Reflective programming in the relational algebra

(extended abstract)

Jan Van den Bussche[*] University of Antwerp

Dirk Van Gucht[†] Indiana University

Gottfried Vossen[‡] Universität Giessen

Abstract

In reflective programming languages it is possible for a program to generate code that is integrated into the program's own execution. We introduce a reflective version of the relational algebra. Reflection is achieved by storing and manipulating relational algebra programs as relations in the database. We then study the expressibility and complexity of the reflective algebra thus obtained. It turns out that there is a close correspondence between reflection and bounded looping. We also discuss the applicability of the reflective algebra.

1 Introduction

The concept of *reflection* has received attention in different areas of computer science. It provides powerful tools to solve problems in, e.g., the construction of interpreters [AR89], polymorphism [S$^+$90], Logic, and A.I. [MN88]. In this paper we will be only interested in *linguistic* reflection [S$^+$92]. A programming language offers this type of reflection if it provides special constructs that allow a program to generate code that is integrated into the program's own execution. The meta-programming constructs provided in various languages for logic or functional programming (e.g., `eval` in Lisp and `clause` in Prolog) are a simple and good example.

The essence of reflection is that programs can be treated as data and vice versa. This idea is of course as old as the concept of computation itself, dating back to the Universal Turing Machine and the von Neumann architecture. For example, it is straightforward to write an interpreter for Pascal programs in Pascal. Indeed, adding reflective features to a computationally complete language will not enhance its expressive power (although of course it will allow a more succinct expression of certain constructions.) This is no longer true, however, if we work with languages that are *not* computationally complete. In this paper, we will study reflective programming in the context of the relational algebra. Programs in the relational algebra express queries, and it is well-known that the relational algebra cannot express all computable queries. Examples of queries that cannot be expressed are transitive closure and parity.

The key observation on which our work is based is that it is not difficult to define a format in which relational algebra programs can be stored into relations [SVGG93]. Reflective capabilities can thus be added to the relational algebra simply by providing an *evaluation* operator, which executes its argument relation containing a program. Importantly, relations containing programs can be created and manipulated in just the same way as relations containing ordinary data. In particular, program relations can be constructed by means of relational algebra computations taking ordinary relations as input. By making novel use of *identifier creation* mechanisms known from object-oriented query languages, this turns out to be possible in an effective and convenient way.

[*]Research Assistant of the NFWO. Address: Dept. Math. & Computer Sci., University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerp, Belgium. Email: vdbuss@wins.uia.ac.be

[†]Computer Sci. Dept., Indiana University, Bloomington, IN 47405-4101, USA. Email: vgucht@cs.indiana.edu

[‡]Fachbereich Mathematik, Arbeitsgruppe Informatik, Universität Giessen, Arndtstrasse 2, D-6300 Giessen, Germany. Email: vossen@informatik.rwth-aachen.de

In the reflective relational algebra thus obtained, one can create and manipulate programs in combination with ordinary data, and execute them dynamically. Since it is thus possible to specify programs, the structure or length of which depends on the size of the database, reflection yields an increase in expressive power. Specifically, we show that using reflection, *bounded looping* can be simulated. With bounded looping, one can express a variety of queries that are not expressible in the relational algebra [Cha81]. For example, all fixpoint queries (in particular transitive closure) and parity can be expressed.

We also prove a converse of the above-stated result. Specifically, we show that every "ordinary" query, expressed by a program in the reflective algebra, can be expressed by a bounded loop program. So, in a sense that will be made precise, adding reflection to the relational algebra is equivalent to adding bounded looping. This close correspondence between reflection and bounded looping will also allow to determine the computational complexity of the reflective algebra. The notion of *expression complexity* [Var82] will be relevant in this respect.

In the initial stages of this investigation, we took inspiration from recent work by Ross [Ros92] on an algebra for first-order logic with higher-order syntax. He extended the relational algebra to deal with relations containing relation names. Since relation names are special cases of relational algebra programs, Ross's algebra is a special case of the reflective algebra.

To our knowledge, Stonebraker et al. [S+84, S+87] were the first to (informally) investigate the treatment of programs as data in the field of databases. They argued that a wide range of applications can benefit from procedural data. All these applications can easily be formalized in the reflective algebra. Our model can also be helpful in the related, but more recent issue of manipulation of methods in object-oriented database systems.

This extended abstract is organized as follows. In Section 2, we introduce our model for reflective programming in the relational algebra. In Section 3, we establish the connection between reflection and bounded looping, and determine the complexity of the reflective algebra. In Section 4, we discuss the applicability of reflection in the context of data dictionaries, Ross's work, the work of Stonebraker et al., and method manipulation in object-oriented database systems.

2 Extending the relational algebra with reflection

In this section, we define a query language based on the relational algebra, and show how programs in this language can be stored in relations. We then introduce the reflective *evaluation* operation.

We will use the following version of the relational database model. We assume disjoint, countably infinite sets of *attribute names* and *relation names.* A *relation scheme* is a finite set of attribute names. A *database scheme* $\mathcal{S}$ is a finite set of relation names, in which each relation name R has an associated relation scheme $\tau(R)$.

We further assume that each attribute name A has an associated *domain*, $dom(A)$, of data elements (also called constants). Given a relation scheme τ, a *tuple* over τ is a mapping t on τ such that, for each $A \in \tau$, $t(A) \in dom(A)$. A *relation* over τ is a finite set of tuples over τ. Finally, given a database scheme $\mathcal{S}$, a *database instance* over $\mathcal{S}$ is a mapping I on $\mathcal{S}$ such that for each $R \in \mathcal{S}$, $I(R)$ is a relation over $\tau(R)$.

We will use the relational algebra-based query language $\mathcal{A}$ defined in [Cha81]. *Programs* in $\mathcal{A}$ are sequences of *statements.* Each statement has the form '$X := E$', where X is a *variable* and E is a *term.* Each variable X has an associated relation scheme $\tau(X)$, and can take relations over $\tau(X)$ as values. Each term is one of the following. A relation name (taken from an arbitrary but fixed database scheme $\mathcal{S}$) is a term; a constant one-attribute one-tuple relation, of the form $\{(A : a)\}$ where A is an attribute name and $a \in dom(A)$, is a term; a variable is a term; if X, X_1, X_2 are variables, then $X_1 \cup X_2$ is a term (union), and so are $X_1 - X_2$ (difference), $X_1 \bowtie X_2$ (join), $\pi_A(X)$ (projecting out attribute A), $\sigma_{A=B}(X)$ (selection for equality of attributes A and B), and $\rho_{A/A'}(X)$ (renaming of attribute A to A').

The semantics of $\mathcal{A}$ programs is obvious. In particular, when we apply a program to a database instance I over $\mathcal{S}$, the relation names take their values from I, the variables are initialized to the

1) $X_1 := R$;
2) $X_2 := \{(P : \text{Fred})\}$;
3) $X_3 := X_1 \bowtie X_2$;
4) $X_4 := \rho_{C/C'}(X_3)$;
5) $X_5 := \rho_{P/C'}(X_1)$;
6) $X_6 := X_4 \bowtie X_5$;
7) $X_7 := \pi_{C'}(X_6)$;
8) $X_3 := X_3 \cup X_7$;
9) $X_8 := \pi_P(X_3)$.

Figure 1: Program of Example 2.1

empty relation, and the statements are executed in order. Since terms are typed (i.e., have an associated relation scheme), typing errors can occur, in which case the term evaluates to the empty relation by default. The final result of the program execution is, again by default, the value of the last statement.

Example 2.1 Consider a database containing a parent-child relation R with attributes $\{P, C\}$. The program shown in Figure 1 computes, in variable X_8, the set of all children and grandchildren of Fred. The statements are numbered for easy reference.

In Lisp, reflective programming is facilitated by the uniform format in which both programs and data are represented, namely lists. Lisp has the reflective `eval` operator which takes a list as argument and executes this list, interpreted as a program. If we define a format in which programs can be stored as relations, we can define an analogous reflective operation for the relational algebra.

Example 2.2 The program shown in Figure 1 can be stored in a relation over the attributes $\{$*id*, *next-id*, *var*, *op*, *att-1*, *att-2*, *arg-1*, *arg-2*, *rel*, *const*$\}$, as shown in Figure 2. There is a tuple for each statement, containing, where applicable, the assigned-to variable *var*, the algebra operator *op*, the possible attribute parameter(s) *att-1*, *att-2* of the operator, and the argument(s) *arg-1*, *arg-2*. Note how statements 1 and 2, which have no operator, have their own encoding format. Finally, every statement has an identifier and a pointer to the identifier of the next statement. Non-applicable entries are filled with a special blank symbol ‘ ’.

Inspired by the above example, we call the relation scheme $\{$*id*, *next-id*, *var*, *op*, *att-1*, *att-2*, *arg-1*, *arg-2*, *rel*, *const*$\}$ the *program scheme*. Relations over the program scheme can be used to store $\mathcal{A}$ programs. The domain of the identifier attributes *id* and *next-id* is equal to $\mathcal{D}^{\text{id}}$, a countably infinite universe of identifier values. We will often use natural numbers for this purpose. Furthermore, *dom*(*op*) equals $\{\cup, -, \bowtie, \pi, \rho, \sigma\}$; *dom*(*att-1*) and *dom*(*att-2*) equal the set of attribute names; *dom*(*var*), *dom*(*arg-1*), and *dom*(*arg-2*) equal the set of variables; and *dom*(*rel*) and *dom*(*const*) equal the set of relation names and the set of constants, respectively. To be entirely correct, the blank symbol should be added to all of the above domains. We will refer to these domains, with the exception of the domain *dom*(*const*) of constants, as the *lexical* domains.

We are now ready to define the *evaluation* operator, leading to the *reflective algebra* $\mathcal{RA}$. Recall the definition of syntax and semantics of $\mathcal{A}$ given in the beginning of this section.

Definition 2.3 A term of $\mathcal{RA}$ is either a term of $\mathcal{A}$, or an expression of the form ‘**eval**(X)’, where X is a variable such that $\tau(X)$ is the program scheme. The semantics of the evaluation operator is the following. If r is a program relation containing an $\mathcal{A}$ program P, then **eval**(r) equals the result of executing P. If r does not contain a syntactically correct $\mathcal{A}$ program, **eval**(r) equals $\emptyset$ by default.

Importantly, relations containing programs need not be stored in advance in the database. Indeed, the evaluation operator applies to an arbitrary variable. Hence, program relations can be constructed just like any other relation, using $\mathcal{A}$ statements. Programs used to construct program relations will be called *meta-programs*. For example, it is straightforward to write an $\mathcal{A}$ meta-program that assembles the program relation shown in Figure 2 from the constant relations $\{(id, 1)\}$, $\{(var, X_4)\}$, $\{(op, \bowtie)\}$, etc. However, if identifier values are only drawn from a priori given constants, only program relations with length bounded by a constant (i.e., the number of different identifier constants appearing in the meta-program) can be constructed. It can be shown that the **eval** of programs constructed in this simple manner would be merely an abbreviation mechanism which can be simulated in $\mathcal{A}$ itself.

id	*next-id*	*var*	*op*	*att-1*	*att-2*	*arg-1*	*arg-2*	*rel*	*const*
1	2	X_1						R	
2	3	X_2		P					Fred
3	4	X_3	$\bowtie$			X_1	X_2		
4	5	X_4	ρ	C	C'	X_3			
5	6	X_5	ρ	P	C'	X_1			
6	7	X_6	$\bowtie$			X_4	X_5		
7	8	X_7	π	C'		X_6			
8	9	X_3	$\cup$			X_3	X_7		
9		X_8	π	P		X_3			

Figure 2: Program of Figure 1 stored in a relation

Indeed, a key potential of reflection is that programs can be specified whose structure or length cannot be a priori determined, but rather depends on the actual database instance to which the meta-program is applied. To this end, we need to add a mechanism to generate a number of new identifiers that is not bounded a priori. Since identifier creation is already well-studied in the context of object-oriented query languages (e.g., [AK89, HY90]), we can simply adapt it to our context. Concretely:

Definition 2.4 We further extend $\mathcal{RA}$ with terms '$\gamma_C(X)$', where X is a variable and C is an attribute name with $dom(C) = \mathcal{D}^{\text{id}}$. The semantics of the γ operator is as follows. For relation r, $\gamma_C(r)$ is obtained by extending r with a new column C, containing a distinct new identifier value for each tuple.

So, informally, identifier generation γ_C is the "converse" of projection π_C.

Example 2.5 Assume the database scheme contains a relation name R with unary relation scheme $\tau(R) = \{A\}$. The following $\mathcal{RA}$ program shows how, starting from 2 a priori given identifiers, $2n$ new ones can be generated, where n is the size of the R relation in the database instance:

$$X_1 := \{(id:1)\}; \qquad X_3 := X_2 \bowtie R;$$
$$X_2 := X_1 \cup \{(id:2)\}; \quad X_4 := \gamma_C(X_3).$$

The result of this program applied to a particular R relation is shown in Figure 3.

$R =$

A
a
b
c

$X_4 =$

id	*A*	*C*
1	*a*	11
2	*a*	12
1	*b*	13
2	*b*	14
1	*c*	15
2	*c*	16

Figure 3: Illustration of identifier-generating program from Example 2.5

It will be convenient to assume that a linear order is known on the domain $\mathcal{D}^{\text{id}}$ of identifiers. Since $\mathcal{D}^{\text{id}}$ is typically implemented using system-generated natural numbers, this assumption is generally harmless. The order can be accessed by allowing in $\mathcal{RA}$ also selections of the form $\sigma_{A<B}$, where A and B are attribute names with $dom(A) = dom(B) = \mathcal{D}^{\text{id}}$. Moreover, as illustrated by Example 2.5, we will make the technical assumption that the γ operator generates the new identifiers in increasing order, grouped according to the non-identifier columns of its argument relation. Also this assumption is easy to implement in practice. It will become clear that we will make only a modest use of the above two assumptions. The only reason why we need them is that the statements in a program relation must be linearly ordered.

3 Reflection and bounded looping

In this section, we show that the expressive power of $\mathcal{RA}$ is closely related to that of $\mathcal{BA}$, the relational algebra augmented with *bounded looping* [Cha81]. We also determine the complexity of $\mathcal{RA}$.

Formally, $\mathcal{BA}$ extends $\mathcal{A}$ with a **for** construct, as follows.

Definition 3.1 Let P be an $\mathcal{A}$ program, and let X be a variable. Then '**for** $|X|$ **do** P **od**' is an allowed statement in $\mathcal{BA}$. Its semantics is that P is repeated n times, where n is the cardinality of the value of X upon entry of the loop.

Example 3.2 Recall from Example 2.1 the program P computing the children and grandchildren of Fred. We can adapt this program to compute all descendants of Fred, by surrounding the main body of the program with a **for**-loop as follows: (statement numbers refer to statements of P)

```
1; 2;
for |X1| do
  3; 4; 5; 6; 7; 8
  od;
9
```

We show the following:

Theorem 3.3 *The $\mathcal{BA}$ statement* **for** $|X|$ **do** P **od** *can be simulated in $\mathcal{RA}$.*

Proof: We only sketch the proof. First, we construct in variable X_1 a program relation for P. Then, using essentially the technique illustrated in Example 2.5, we generate n copies of X_1 in program relation variable X_2, where n is the cardinality of the current value of X. Using the linear order on identifiers, we link the n copies together, yielding one single program that equals the n-fold composition of P with itself. The latter program is finally executed using **eval**, with the desired effect. □

Theorem 3.3 shows that bounded looping can be simulated by run-time program construction and execution. Of course, one cannot expect the verbatim converse to this theorem to hold. Indeed, in an application of **eval**(X), X could hold an arbitrary program relation stored in the database. In this case we do not have any clue as to how the program stored in X looks like. Hence, proving that general reflection can be simulated in $\mathcal{BA}$ would imply that one can construct an interpreter for $\mathcal{A}$ in $\mathcal{BA}$. But this is implausible, since the expression complexity of $\mathcal{A}$ is PSPACE-complete, while the data complexity of $\mathcal{BA}$ is PTIME [Var82]. The existence of an interpreter for $\mathcal{A}$ in $\mathcal{BA}$ would therefore imply that PTIME equals PSPACE. In [SVGG93], it was shown that $\mathcal{A}$ programs can be interpreted using bounded looping on *nested* relations of nesting depth 1. The data complexity of the latter construct is EXPTIME-complete.

However, we can disregard the interpreting power of reflection, and view it exclusively as a tool to execute $\mathcal{A}$ programs that are constructed at run-time. We can make this formal by the following two assumptions.

1. *Lexical-freeness:* The lexical domains (cfr. the discussion after Example 2.2) used to construct program relations are disjoint from the data domains used in the "ordinary" relations stored in the database.

2. *Constant-freeness:* Constants do not appear in the program relations to which **eval** is applied. This can be easily checked at run-time in $\mathcal{A}$; the *const* column must be completely blank.

Note that the proof of Theorem 3.3 remains valid under these assumptions. Note also that the assumptions do *not* prevent that the *structure* or *length* of the constructed programs can depend on the contents of the stored relations. Finally, note that constant-freeness neither prevents the use of a *fixed, finite* number of constant relations in the program relations. For instance, in Figure 2, we can always assume that there is a constant relation $\{(P : \text{Fred})\}$ (with that name) stored in the database. We can now prove the following converse to Theorem 3.3:

Theorem 3.4 *On databases over lexical-free domains, every constant-free* **eval***-statement occurring in an $\mathcal{RA}$ program can be simulated in $\mathcal{BA}$.*

Proof: We only sketch the argument. Let Q be an $\mathcal{RA}$ program, and consider a statement: X_1 := **eval**(X_2) occurring in Q. First, we check that X_2 indeed contains a (constant-free) $\mathcal{A}$ program P. Next, we execute P. Finally, we assign to X_1 the result of this execution, i.e., the result of the last statement. In this proof sketch we focus on the second step. The execution of P is simulated using a **for**-loop, which repeatedly takes the next statement from P and executes it. There is only a finite number of possibilities for the statement. Indeed, it is constant-free, and by the lexical-free domain assumption, is built up from the finite number of lexical symbols explicitly appearing as constants in Q. Hence, the statement can be executed after inspection by a constant number of if-then-else tests. More specifically, the loop has the following structure:

```
$X := X_2$;
for $|X_2|$ do
  /* put in $S$ the first statement remaining in $X$ */
  $S := \{t \in X \mid \neg\exists t' \in X : t'(\mathit{next\text{-}id}) = t(\mathit{id})\}$;
  $X := X - S$;
  /* let $S = \{t\}$ */
  if $t(\mathit{var})$ = '$Y_i$' then
    if $t(\mathit{op})$ = '$\bowtie$' then
      if $t(\mathit{arg\text{-}1})$ = '$Y_j$' then
        if $t(\mathit{arg\text{-}2})$ = '$Y_k$' then
          $Y_i := Y_j \bowtie Y_k$
        else ...
      else ...
    else ...
  else ...
od
```

The tuple relational calculus formula and the if-then-else tests are only shorthands and can be translated into $\mathcal{A}$. □

Note that the simulation procedures of Theorems 3.3 and 3.4 yield reductions in constant space and logarithmic space, respectively. Furthermore, the output size of the first reduction is linear, and that of the second is polynomial. From these observations and the preceding discussion, we obtain:

Corollary 3.5 *1. The data complexity of unrestricted $\mathcal{RA}$ is PSPACE-complete, being the expression complexity of $\mathcal{A}$. (The expression complexity is in EXPSPACE.)*

2. The data complexity and expression complexity of constant- and lexical-free $\mathcal{RA}$ is PTIME-complete and EXPTIME, respectively, being the data complexity and expression complexity of $\mathcal{BA}$. □

We conclude this section with a remark. We did only consider one-level reflection in $\mathcal{RA}$; the programs that are executed dynamically using **eval** are $\mathcal{A}$ programs and hence do not in turn contain applications of the **eval** operator. Similarly, the body of a **for**-loop in $\mathcal{BA}$ is an $\mathcal{A}$ program and does not contain a **for**-loop in turn. Extensions to deeper levels of reflection and looping are an interesting direction for further research. In this respect, we would like to point out that, to our knowledge, it is an open problem whether nesting of **for**-loops yields a strict increase in expressive power.

4 Applications

In this section, we discuss the applicability of the reflective algebra, and situate it within related work. The proofs of the claims are omitted.

4.1 Data dictionaries

Recall that a program relation is built up from *lexical* domain elements that make up the program the relation contains. Two particular kinds of lexical symbols are relation names and attribute names. These symbols are not only useful for program relations; they can also be used to describe the database scheme. In fact, most relational database systems store a description of the database scheme in a relation called the *data dictionary*. Using the reflective algebra, we can better exploit the presence of the data dictionary.

Consider the data dictionary shown in Figure 4 (left). Now consider the query "What is known about John?" As answer to this query, we want all triples $(\mathit{rel} : R, \mathit{att} : A, \mathit{val} : v)$, such that the R relation contains a tuple t in which 'John' appears, and $t(A) = v$. An illustration is given in Figure 4 (right). For every fixed database scheme, this query can be expressed in the relational algebra: we use the program consisting of all statements of the form

$$X := X \cup \{(\mathit{rel} : R)\} \bowtie \{(\mathit{att} : A)\} \bowtie \pi_A \sigma_{A'=\text{John}}(R)$$

where R is a relation name of the scheme and A, A' are attributes in $\tau(R)$. This program depends on the scheme.

However, in the reflective algebra, we can write one single program that will work for *any* scheme.

rel	*att*
Persons	*name*
Persons	*age*
Children	*parent*
Children	*child*
Hobbies	*person*
Hobbies	*hobby*

rel	*att*	*val*
Persons	*name*	John
Persons	*age*	24
Children	*parent*	John
Children	*child*	Steve
Children	*child*	Iris
Hobbies	*person*	John
Hobbies	*hobby*	ping-pong
Hobbies	*hobby*	math

Figure 4: Left: Example of a data dictionary. Right: All that is known about John in a database over this dictionary. (John's age is 24, has children Steve and Iris, etc.)

The idea is to write a meta-program that constructs the above program only at run time, by accessing the data dictionary. The program can then be executed using **eval**. The result, but no longer the query itself, will still depend on the scheme (and the contents) of the database under consideration.

Recently, Ross [Ros92] proposed a model and an algebra supporting databases where not only the data dictionary, but also other relations can contain relation names. Ross's algebra extends the relational algebra with the *expansion* operator α. Slightly adapted to the present context, this operator takes a relation scheme τ as parameter and a unary relation r as argument, and replaces every relation name R with $\tau(R) = \tau$ appearing in r by its associated relation. It should be clear that, using a similar technique as described above, we can express the α operator in the reflective algebra.

Some relational database systems also store view definitions in a data dictionary relation. Rather than (relation name, attribute name) pairs, this relation contains (view name, view definition) pairs. The view definitions are often (equivalent to) $\mathcal{A}$ programs. Hence, in our model, we store them not in the dictionary itself, but in program relations that have the same name as the view name. Interestingly, we can then translate a program which uses views to an $\mathcal{RA}$ program that looks up the definition of each view only at run time, using the view dictionary and Ross's α operator.

4.2 Procedural data

Storing programs as data has been investigated earlier by Stonebraker et al. [S+84, S+87]. In a practical context, they proposed a system, called QUEL+, allowing QUEL programs to be stored as strings in tuple components and executed dynamically. QUEL+ can easily be formalized in the reflective algebra.

Recall the Persons-Hobbies database from the previous discussion. In QUEL+, we could add an attribute *hobbies* to the Persons relation, containing a QUEL program retrieving all hobbies of a specific person. A typical tuple in this extended Persons relation would be $(name : \text{John}, age : 24, hobbies : Q)$, where Q is the QUEL query: `retrieve` *Hobbies.hobby* `where` *person*=param-1. To find all age-hobby pairs, we can then use the QUEL+ query:

retrieve (*Persons.age*,
Persons.hobbies **with** *name*)

The **with** operator binds the parameter of the executed queries.

To model the above example, we store Q in a program relation. The selection '*person*=param-1' is expressed using a join with a constant holding, initially, the literal symbol 'param-1'. The above QUEL+ query is then expressed by constructing, for each person in the Persons relation a copy of Q where param-1 is replaced by the person's name, and to which a join with the person's age is attached. Finally, using identifier generation, the programs thus obtained are united into one single program, which is finally executed using **eval**. Note that in this simple example, every Person tuple contains the same QUEL query Q. This need not be the case in general.

Stonebraker et al. argued that using procedural data, unnormalized (nested) relations and complex objects can be represented. For example, the procedural field *hobbies* can be alternatively viewed as a nested relation.

However, the manipulative aspect of this representation has not yet been considered. For example, one could think of the above-stated QUEL+ program as *unnesting* the hobbies field. For more complex operations however, simple execution of stored procedures is not always sufficient, and one needs the ability to construct new procedures. As stressed from the outset, the reflective algebra supports such constructions. As a simple example, the nesting $\nu_{\{B\}}(R)$ of a relation $R(A, B)$ can be constructed as the program relation containing $\pi_B(R \bowtie \{(A : \text{param-1})\})$.

4.3 Manipulation of methods in OODBS

In object-oriented database systems [KL89], objects have not only data attributes but also procedural attributes, called *methods*. Just like the view dictionary discussed in Section 4.1, it would be interesting if these methods can be stored in the database itself. If the methods are implemented using the relational algebra (e.g., [HS89]), we can again store them in program relations.

The reflective algebra can now be used to model applications that require the construction of derived method implementations from existing ones. For example, consider an OODB application where different persons can have different implementations of the *hobbies* method, due to overriding. Each person tuple contains the name of the program relation containing its method implementation. Suppose we now want to define a view consisting of all married couples, having a method *hobbies* which returns the union of the hobbies of the husband and those of the wife. To do this, we need to construct for each couple a new program that computes the union of the results of two other programs. This construction can be carried out in the reflective algebra.

5 Conclusion

The ideas presented in this paper can serve as the beginning of a more comprehensive investigation of the representation, manipulation, and execution of programs (queries, procedures, methods, ...) that are stored in the database together with "ordinary" data. Noteworthy is that in this context, the notion of expression complexity becomes equally important as the conventional notion of data complexity. An interesting open problem, inspired by a remark made by Beeri, is: *How can we extend the notion of* computable query [CH80] *in this setting?* From a more practical perspective, it would be useful to design syntactic variants of $\mathcal{RA}$ (or other reflective database languages) that are more amenable to static type checking, an issue which we have ignored in this paper.

References

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In Clifford et al. [CLM89], pages 159–173.

[AR89] H. Abramson and M.H. Rogers, editors. *Meta-Programming in Logic Programming.* MIT Press, 1989.

[CH80] A. Chandra and D. Harel. Computable queries for relational database systems. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.

[Cha81] A. Chandra. Programming primitives for database languages. In *Proceedings 8th ACM Symposium on Principles of Programming Languages*, pages 50–62, 1981.

[CLM89] J. Clifford, B. Lindsay, and D. Maier, editors. *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, volume 18:2 of *SIGMOD Record.* ACM Press, 1989.

[HS89] R. Hull and J. Su. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In Clifford et al. [CLM89], pages 147–158.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In D. McLeod,

R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 455–468. Morgan Kaufmann, 1990.

[KL89] W. Kim and F.H. Lochovsky, editors. *Object-Oriented Concepts, Databases, and Applications.* Frontier Series. ACM Press, Addison-Wesley, 1989.

[MN88] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection.* North-Holland, 1988.

[Ros92] 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.

[S^+84] 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.

[S^+87] M. Stonebraker et al. Extending a database system with procedures. *ACM Transactions on Database Systems*, 12(3):350–376, 1987.

[S^+90] D. Stemple et al. Exceeding the limits of polymorphism in database programming languages. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, *Advances in Database Technology—EDBT'90*, volume 416 of *Lecture Notes in Computer Science*, pages 269–285. Springer-Verlag, 1990.

[S^+92] D. Stemple et al. Type-safe linguistic reflection: a generator technology. Research report CS/92/6, Univ. St Andrews, 1992.

[SVGG93] L.V. Saxton, D. Van Gucht, and M. Gandhi. Universal queries for relational query languages. Technical Report 374, Indiana University Dept. Computer Sci., 1993.

[Var82] M. Vardi. The complexity of relational query languages. In *14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.