to a relation $[\![e]\!]^{\mathcal{K}}$ of type τ. We only define $[\![e]\!]^{\mathcal{K}}$ for cases 6 and 7 of Definition 3.1; the first 5 cases are completely analogous to the semantics of the standard relational algebra.

- $[\![\langle e\rangle]\!]^{\mathcal{K}} := \{(e)\}$, for an $\mathcal{A}$-expression e.
- $[\![\texttt{extract}_{i:m}(e)]\!]^{\mathcal{K}} := \{(x_1, \ldots, x_n, x) \mid (x_1, \ldots, x_n) \in [\![e]\!]^{\mathcal{K}}$ and x is a subexpression[4] of x_i that is of arity $m\}$.
- $[\![\texttt{rewrite-one}_{i:\alpha\to\beta}(e)]\!]^{\mathcal{K}} := \{(x_1, \ldots, x_n, x) \mid (x_1, \ldots, x_n) \in [\![e]\!]^{\mathcal{K}}$ and x is obtained from x_i by replacing *one* occurrence of $f(\alpha)$ as a subexpression in x_i by $f(\beta)\}$. Here f is the mapping on the expression variables occurring in the rewrite rule defined by $f(\Box_j) := x_j$.
- $[\![\texttt{rewrite-all}_{i:\alpha\to\beta}(e)]\!]^{\mathcal{K}}$ is defined similarly, but now *every* occurrence of $f(\alpha)$ in x_i is replaced by $f(\beta)$.
- $[\![\texttt{eval}_i(e)]\!]^{\mathcal{K}} := \{(x_1, \ldots, x_n, y_1, \ldots, y_\ell) \mid (x_1, \ldots, x_n) \in [\![e]\!]^{\mathcal{K}}$ and $(y_1, \ldots, y_\ell) \in [\![x_i]\!]^{\mathcal{I}}\}$.

So, an $\mathcal{MA}$-expression $e : \tau$ over $(\mathcal{S}, \mathcal{M})$ defines a mapping $[\![e]\!]$ from the set of combined instances of $(\mathcal{S}, \mathcal{M})$ to the set of relations of type τ. Such a mapping is called a *query over* $(\mathcal{S}, \mathcal{M})$ *of type* τ.

[3] Recall Definition 2.2 for the notion of expression column.

[4] By *subexpression* we mean direct *and* indirect ones. So the subexpressions of $\pi_{1,4}\sigma_{2=3}(R \times S)$ are the expression itself; $\sigma_{2=3}(R \times S)$; $R \times S$; R; and S.

Examples. We next illustrate the meta algebra by means of two examples. Illustrations of the working of individual operators, complete with input and output, have been placed in an Appendix.

The first example is an illustration of the kind of syntactical manipulations on stored expressions that are possible. The second illustrates the use of `eval` to interpret stored expressions semantically.

Recall the meta-level relation *Log* of type $[0, \langle 1\rangle]$ from Example 2.5. We want to compute the query q of type $[\langle 1\rangle, \langle 4\rangle]$ defined as follows: given a combined instance $\mathcal{K}$, $q(\mathcal{K})$ is the set of all pairs (x, y) such that x is a stored expression in $\mathcal{K}(Log)$, i.e., $x \in [\![\pi_2(Log)]\!]^{\mathcal{K}}$, and y is a subexpression of x occurring at least twice in x. The naive attempt

$$\pi_{2,3}\sigma_{3=4}\texttt{extract}_{2:4}\texttt{extract}_{2:4}(Log)$$

is incorrect; to distinguish different occurrences of the same subexpression we have to mark them in some way. This can be done using `rewrite-one`. Assume we have some dummy relation name $D \in \mathcal{S}$ of arity 0. An occurrence x of a subexpression can be marked by rewriting it into $x \times D$. So if **mark** is the following $\mathcal{MA}$-expression:

$$\pi_{2,3,4}\texttt{rewrite-one}_{2:\square_3 \to \square_3 \times D}\texttt{extract}_{2:4}(Log),$$

then the wanted query q is defined by the $\mathcal{MA}$-expression

$$\pi_{1,2}\sigma_{3\neq 6}\sigma_{2=5}\sigma_{1=4}(\mathbf{mark} \times \mathbf{mark}).$$

For the second example, assume for convenience that the object-level schema consists of one single relation name S of arity, say, 5. If we want to see for every user u the results of all queries posed by u (as recorded in *Log*) evaluated on the current instance, we simply write

$$\pi_{1,3}\texttt{eval}_2(Log).$$

Now suppose we are given a meta-level relation U of type $[\langle 5\rangle]$ containing $\mathcal{A}$-expressions to be interpreted as possible new contents for relation S (the letter 'U' stands for 'update'). So given a combined instance $\mathcal{K}$ each $x \in \mathcal{K}(U)$ stands for a potential update from $\mathcal{K}(S)$ to $[\![x]\!]^{\mathcal{K}}$. Then we may want to compute the query q of type $[0, 0, \langle 5\rangle]$ defined as follows: $q(\mathcal{K})$ is the set of all tuples (u, v, x) such that v is in the result of a query posed by u, evaluated not on $\mathcal{K}(S)$ but on its update as given by x. To do this we use the `rewrite-all` operator as follows:

$$\pi_{1,5,3}\texttt{eval}_4\texttt{rewrite-all}_{2:S\to\square_3}(Log \times U).$$

4 Expressive power of $\mathcal{MA}$

In this section we investigate the expressive power of our formalism. Due to space limitations, proofs of theorems will only be sketched.

4.1 Non-redundancy and conservative extension

A natural question to ask is whether $\mathcal{MA}$ is non-redundant, i.e., whether each operator provided in $\mathcal{MA}$ is primitive (not definable using the other operators).

Theorem 4.1 *$\mathcal{MA}$ is not redundant.*

The most interesting case is that of `eval`, which is based on the following lemma (proof omitted):

Lemma 4.2 *Assume every meta-level relation is of a type having only expression columns. Then every $\mathcal{MA}$-expression that does not use* `eval` *is equivalent, up to reordering of columns,*[5] *to a union of $\mathcal{MA}$-expressions of the form $e_1 \times e_2$, where e_1 is an $\mathcal{A}$-expression and e_2 is an $\mathcal{MA}$-expression of a type having only expression columns.*

To see how primitivity of `eval` follows from this lemma, let $\mathcal{S} = \{S : 1\}$ and $\mathcal{M} = \{R : [\langle 1\rangle]\}$. Assume, for the sake of contradiction, that the $\mathcal{MA}$-expression $\pi_2\texttt{eval}_1(R)$, of type $[0]$, is expressible in $\mathcal{MA}$ without `eval`. Since its type has no expression columns, by the lemma it then is even equivalent to an $\mathcal{A}$-expression, say e. Now take any instance $\mathcal{K}$ of $\mathcal{S}$ such that $\mathcal{K}(S) \neq \emptyset$, and take the $\mathcal{A}$-expression $e' := S - e$. Then $[\![e]\!]^{\mathcal{K}} \neq [\![e']\!]^{\mathcal{K}}$. Extend $\mathcal{K}$ to a combined instance by putting $\mathcal{K}(R) := \{(e')\}$. Then

$$[\![\pi_2\texttt{eval}_1(R)]\!]^{\mathcal{K}} = [\![e']\!]^{\mathcal{K}} \neq [\![e]\!]^{\mathcal{K}},$$

contradicting our assumption that e is equivalent to $\pi_2\texttt{eval}_1(R)$.

We omit the proofs of primitivity for `extract` and the `rewrite` operators. Regarding primitivity of the five relational algebra operators: it is well known (e.g., [5]) that each of them is primitive within $\mathcal{A}$; of course this does not automatically imply primitivity within $\mathcal{MA}$. The latter follows nevertheless because we have the following *conservative extension property*:

Theorem 4.3 *Let $\mathcal{S}$ be a schema and let q be a query over $(\mathcal{S}, \emptyset)$ of type $[0, \ldots, 0]$ (n zeros). If q is definable in $\mathcal{MA}$ then q is already definable in $\mathcal{A}$.*

To paraphrase, $\mathcal{MA}$ provides no power above that of $\mathcal{A}$ if only classical queries not involving meta-level relations are under consideration. The theorem can be proven by observing that if there are only object-level relations, the set of expressions that can appear in the evaluation of a fixed $\mathcal{MA}$-expression e on any instance is finite. Using this observation, we can show that if the meta-level schema is empty, `eval` can be eliminated from $\mathcal{MA}$-expressions. It then suffices to apply Lemma 4.2.

4.2 An equivalent calculus

Codd's classical theorem [8] says that the queries expressible in the relational algebra are precisely the queries definable in first-order logic (in this context referred to as the *relational calculus*). We next indicate how this equivalence can be extended to the meta algebra by introducing $\mathcal{MC}$, the *relational meta calculus*.

Fix a combined schema $(\mathcal{S}, \mathcal{M})$. Our calculus uses two kinds of variables: *data variables* and *expression variables*. Data variables will range over $\mathbf{V}$ (the universe of data values). Expression variables were already used in the rewrite rules of $\mathcal{MA}$; they have an associated arity and range over the $\mathcal{A}$-expressions of that arity.

A *term* is either a data variable, in which case it is said to be *of sort* 0, or an $\mathcal{A}$-expression over the augmentation of $\mathcal{S}$ with a finite set of expression variables, in which case it is said to be *of sort* $\langle n\rangle$, where n is the arity of the expression.

Atomic formulas can be of one of the following forms: $S(x_1, \ldots, x_n)$, where $S : n \in \mathcal{S}$ and each x_i is a data variable; $R(t_1, \ldots, t_n)$, where $R : [\tau_1, \ldots, \tau_n] \in \mathcal{M}$ and each t_i is a term of sort τ_i; $t_1 = t_2$ and $t_1 \leq t_2$, where t_1 and t_2

[5] Note that reordering of columns is expressible using projection.

are terms of the same sort; $\texttt{rewrite-one}(t_1, t_2, t_3, t_4)$ and $\texttt{rewrite-all}(t_1, t_2, t_3, t_4)$, where $t_1, \ldots, t_4$ are terms such that t_1 and t_4 have the same sort, and t_2 and t_3 have the same sort; $\texttt{eval}(t, x_1, \ldots, x_n)$, where t is a term of sort $\langle n \rangle$ and $x_1, \ldots, x_n$ are data variables.

Formulas, finally, are built from atomic formulas in the standard manner using Boolean connectives and quantifiers. The set of all formulas is denoted by $\mathcal{MC}$.

Given an $\mathcal{MC}$-formula φ, a combined instance $\mathcal{K}$ of $(\mathcal{S}, \mathcal{M})$, and a valuation ρ of the free variables of φ, the truth of φ in $\mathcal{K}$ under ρ, denoted by $\mathcal{K} \models \varphi[\rho]$, is defined in the standard way given the following semantics for the above predicates: $t_1 \leq t_2$ means that t_1 is a subexpression of t_2; $\texttt{rewrite-one}(t_1, t_2, t_3, t_4)$, respectively $\texttt{rewrite-all}(t_1, t_2, t_3, t_4)$, means that t_4 is obtained from t_1 by replacing one, respectively every, occurrence of t_2 in t_1 by t_3; and $\texttt{eval}(t, x_1, \ldots, x_n)$ means that $(x_1, \ldots, x_n)$ is in the result of evaluating t.

An $\mathcal{MC}$-formula φ with free variables $x_1, \ldots, x_n$ of sorts $\tau_1, \ldots, \tau_n$, respectively, defines the query q of type $[\tau_1, \ldots, \tau_n]$ defined by $q(\mathcal{K}) = \{(\rho(x_1), \ldots, \rho(x_n)) \mid \mathcal{K} \models \varphi[\rho]\}$. Of course this is only well-defined if $q(\mathcal{K})$ is finite for every $\mathcal{K}$. However, a syntactical restriction called *safety* can be put on $\mathcal{MC}$-formulas such that finiteness is guaranteed. Our notion of safety is a natural extension of the well-known notion for the classical relational calculus (see [23]) to our setting.

Specifically, we call an $\mathcal{MC}$-formula *safe* if it does not contain $\forall$; every variable is quantified only once; any subformula of the form $\varphi \vee \psi$ is such that φ and ψ have the same free variables; and in any maximal conjunctive subformula, all free variables are limited. Here a variable is said to be *limited* if it occurs in a conjunct of one of the following forms:

- $R(\ldots)$, with R a relation name;
- $t_1 = t_2$, where either all variables occurring in t_1 or all in t_2 are limited;
- $t_1 \leq t_2$, where all variables occurring in t_2 are limited;
- $\texttt{rewrite-one}(t_1, t_2, t_3, t_4)$ or $\texttt{rewrite-all}(t_1, t_2, t_3, t_4)$, where all variables occurring in t_1, t_2 and t_3 are limited;
- $\texttt{eval}(t, x_1, \ldots, x_n)$, where all variables occurring in t are limited.

Example 4.4 Let $R : [\langle 1 \rangle] \in \mathcal{M}$. Let q be the query of type $[\langle 1 \rangle]$ defined as follows: given an instance $\mathcal{K}$, $q(\mathcal{K})$ is the set of expressions in $\mathcal{K}(R)$ having a subexpression of the form $e \cup \pi_{1,4}\sigma_{2=3}(e \times e')$, where e is any expression of arity 2 and e' is any expression of arity 3. Such a "pattern matching query" can be naturally defined by the following safe $\mathcal{MC}$-formula:

$$R(x) \wedge (\exists x_1)(\exists x_2) x_1 \cup \pi_{1,4}\sigma_{2,3}(x_1 \times x_2) \leq x,$$

where x is of sort $\langle 1 \rangle$, x_1 of sort $\langle 2 \rangle$, and x_2 of sort $\langle 3 \rangle$. ■

We establish:

Theorem 4.5 *The class of queries definable in $\mathcal{MA}$ coincides with the class of queries definable by safe $\mathcal{MC}$-formulas.*

To illustrate how this Theorem can be proven, we show how the $\mathcal{MC}$-formula from Example 4.4 can be translated in $\mathcal{MA}$. We begin by "flattening" the formula a bit:

$$R(x) \wedge (\exists y)(y \leq x \wedge (\exists x_1)(\exists x_2) y = x_1 \cup \pi_{1,4}\sigma_{2,3}(x_1 \times x_2)).$$

Note that x is limited by $R(x)$; y is limited by $y \leq x$; and x_1 and x_2 are limited by $y = x_1 \cup \pi_{1,4}\sigma_{2,3}(x_1 \times x_2)$. An equivalent $\mathcal{MA}$-expression is

$$\pi_1\sigma_{5=2}\texttt{rewrite-one}_{3:\square_3 \to \square_3 \cup \pi_{1,4}\sigma_{2=3}(\square_3 \times \square_4)} \texttt{extract}_{2:3}\texttt{extract}_{2:2}\texttt{extract}_{1:2}(R).$$

Note how the order in which variables can be proven to be limited determines the order in which the operators are applied. Starting from R, which produces values for variable x, we extract values for y, and from there we extract values for x_1 and x_2. Then we rewrite the column for x_1 into $x_1 \cup \pi_{1,4}\sigma_{2=3}(x_1 \times x_2)$. Finally, we compare the result of the rewriting to the column for y and project on the result variable x.

4.3 Limitations of the typed approach

$\mathcal{MA}$ and $\mathcal{MC}$ are strictly typed formalisms. It is impossible to define relations with columns containing expressions of different arities. However, we can give an example of a natural and simple query that seems to have the property that computing it really requires such untyped intermediate results:

Theorem 4.6 *Let $R : [\langle 1 \rangle] \in \mathcal{M}$. Let q be the query of type $[\langle 1 \rangle]$ defined as follows: given an instance $\mathcal{K}$, $q(\mathcal{K})$ is the set of expressions in $\mathcal{K}(R)$ that are of the form $\pi_1(\ldots)$. This query is not definable in $\mathcal{MA}$.*

The equivalence of $\mathcal{MA}$ with $\mathcal{MC}$ allows an elegant model-theoretic proof of this theorem, which we sketch next. The $\mathcal{A}$-expressions over a schema $\mathcal{S}$ form a structure (in the sense of mathematical logic [9]) consisting of the relation names in $\mathcal{S}$ as constants, the operators as functions, and the relations $\leq$ (subexpression), `rewrite-one`, and `rewrite-all`. This structure is many-sorted: for example, we do not have one single function $\times$ but rather have a separate one $\times_{n,m}$ of sort $(\langle n \rangle, \langle m \rangle) \to \langle n + m \rangle$ for all arities n and m.

Now suppose, for the sake of contradiction, that there is an $\mathcal{MC}$-formula φ defining the query q from the theorem. Since the query is independent of the object-level instance we can as well assume that all object-level relations are empty. Hence, we may assume without loss of generality that φ neither uses data variables, object-level relation names, nor `eval`.

So φ is essentially a first-order logic formula, evaluated over the above-described structure of $\mathcal{A}$-expressions, call it $\mathcal{E}$, expanded with a relation R of sort $\langle 1 \rangle$. Let n be strictly larger than the arity of any term occurring in φ. Then φ looks only at $\mathcal{E}|_{<n}$, the restriction of $\mathcal{E}$ to sorts $\langle m \rangle$ with $m < n$.

Define the following function f on $\mathcal{A}$-expressions e: $f(e)$ is obtained from e by replacing each occurrence of a subexpression of the form $\pi_1(e')$, where e' is n-ary, by $\pi_2(e')$, and conversely, replacing each occurrence of a subexpression of the form $\pi_2(e')$, where e' is n-ary, by $\pi_1(e')$. This function is an automorphism of $\mathcal{E}|_{<n}$. It maps $\pi_1(S^n)$ to $\pi_2(S^n)$ and back, where S^n stands for $S \times \cdots \times S$ (n times).

Hence, on an instance in which R consists of the two expressions $\pi_1(S^n)$ and $\pi_2(S^n)$, the query defined by φ will either contain both expressions in the result, or none of them, since first-order logic formulas cannot distinguish between automorphic elements. This yields the desired contradiction, since $\pi_2(S^n)$ is not of the form $\pi_1(\ldots)$.

5 Discussion

We have presented typed query languages for databases that contain, besides ordinary data values, also queries. Theorem 4.6 offers the most challenging direction for further research. How can our formalism (in particular its type system) be generalized so that queries of the kind mentioned in the theorem become expressible, *at the same time* not giving up on type-safety of `eval`?

Note that Theorem 4.6 may be compared to a similar situation in the design of computationally complete query languages. The language QL, proposed and studied by Chandra and Harel [5], is an adaptation of the relational algebra designed to work with "untyped" relations of variable width, to which a while-loop construct is added. QL is computationally complete. If, however, the ordinary "typed" relational algebra is extended with while-loops, one gets a language whose expressiveness remains within PSPACE [6, 2].

Another situation to which Theorem 4.6 may be compared to is that of the lambda calculus. Functions on the natural numbers, encoded as functions on Church numerals, are typed. But again the computation of many such functions requires intermediate results that are untyped: in the untyped lambda calculus all partial recursive functions are definable, while in the simply-typed lambda calculus only a restricted class of functions, the so-called extended polynomials, are definable [3, 18].

Two other obvious directions for further research left open by our work is *(i)* to experiment with how our model for typed meta database programming can be applied in practice; and *(ii)* to better understand the precise expressive power of $\mathcal{MA}$. Concerning *(i)*, it could be interesting to try to integrate our model into the SQL3 or OQL context. Concerning *(ii)*, a concrete open problem is whether or not the query "give all expressions of maximal length stored in relation R" is expressible in $\mathcal{MA}$.

A natural direction for extending $\mathcal{MA}$ would be to allow for data to be moved between the data columns and the expression columns of a relation. Such a functionality could be achieved by considering constant relations as expressions. The algebra could then be extended with a `wrap` operation for turning relations (or subrelations obtained by a group-by-like operation) into constant relations, and an inverse `unwrap` operation for extracting the contents of constant relations. The potential of this functionality has yet to be investigated.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. *The VLDB Journal*, 4(4):727–794, 1995. Originally INRIA Research Report 846, 1988.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] H.P. Barendregt. *The Lambda Calculus*. North-Holland, 1984.

[4] C. Beeri and T. Milo. On the power of algebras with recursion. In *Proceedings of the 1993 ACM SIGMOD Internation Conference on Management of Data*, volume 22:2 of *SIGMOD Record*, pages 377–386. ACM Press, 1993.

[5] A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

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

[7] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[8] E. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, 1972.

[9] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[10] M. Gyssens, J. Paredaens, and D. Van Gucht. A grammar-based approach towards unifying hierarchical data models. *SIAM Journal on Computing*, 23(6):1093–1137, 1994.

[11] J.-W. Klop. Term rewriting systems: A tutorial. *Bulletin of the EATCS*, 32:143–183, 1987.

[12] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[13] L.V.S. Lakshmanan, F. Sadri, and I.N. Subramanian. On the logical foundations of schema integration and evolution in heterogeneous database systems. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Deductive and Object-Oriented Databases*, volume 760 of *Lecture Notes in Computer Science*, pages 81–100. Springer-Verlag, 1993.

[14] G. Ozsoyoglu, Z.M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Transactions on Database Systems*, 12(4):566–592, 1987.

[15] J. Paredaens, J. Van den Bussche, and D. Van Gucht. Towards a theory of spatial database queries. In *Proceedings 13th ACM Symposium on Principles of Database Systems*, pages 279–288. ACM Press, 1994.

[16] J. Paredaens and D. Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Transactions on Database Systems*, 17(1):65–93, 1992.

[17] K. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings 11th ACM Symposium on Principles of Database Systems*, pages 346–353, 1992.

[18] H. Schwichtenberg. Definierbare Funktionen im λ-Kalkül mit Typen. *Arch. Math. Logik Grundlagenforsch.*, 17(3–4):113–114, 1975.

[19] T. Sheard and J. Hook. Type safe meta-programming. Manuscript, Oregon Graduate Institute, 1994.

[20] M. Stonebraker et al. QUEL as a data type. In B. Yormark, editor, *Proceedings of SIGMOD 84 Annual Meeting*, volume 14:2 of *SIGMOD Record*, pages 208–214. ACM Press, 1984.

[21] M. Stonebraker et al. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, 1987.

[22] D. Suciu. Bounded fixpoints for complex objects. *Theoretical Computer Science*, 176(1–2):283–328, 1997.

[23] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.

[24] J. Van den Bussche, D. Van Gucht, and G. Vossen. Reflective programming in the relational algebra. *Journal of Computer and System Sciences*, 52(3):537–549, June 1996.

[25] L. Wong. Normal forms and conservative extension properties for query languages over collection types. *Journal of Computer and System Sciences*, 52(3):495–505, 1996.

Appendix

We give simple illustrations of the semantics of the new operators of $\mathcal{MA}$ using the following example: Suppose $\mathcal{S} = \{S : 2, T : 2, U : 2\}$ and $\mathcal{M} = \{R : [0, \langle 4\rangle, 0, \langle 2\rangle]\}$. Figure 1 shows an instance of R, followed by the results of

1. $\texttt{extract}_{2:4}(R)$, obtained by extracting from column 2 all subexpressions of arity 4;

2. $\texttt{rewrite-one}_{2:\square_4 \to S}(R)$, obtained by rewriting every one occurrence of an expression from column 4 in column 2 by S;

3. $\texttt{rewrite-all}_{2:\square_4 \to S}(R)$, obtained by rewriting all occurrences of an expression from column 4 in column 2 simultaneously by S; and

4. $\texttt{eval}_2(R)$, obtained by evaluating the expressions in column 2 of R on the following instances of relations S and T:

S	
x	y
x	x

T	
u	u
u	x

(For the purpose of this example there is no need to give an instance of U, since U does not occur in the second column of our example relation R.)

Instance of R:

a	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$	d	T
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T

$\texttt{extract}_{2:4}(R)$:

a	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$	d	T	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	$\sigma_{1=4}(S \times T) \cup (S \times S)$
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	$\sigma_{1=4}(S \times T)$
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	$S \times S$
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$

$\texttt{rewrite-one}_{2:\square_4 \to S}(R)$:

a	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$	d	T	$\sigma_{1=2}(S) \times \sigma_{1=2}(S)$
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times S \times S \times T)$
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times S)$

$\texttt{rewrite-all}_{2:\square_4 \to S}(R)$:

a	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$	d	T	$\sigma_{1=2}(S) \times \sigma_{1=2}(S)$
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times S \times S \times S)$

$\texttt{eval}_2(R)$:

a	$\sigma_{1=2}(S) \times \sigma_{1=2}(T)$	d	T	x	x	u	u
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	y	u	x
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	x	u	x
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	y	x	y
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	y	x	x
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	x	x	y
b	$\sigma_{1=4}(S \times T) \cup (S \times S)$	e	U	x	x	x	x
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	x	y	u	x
c	$\pi_{1,2,3,4}\sigma_{2=6}\sigma_{4=5}(S \times T \times S \times T)$	f	T	x	x	u	x

Figure 1: Examples of the novel $\mathcal{MA}$ operators.