JOURNAL OF COMPUTER AND SYSTEM SCIENCES **52**, 537–549 (1996)
ARTICLE NO. 0040

Reflective Programming in the Relational Algebra*

JAN VAN DEN BUSSCHE†

Department of Mathematics and Computer Science, University of Antwerp (UIA), Universiteitsplein 1, B-2610 Antwerp, Belgium

DIRK VAN GUCHT‡

Computer Science Department, Indiana University, Bloomington, Indiana 47405-4101

AND

GOTTFRIED VOSSEN§

Institut für Wirtschaftsinformatik, University of Muenster, Grevenerstrasse 91, D-48159 Muenster, Germany

This paper introduces a reflective extension of the relational algebra. Reflection is achieved by storing and manipulating relational algebra programs as relations and by adding a LISP-like evaluation operation to the algebra. We first show that this extension, which we call the reflective algebra, can serve as a unifying formalization of various forms of procedural data management which have been considered in database systems research. We then study the expressive power and complexity of the reflective algebra. In particular, we establish a close correspondence between reflection and bounded looping, and between tail-recursive reflection and unbounded looping. These correspondences yield new logical characterizations of PTIME and PSPACE.

1. INTRODUCTION

The concept of *reflection* was introduced by Smith [23, 24] to study programs that need to analyze, and potentially modify, their own behavior. Reflection has since received attention in different areas of computer science. In programming languages, it has been used as a tool in the study of interpreters [3, 16, 18, 33], the design of extensible programming languages [13], and polymorphism [26]. In A.I., reflection has been used to study programs that must explain their own reasoning strategy [4, 15]. In databases, Stemple *et al.* [25] applied reflection in the dynamic creation of data types in database programming languages.

A very rough definition of reflection is that programs can be treated as data and vice versa. The meta-programming facilities of languages like LISP provide a simple and good example. In LISP, both programs and data are represented in a uniform format, namely lists. It is thus possible to build representations of LISP programs by using the **quote** function and list constructors. Programs represented as data in this form can then be executed dynamically by explicit application of LISP's evaluation operator **eval**. (In the literature, the process of constructing data representations of programs is often called *reification*, and only the process of evaluating these representations is then called reflection.)

Of course, the idea of reflection is as old as the concept of computation, dating back to the universal Turing machine, universal recursive functions, 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 programming language will not enhance its expressive power; the features are typically only meant to allow for a more natural or succinct expression of certain advanced programming 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. Viewing the relational algebra as a programming language to express database queries, this language is certainly not computationally complete (in fact, it is contained in LOGSPACE [32]). Well-known examples of computable queries not expressible in the relational algebra are parity checking and transitive closure computation.

Our motivation for this study stems from the increasing attention that is being paid to *procedural data management* in database systems. To our knowledge, Stonebraker *et al.* [27, 28] were the first to investigate the treatment of programs as data in the field of databases. Taking an informal point of view, they argued that a wide range of applications can benefit from procedural data. Then there is the current interest in so-called *active databases* [21, 22],

* A preliminary version of this paper was presented at the Twelfth ACM Symposium on Principles of Database Systems (Washington, DC, May 1993).

† E-mail address: vdbuss@wins.uia.ac.be. Research assistant of the Belgian National Fund for Scientific Research (NFWO).

‡ E-mail address: vgucht@cs.indiana.edu.

§ E-mail address: vossen@uni-muenster.de.

0022-0000/96 $18.00

dealing with procedural data in the form of *rules* which can be stored together with the ordinary data in the database (e.g., [29]). Also in object-oriented database systems [34], procedural data in the form of *methods* play a central role. Moreover, note that even current commercial relational database systems maintain various *data dictionary* relations containing meta-information like relation names or view definitions. To date, most of the discussion on the management of mete data, and more importantly procedural data, in database systems has been held on an informal level, and only few theoretical models are available. F-Logic, for instance, unifies data and mete data [14]. Ross [19] introduced an algebra for first-order logic with higher order syntax (HiLog [7]), extending the relational algebra with a modest form of reflection to deal with relations containing relation names.

As already mentioned, our purpose in the present paper is to define an extension of the relational algebra with a general reflection mechanism and to study its properties. Our extension is based on a format for storing relational algebra programs in relations introduced by Saxton, *et al.* [20]. In analogy with LISP, 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.

The further contents of this paper can be summarized as follows. In Section 2, we introduce our model for reflective programming in the relational algebra. In Section 3, we show that this model can serve as a unifying formalization of the various forms of procedural and mete data management in database systems mentioned earlier.

In Section 4, we turn to expressiveness and complexity issues. Since reflection treats programs as data, the notion of *expression complexity* [32] is very relevant in this context. We also study the power of reflection when used purely as a language construct for expressing conventional queries. Since reflection makes it possible to specify programs, the structure or length of which can depend on the input database, the reflective algebra will turn out to be more expressive than the ordinary relational algebra. Specifically, we show that adding basic reflection is in a sense equivalent to adding bounded looping, and that adding tail-recursive reflection is equivalent to adding unbounded looping. These correspondences yield new logical characterizations of PTIME and PSPACE.

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 **eval** operation.

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

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

Remark 2.1. 1. We assume that a total order is known on the universe $\mathscr{U}$ of all data elements. The results of this paper rely heavily on this assumption.

2. We also assume that there is a fixed total order on the universe of attributes (this order will always be clear from the context). Combining this assumption with the previous one, it follows that there is a natural lexicographical ordering on the tuples of every relation.

3. Finally, we assume that the universe of attributes includes infinitely many *natural number attributes* whose domain is the set of natural numbers, and that the natural ordering of the natural numbers is compatible with the total order on $\mathscr{U}$.

Following [5], we view the relational algebra as a programming language, denoted $\mathscr{A}$, as follows. Programs in $\mathscr{A}$ are sequences of *assignment statements*. Each assignment statement has the form "$X := E$", where X is a *variable* and E is a *term*. Each variable X has an associated relation scheme $\mathrm{sch}(X)$, and can take relations over $\mathrm{sch}(X)$ as values. Each term is one of the following:

- a relation name is a term;
- a constant one-attribute one-tuple relation, of the form $(A: a)$ where A is an attribute and $a \in \mathrm{dom}(A)$, is a term;
- a variable is a term;
- if X, X_1, X_2 are variables, then the following are terms:

$X_1 \cup X_2$	(union);
$X_1 - X_2$	(difference);
$X_1 \bowtie X_2$	(natural join);
$\hat{\pi}_A(X)$	(projecting out attribute A);
$\sigma_{A=B}(X)$	(equality selection);
$\sigma_{A<B}(X)$	(less-than selection); and
$\rho_{A/A'}(X)$	(renaming of A to A').

(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) $X_7 := \hat{\pi}_{C'}(X_6)$;
(8) $X_3 := X_3 \cup X_7$;
(9) $X_8 := \hat{\pi}_P(X_3)$.

FIG. 1. Program of Example 2.2.

We assume familiarity with the usual operators of the relational algebra (cf. [30]); the only unusual one here is $\hat{\pi}$. Indeed, we define projection $\hat{\pi}_A$ as projecting *out* the attribute A; formally, if r is a relation over scheme τ. then $\hat{\pi}_A(r)$ equals $\pi_{\tau - \{A\}}(r)$ where π is the classical projection operator of the relational algebra. While π has a variable number of attribute parameters, $\hat{\pi}$ has exactly one. This will make it easier in Subsection 2.1 to reify $\mathscr{A}$-programs. Obviously, π can be defined in terms of $\hat{\pi}$.

Now given an $\mathscr{A}$-program P, let $\mathscr{S}$ be the set of all relation names occurring in P. Then P can be applied to any database instance I over $\mathscr{S}$. The relation names take their values from I, the variables are initialized to the empty relation, and the assignment statements are executed in order. The final result of the program is, by default, the value of the variable occurring on the left-hand side of the last assignment statement. Since relation names and variables are typed (by their associated relation schemes), programs should be type-checked, but we will ignore this in this discussion.

EXAMPLE 2.2. Consider a database containing a parent-child relation R over scheme $\{P, C\}$. The program shown in Fig. 1 computes, in variable X_8, the set of all children and grandchildren of Fred. The statements are numbered for easy reference.

2.1. *Reification of Relational Algebra Programs*

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 constituting a program (we say that the program is *reified*) as argument and executes it. If we define a format in which programs can be stored as relations we can define an analogous reflective operation for the relational algebra. We next show how this is possible, following [20].

EXAMPLE 2.3. The program shown in Fig 1 can be stored in a relation over the attributes {*sno*, *var*, *op*, *att*-1, *att*-2, *arg*-1, *arg*-2, *rel*, *const*}, as shown in Fig. 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

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

FIG. 2. Program of Fig. 1 stored in a relation.

format. Non-applicable entries are filled with a blank (or some other fixed data element). Finally, every statement has a statement number *sno*, a natural number. ▮

We call the relation scheme {*sno*, *var*, *op*, *att*-1, *att*-2, *arg*-1, *arg*-2, *rel*, *const*} the *program scheme*. Using the format illustrated in Example 2.3, every $\mathscr{A}$-program can be stored in a relation over the program scheme; such relations are called *program relations*. The domains of the program-scheme attributes are as follows.

- *sno* is a natural number attribute, i.e., dom(*sno*) is the set of natural numbers;
- dom(*op*) equals $\{\cup, -, \bowtie, \hat{\pi}, \rho, \sigma_=, \sigma_<\}$;
- dom(*att*-1) and dom(*att*-2) equal the set of attributes;
- dom(*var*), dom(*arg*-1), and dom(*arg*-2) equal the set of variables;
- dom(*rel*) is the set of relation names; and
- dom(*const*) is the set of all data elements.

To be entirely correct, the blank symbol should be added to all of these domains.

Remark 2.4. 1. Variables, operator symbols, attributes, relation names, and the blank symbol are called *lexical symbols*. Note that the above implies that we consider each lexical symbol to be contained in the universe of data elements.

2. In a program relation, statements need not be numbered consecutively. It suffices that the attribute sno is a key for the relation. The first statement is then the statement with the smallest number, and for each statement in the sequence (except the last), the next statement is the statement with the smallest higher number.

Program relations could be given in the database, but they can also be constructed using $\mathscr{A}$-programs. A very simple case of the latter is given by the following easy but important lemma:

LEMMA 2.5. *For every program relation r there exists an $\mathscr{A}$-program that computes r.*

Proof. The program P assembles r from constant relations using union and join. The constant relations hold the

different symbols used in r. For example, for the program relation of Fig. 2, we will have $(sno\colon 1)$, $(var\colon X_1)$, $(op\colon \bowtie)$, etc. ∎

Programs used to construct program relations are called *meta-programs*. Of course, meta-programs of the basic type considered in Lemma 2.5 are not sufficient, as only constant program relations can be constructed in this way. Indeed, one of the key potentials of reflection seems to be that programs can be specified whose structure or length is not determined a priori, but rather depends on the actual database instance to which the meta-program is applied. In order to be able to exploit this potential, we need at least be able to generate sets of new statement numbers. A practically convenient approach to accomplish this is to use an *invention* mechanism similar to the one used in update languages [2] and object-oriented query languages [1, 11]. In our algebraic context, we define this mechanism as an additional algebra operator on relations as follows:

DEFINITION 2.6. Let τ be a relation scheme, and let N be a natural number attribute not in τ. Let r be a relation over τ. Then $\mathbf{number}_N(r)$ is a relation over $\{N\} \cup \tau$ obtained by extending the tuples of r with a new attribute N, such that each tuple is numbered with a new natural number, in increasing order according to the lexicographical ordering of the tuples in the relation.

We can extend the programming language $\mathscr{A}$ by allowing terms of the form "$\mathbf{number}_N(X)$" with X a variable.

EXAMPLE 2.7. Recall the motivation for introducing the numbering operator, given before Definition 2.6. Assume the database scheme contains a relation name R with unary relation scheme $\mathrm{sch}(R) = \{A\}$. The following program shows how, starting from a constant (k) a priori given statement numbers, $k \cdot n$ new ones can be generated, where n is the size of relation R in the database instance. In this example, $k = 2$.

$$X_1 := (sno\colon 1);$$
$$X_2 := X_1 \cup (sno\colon 2);$$
$$X_3 := R \bowtie X_2;$$
$$X_4 := \mathbf{number}_N(X_3).$$

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

$R =$

A
a
b
c

$X_4 =$

N	A	sno
11	a	1
12	a	2
13	b	1
14	b	2
15	c	1
16	c	2

FIG. 3. Illustration of identifier-generating program from Example 2.7.

2.2. *Reflection in the Relational Algebra*

We are now ready to extend the language $\mathscr{A}$ with a LISP-like evaluation operator (and the numbering operator of the previous subsection), yielding the *reflective relational algebra*, denoted $\mathscr{RA}$.

DEFINITION 2.8. A term of $\mathscr{RA}$ is either

1. a term of $\mathscr{A}$;
2. an expression of the form $\mathbf{number}_N(X)$, with W a variable; or
3. an expression of the form $\mathbf{eval}(X)$, with X a variable over the program scheme.

The semantics of the **number** operator was already defined in Definition 2.6. The semantics of a term $\mathbf{eval}(X)$ is as follows. If X holds a program relation representing an $\mathscr{A}$-program P. then $\mathbf{eval}(X)$ executes P and evaluates to P's final result. If X does not hold a syntactically correct program, then $\mathbf{eval}(X)$ yields the empty relation by default.

3. APPLICATIONS

In this section, we discuss the applicability of the reflective algebra, and situate it within related work. In particular, we demonstrate, mostly by way of examples, that the reflective algebra leads to an improved exploitation of a data dictionary as well as to a uniform treatment of procedural data. As will be seen, this has additional implications, including the possibility to simulate the algebra introduced in [19].

3.1. *Using Data Dictionaries*

Recall that a program relation is built up using *lexical* data 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, relational database systems typically store a description of every database scheme they handle in a relation called the *data dictionary*. Using the reflective algebra, we can better exploit the presence of such a dictionary.

Consider the data dictionary shown in Fig. 4 (left). Now consider the query "What is known about John?" As answer to this query, we want all triples $(rel\colon R, att\colon A, val\colon v)$, such that the R relation contains a tuple t in which "John" appears, and $t(A) = v$. An illustration is given in Fig. 4 (right). For every fixed database scheme, this query can be expressed in the relational algebra as

$$\bigcup_{R,\,A,\,A'} (rel\colon R) \bowtie (att\colon A) \bowtie \rho_{A/val}\pi_A\sigma_{A' = \mathrm{John}}(R)$$

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

FIG. 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, he has children Steve and Iris, etc.)

where R is a relation name of the scheme and A, A' are (not necessarily distinct) attributes in $\mathrm{sch}(R)$. We can easily implement this expression with an $\mathscr{A}$-program, but of course the program (like the expression) will depend on the database scheme.

However, in the reflective algebra, we can write one single program that will work for *any* database scheme. The idea is to write a meta-program that implements the above expression only at run time, by accessing the data dictionary. The program thus constructed 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.

Roughly, the meta-program constructs the program according to the following procedure. First, using the dictionary, all triples of the form (R, A, A'), where both A and A' are attributes of R. are generated by a straightforward $\mathscr{A}$ program. Next, auxiliary program relation variables for the necessary selections and projections are constructed, which are then joined appropriately with the previously derived triples, in order to turn the triples into algebra statements. Finally, the auxiliary program relations are united, and the **number** operator is applied to the result in order to obtain a proper statement sequencing.

Another aspect to be mentioned in connection with dictionaries, which represents another application area for the reflective algebra, is query optimization. Assume that P is a program relation whose contents represents the relational algebra query "$\sigma_{A=a}(R \bowtie S)$" In $\mathscr{A}$, this query is written as follows:[1]

$$X_1 := R; \qquad X_2 := S;$$

$$X_3 := X_1 \bowtie X_2; \quad X_4 := \sigma_{A=a}(X_3);$$

[1] The selection by attribute value $\sigma_{A=a}$ is a shorthand for a join with the constant relation $(A : a)$.

If A is an attribute of R, but not of S, query optimization would suggest rewriting the given query as "$\sigma_{A=a}(R) \bowtie S$", which is represented by the following program:

$$X_1 := R; \qquad X_2 := S;$$

$$X_3 := \sigma_{A=a}(X_1); \quad X_4 := X_3 \bowtie X_2;$$

Besides a renumbering of some of the variables occurring in this program, the latter basically results from the former by an exchange of two statements. Hence, it is possible to write an expression in the reflective algebra which detects, by an inspection of a given program relation and the dictionary, whether query optimization is possible and, if so, modifies P such that this program relation finally contains the optimized expression.

3.2. *Simulation of Ross's Algebra*

Recently, Ross [19] 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 an *expansion* operator called α. Slightly adapted to the present context, this operator takes a relation scheme τ as parameter and a unary relation U as argument, and replaces every relation name R with $\mathrm{sch}(R) = \tau$ appearing in U by its associated relation.

We illustrate this operator using a simple example. Assume relation U has scheme $\{rel\}$ and contains $\{R, S, T\}$, where R and S are relation names with scheme $\{A, B\}$ and T is a relation name having another scheme. Let R and S have the following contents:

R:	A	B
	a	b
	b	c

S:	A	B
	b	c
	d	e

Then $\alpha_{A,B}(U)$ would produce a relation with the following contents:

R	a	b
R	b	c
S	b	c
S	d	e

Using a similar technique as described in the previous subsection, we can express the α operator in the reflective algebra; we now indicate how to do this using the above example. First, we select from the data dictionary those

relation names in U that have scheme $\{A, B\}$, which is easily expressed in relational algebra. Let the result be $\{R_1, ..., R_k\}$. Then, a relation is constructed containing the following statements for each R_i:

$$X := R_i; \quad Y = (rel: R_i); \quad Z := Y \bowtie X.$$

Finally, using the **number** operator, the different groups of statements are linked together by newly generated statement numbers, yielding a program which when evaluated using **eval** will simulate $\alpha_{A,B}(U)$.

As a further implication of the above exposition, the following can be noted: Relational database systems commonly also store view definitions in a specific relation of the data dictionary. Rather than (relation name, attribute name) pairs, this relation contains (view name, view definition) pairs. Since relational algebra can be used as a tool for defining views, view definitions are often (equivalent to) $\mathscr{A}$ 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. For example, suppose we have a program relation named *john-hobbies* whose content is a program to retrieve John's hobbies from relation *Hobbies*, and assume that *john-hobbies* appears in the dictionary. Construct a program relation P holding the statement "$X_1 :=$ *john-hobbies*". Then

$$X_1 := \mathbf{eval}(P);$$

yields the contents of *john-hobbies*, and

$$X_2 := \mathbf{eval}(X_1);$$

computes the view.

It follows that we can translate an ordinary program which uses views to an $\mathscr{RA}$ program that looks up the definition of each view only at run time.

3.3. *Procedural Data*

Analogous to the way Ross's model and algebra generalize the notion of a dictionary, *procedural* data generalizes the idea of storing view definitions in relations. Storing programs as data has been investigated earlier by Stonebraker *et al.* [27, 28]. 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, as will be described next.

Recall the Persons-Hobbies database from Subsection 3.1. 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*: 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 following 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 which we also call Q. The selection "*person* = param-1" is expressed using a join with a constant holding, initially, the literal symbol "param-1". Thus, program relation Q will encode the following program:

$$X_0 := Hobbies;$$
$$X_1 := (person: \text{param-1});$$
$$X_2 := X_0 \bowtie X_1;$$
$$X_3 := X_3 \cup X_2$$

This program has to be executed repeatedly, once for each value of the parameter. Hence the union at the end, which collects the results of all executions (X_3 is initially empty).

The QUEL+ query is 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, which is not included below). Finally, using number generation, the programs thus obtained are concatenated and the resulting concatenation is executed. Formally (for clarity of presentation, some operators have been composed)

$$X_4 := \hat{\pi}_{age}(Person);$$
$$X_5 := X_4 \bowtie Q;$$
$$X_6 := \sigma_{const = \text{param-1}}(X_5);$$
$$X_7 := \sigma_{name = name'}(\rho_{name/name'}(X_6) \bowtie X_4);$$
$$X_8 := \rho_{name'/const}\hat{\pi}_{const}(X_7);$$
$$X_9 := (X_5 - X_6) \cup X_8;$$
$$X_{10} := \mathbf{number}_N(X_9);$$
$$X_{11} := \rho_{N/sno}\hat{\pi}_{sno}\hat{\pi}_{name}(X_{10});$$
$$\mathbf{eval}(X_{11}).$$

Note that it is important that the application of the **number** operator correctly groups together the statements of each program copy in the concatenation. Thereto, recall that statement numbers are generated in accordance with the

lexicographical ordering of the tuples in X_9. For the operation to work correctly, it suffices that in this lexicographical ordering, the *name* attribute (coming from X_4) is considered as the first component of each tuple, and the *sno* attribute (coming from Q) as the second.

In this simple example, every *Person* tuple contains the same QUEL query Q. This need not be the case in general. Different queries can be stored in different tuples by the name of their program relation. Instead of joining each person with the same program Q as above, one then uses the α operator of the previous subsection to look up the appropriate program for each person. The formal details are tedious but straightforward.

Stonebraker *et al.* argued that by 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.[2] However, the manipulative aspect of this representation has not yet been considered. For example, one could think of the above-discussed QUEL+ program as *unnesting* the hobbies field. For more complex operations, however, a 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 over the scheme $\{A, B\}$ can be constructed as the program relation containing the relational algebra expression $\pi_B(R \bowtie (A: \text{param-1}))$.

We finally mention that the reflective algebra allows one not only to store procedural data like ordinary data, but also the possibility of *querying* procedural data. For example, a query like "Which programs in the database depend on relation R?" can be answered by inspecting a program relation. One application of this feature is the area of *active databases* (cf. the Introduction), which commonly support rules of the form "if condition then action," since both the condition and the action can be modeled as programs in the database.

3.4. *Manipulation of Methods in OODBS*

In object-oriented database systems, objects have not only data attributes but also procedural attributes, called *methods*. Just like the view dictionary discussed in Section 3.1, it would be interesting if these methods could be stored in the database itself. If the methods are implemented using the relational algebra (e.g., [10]), 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 also be carried out in the reflective algebra.

[2] For an introduction to nested relations, see [17, Chap. 7].

4. EXPRESSIVENESS AND COMPLEXITY

In Sections 2 and 3, we used value invention (the **number** operator) as a practically convenient tool to generate natural numbers as statement numbers in run-time created program relations. However, there is an alternative approach which avoids the introduction of new data elements (up to a constant number), and which is more amenable to theoretical study, allowing sharper expressiveness and complexity results. The main idea behind this approach is to better exploit the order assumption on data elements (cf. Remark 2.1). This idea is worked out in Subsection 4.1.

To formally study the expressiveness and complexity of $\mathscr{R}\mathscr{A}$ and some of its variations, we need to introduce additional query languages and review the notions of the data and expression complexity of a query language. This is done in Subsection 4.2.

In Subsection 4.3, we then establish various expressiveness and complexity results about $\mathscr{R}\mathscr{A}$. In this analysis, it will prove useful to distinguish between **eval**'s interpretative power and its implicit looping semantics.

Finally, in Subsection 4.4, we extend $\mathscr{R}\mathscr{A}$ in turn by allowing **eval** statements to occur in program relations. The resulting language will be called the *recursive* reflective algebra, denoted $\mathscr{R}^2\mathscr{A}$ We establish various expressiveness and complexity results about this language.

4.1. *Reflection without Object Creation*

Consider an instance I over scheme $\mathscr{S}$. Define the *active domain* of I, denoted $\text{adom}(I)$, to be the set of data elements occurring in I (this unary relation can be derived via a program in $\mathscr{A}$). Without loss of generality, we assume that $\text{adom}(I)$ has at least two elements. (Indeed, we can always write an $\mathscr{A}$ program that uses constant terms to meet this requirement.) Because data elements are totally ordered (cf. Remark 2.1), it is natural to think of $\text{adom}(I)$ as representing the initial segment $1, ..., |\text{adom}(I)|$ of the (positive) natural numbers. We can therefore use data elements in $\text{adom}(I)$ in the role of *sno* values. To reify $\mathscr{A}$ programs of length greater than $|\text{adom}(I)|$, the join operation can be used to represent larger numbers than $|\text{adom}(I)|$. For example, assuming that $\text{adom}(I)$ is

defined over the relation scheme $\{A\}$, the statement $X := \text{adom} \bowtie \rho_{A/A'}(\text{adom})$ will generate a set of pairs which can be interpreted as representing the initial segment 1, ..., $|\text{adom}(I)|^2$. Additional joins are required to represent larger numbers.

Another remark related to the issue of statement numbers is that any relation r, because it is also ordered, can be thought of as representing the initial segment 1, ..., $|r|$. This remark will become relevant in the proof of Theorem 4.9.

The previous discussion gives two strategies to exploit the order on data elements as a means to represent natural numbers which, in turn, can be used as statement numbers in program relations. However, it is also clear that, under these strategies, we can no longer maintain the notion of a single program scheme as in Section 2.1; in fact, there now are infinitely many variable-sized program schemes:

DEFINITION 4.1. A program scheme is a relation scheme $\{sno_1, ..., sno_k, var, op, att\text{-}1, att\text{-}2, arg\text{-}1, arg\text{-}2, rel, const\}$, for some $k \geqslant 1$. The domains of *var*, *op*, *att*-1, ..., *const* are as in Section 2.1; however, sno_1, ..., sno_k are no longer required to be natural number attributes.

We can without difficulty adapt the definition of the **eval** operation to this setting. The only difference is that the sequencing of statements in program relations is secured by natural numbers represented as series of *sno*-attributes values (as just explained above), instead of as single *sno*-attribute natural number values (as used in Section 2.1).

Henceforth, we will no longer consider the **number** operator as part of the language $\mathscr{RA}$. Indeed, we have just shown that its use for reification purposes can be circumvented. It should be pointed out, however, that "value-invention" operations like the **number** operator also have other important uses, e.g., in obtaining computationally complete query languages and in object-oriented query languages [1, 2, 11]. We will not consider these aspects in the present paper.

4.2. *Query Languages and Complexity*

Two relational query languages will play a key role in our further analysis of the expressiveness and complexity of reflection: $\mathscr{BA}$, the relational algebra extended with *bounded* looping, and $\mathscr{WA}$, the relational algebra extended with *unbounded* looping [5].

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

DEFINITION 4.2. Let P be an $\mathscr{A}$ program, and let X be a variable. Then "**for** $|X|$ **do** P **od**" is an allowed statement in $\mathscr{BA}$. Its semantics is that P is repeated n times, where n is the cardinality of the value of X upon entry of the loop.[3]

[3] Note that loop-body P must be an $\mathscr{A}$ program and cannot be $\mathscr{BA}$ program in turn. On ordered databases, nesting of for-loops does not yield more expressive power, but whether the same holds on unordered databases is, to our knowledge, an open problem.

EXAMPLE 4.3. Recall from Example 2.2 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; 3;
for |X1| do
  4; 5; 6; 7; 8
  od;
9
```

Formally, $\mathscr{WA}$ extends $\mathscr{A}$ with a while construct, as follows.

DEFINITION 4.4. Let P be an $\mathscr{A}$ program, and let X be a variable. Then "**while** $X \neq \emptyset$ **do** P **od**" is an allowed statement in $\mathscr{WA}$. Its semantics is that P is repeated as long as the value of X is not the empty relation (which might be indefinitely).[4]

EXAMPLE 4.5. The $\mathscr{BA}$ program in Example 4.3 can be easily adapted to a $\mathscr{WA}$ program as follows:

```
X0 := R; 1; 2; 3;
while X0 ≠ ∅ do
   4; 5; 6; 7; 8; X0 := X3 − X0
   od;
9
```

We also need to review the notions of data and expression complexity of queries and query languages [32].

DEFINITION 4.6. Let $\mathscr{S}$ be a database scheme and τ a relation scheme. A *query* of type $\mathscr{S} \to \tau$ is a function Q, mapping instances over $\mathscr{S}$ to relations over τ, such that $\text{adom}(Q(I)) \subseteq \text{adom}(I)$ for each I.

A query Q is said to be in some complexity class $\mathscr{C}$ if the set $\{(I, t) \mid I \text{ an instance}, t \in Q(I)\}$ is in $\mathscr{C}$. The query is called complete for $\mathscr{C}$ if the set is complete for $\mathscr{C}$. A query language $\mathscr{L}$ is said to have *data complexity* in $\mathscr{C}$ if every query expressible in $\mathscr{L}$ is in $\mathscr{C}$. If one of the queries is complete for $\mathscr{C}$ then the data complexity is said to be complete for $\mathscr{C}$.

There is also another measure of complexity of a query language, which is based on the size of the programs rather than on the size of the data. A query language $\mathscr{L}$ is said to have *expression complexity* in $\mathscr{C}$ if for every database instance I, the set $\{(Q, t) \mid Q \text{ a query in } \mathscr{L} \text{ defined on } I, t \in Q(I)\}$ is in $\mathscr{C}$. If one of these sets is complete for $\mathscr{C}$ then the expression complexity of $\mathscr{L}$ is said to be complete for $\mathscr{C}$.

[4] It is known [2] that nesting of while-loops does not increase the expressive power.

To appreciate the difference between data complexity and expression complexity, we recall the following result of Vardi [32], which we will also need later on:

LEMMA 4.7. *The data complexity of $\mathscr{A}$ is in LOG-SPACE. The expression complexity of $\mathscr{A}$ is PSPACE-complete.*

It is easy to see that the data complexities of $\mathscr{BA}$ and $\mathscr{WA}$ are in PTIME and PSPACE, respectively. Actually, also the converse holds:

LEMMA 4.8. 1. *Every PTIME query is expressible in $\mathscr{BA}$.*

2. *Every PSPACE query is expressible in $\mathscr{WA}$.*

Proof. 1. It is easy to see that every fixpoint query [12] is expressible in $\mathscr{BA}$. Also, it is well-known [12, 32] that every PTIME query is a fixpoint query on ordered databases. Since in this paper, we work with ordered databases, this result applies to our context.

2. This is well-known [32]. ∎

4.3. *Reflection and Bounded Looping*

As a first illustration of the expressiveness of reflection as a query language construct, we show that bounded looping can be simulated by run-time program construction and execution:

THEOREM 4.9. *The $\mathscr{BA}$ statement* **for** $|X|$ **do** P **od** *can be simulated in $\mathscr{RA}$.*

Proof. First, using Lemma 2.5, we construct in variable X_0 a program relation for P. Next, we perform the following statements: (we assume that the scheme of X is $\{A_1, ..., A_k\}$)

$$
\begin{aligned}
X_1 &:= \rho_{A_1/sno_1}(X);\\
X_2 &:= \rho_{A_2/sno_2}(X_1);\\
&\vdots\\
X_k &:= \rho_{A_k/sno_k}(X_{k-1});\\
X_{k+1} &:= \rho_{sno_1/sno_{k+1}}(X_0);\\
X_{k+2} &:= X_k \bowtie X_{k+1};\\
result &:= \mathbf{eval}(X_{k+2});
\end{aligned}
$$

The first k statements simply rename the attributes $A_1, ..., A_k$ into $sno_1, ..., sno_k$, respectively. The contents of X_k is therefore a copy of X (but with renamed attributes). As discussed in Subsection 4.1, the order assumption allows us to interpret X_k as representing the initial segment of the natural numbers 1, ..., $|X|$.

The $(k+1)$th statement renames the statement number attribute *sno* in X_0 to sno_{k+1}.

In the $(k+2)$ th statement, X_{k+2} is assigned the cartesian product of X_k and X_{k+1}. The scheme of X_{k+2} is the program scheme $\{sno_1, ..., sno_k, sno_{k+1}, var, op, att\text{-}1, att\text{-}2, arg\text{-}1, arg\text{-}2, rel, const\}$, and the contents of X_{k+2} corresponds to an $\mathscr{A}$ program representing the $|X|$-fold composition of P.

Hence, applying the **eval** operator to X_{k+2} yields the same result as running the original **for** loop. ∎

Lemma 4.8 and Theorem 4.9 imply:

COROLLARY 4.10. *Every PTIME query is expressible in $\mathscr{RA}$.*

The converse of Corollary 4.10, i.e., the possibility that the data complexity of $\mathscr{RA}$ is in PTIME, is highly implausible. Indeed, we have:

PROPOSITION 4.11. *The data complexity of $\mathscr{RA}$ is PSPACE-hard.*

Proof. It suffices to make the following observation:

> The *data* complexity of $\mathscr{RA}$ is at least the *expression* complexity of $\mathscr{A}$.

Indeed, the expression complexity of $\mathscr{A}$ is defined in terms of the sets $\{(Q, t) \mid t \in Q(I)\}$ for each database instance I. Each such set corresponds to the query in $\mathscr{RA}$ which takes as input a program relation Q and returns as output the result of Q applied to I (using the **eval** operator). Since the expression complexity of $\mathscr{A}$ is PSPACE-complete (cf. Lemma 4.7) the proposition follows. ∎

Informally, the expression complexity of some query language $\mathscr{L}$ is the complexity of an *interpreter* for $\mathscr{L}$ (in the sense of programming languages). From the preceding discussion it follows that it is implausible that one can write an interpreter for $\mathscr{A}$ in $\mathscr{BA}$.[5]

As just observed, the intractability of the **eval**(X) operator stems from the fact that X can hold completely arbitrary $\mathscr{A}$-programs. In particular, we have no a priori knowledge about the relation variables and attribute names used in the program relations to be evaluated. It is therefore natural to consider the situation wherein we do have this knowledge. We can make this formal as follows.

DEFINITION 4.12. An $\mathscr{RA}$ program P is called *lexically constrained* if there exists a finite set C of lexical symbols (cf. Remark 2.4), such that all lexical symbols occurring in

[5] It is worthwhile to mention that it *is* possible to interpret $\mathscr{A}$ using bounded looping on *nested* relations [20]. The data complexity of the latter language is EXPTIME-complete [31].

program relations to which **eval** is applied during any possible execution of P are elements of C.

EXAMPLE 4.13. The simulation of the QUEL+ program shown in Section 3.3 is lexically constrained (since only a single QUEL query Q is used). On the other hand, suppose R is a relation in the database whose scheme is a program scheme. Then the program $X_1 := R$; $X_2 :=$ **eval**(X_1) is not lexically constrained.

Note that the proof of Theorem 4.9 remains valid under the lexically constrained program assumption, provided the appropriate set of lexical constants is chosen. Note also that the assumption does *not* prevent that the *structure* or *length* of the constructed programs can depend on the contents of the stored relations. We can now prove the following converse to Theorem 4.9:

THEOREM 4.14. *If P is a lexically-constrained $\mathscr{RA}$ program, then P can be simulated in $\mathscr{BA}$.*

Proof. Let $X_2 :=$ **eval**(X_1) be an eval statement occurring in P. First, we need to check that X_1 indeed contains an $\mathscr{A}$ program. If so, since P is lexically constrained by some finite set C, all lexical symbols occurring in X_1 belong to C. Therefore, the number of possibilities for the (sub)-tuples of X_1 defined over the attributes *var*, *op*, *att*-1, *att*-2, *arg*-1, *arg*-2, *rel*, and *const* is finite, whence the syntax check can be performed in $\mathscr{A}$ following a tedious but straightforward procedure.

The evaluation of X_1 is now simulated using a **for** statement, which repeatedly takes the next statement from X_1 and executes it after inspection by a constant number of if-then-else tests. More specifically, the loop has the following structure:

$X := X_1$;
for $|X|$ **do**
  /* put in S the first statement remaining in X */
  /* we assume that S has k *sno*-attributes */
  $S := \{t \in X \mid \neg \exists t' \in X\colon t'(sno_1, ..., sno_k) < t(sno_1, ..., sno_k)\}$;
  $X := X - S$;
  /* let $S = \{t\}$ */
  if $t(var) =$ 'Y_i' **then**
    if $t(op) =$ '$\bowtie$' **then**
      if $t(arg\text{-}1) =$ 'Y_j' **then**
        if $t(arg\text{-}2) =$ 'Y_k' **then**
          $Y_i := Y_j \bowtie Y_k$
        else…
      else…
    else…
  else…
od

The comparison $t'(sno_1, ..., sno_k) < t(sno_1, ..., sno_k)$ in the relational calculus formula is with respect to the lexicographical order and can be expressed as a Boolean combination of atomic comparisons of the form $t'(sno_i) < t(sno_i)$. The tuple relational calculus formula and the if-then-else tests are only shorthands and can be translated into $\mathscr{A}$ [30].

The only remaining technical detail is to indicate how, inside the above **for** loop, statements involving constant relations, i.e., of the form "$Y := (A\colon a)$" are handled. Thereto, the following subprogram is used:

if $t(var) =$ 'Y' **then**
  if $t(att\text{-}1) =$ 'A' **then**
    $Constant := \{t(const)\}$;
    $Y := \rho_{const/A}(Constant)$
  else…
else… ∎

As a corollary to Lemma 4.8 and Theorems 4.9 and 4.14, we obtain:

COROLLARY 4.15. *The class of lexically-constrained $\mathscr{RA}$ programs coincides with the class of PTIME queries.*

A drawback of the concept of lexically constrained programs is that it is semantic rather than syntactic:

PROPOSITION 4.16. *The problem whether a given $\mathscr{RA}$ program is lexically constrained is undecidable.*

Proof. We reduce the equivalence problem for $\mathscr{A}$ programs, well-known to be undecidable, to the lexically constrainedness problem for $\mathscr{RA}$ programs. Let P_1 and P_2 be two $\mathscr{A}$ programs working on an input relation R. We assume, without loss of generality, that P_1 and P_2 use different variables. Let the result variable of P_i be X_i, for $i = 1, 2$. Let X be another variable, and let P be an $\mathscr{A}$ meta-program which constructs in a program relation variable Z the program consisting of all possible statements of the form $Y := l$, where Y is some fixed variable and l is a data element occurring in the value of X. Then the following $\mathscr{RA}$ program is lexically constrained iff P_1 and P_2 are equivalent:

P_1; P_2;
if $X_1 = X_2$ **then** $X := \varnothing$
else $X := R$;
P;
eval(Z).

Indeed, assume P_1 and P_2 do not yield the same result on some input relation R. Then they will not yield the same result on an infinite number of other input relations R' isomorphic to R. Some of these relations R' will consist exclusively of lexical symbols such as relation variables.

Hence, the application **eval**(Z) is not lexically constrained. The converse implication is trivial. ▮

However, we can avoid this problem by considering a (decidable!) restriction on the databases, rather than on the programs. We define:

DEFINITION 4.17. Let C be a finite set of lexical symbols. A database instance I is called *C-lexical-free* if all lexical symbols (cf. Remark 2.4) appearing as data elements in one of the relations of I are among C.

Since lexical symbols are only useful to store program relations, lexical-free databases can be thought of as databases in which the possible procedural or meta-data are "static". In particular, $\varnothing$-lexical-free databases are "ordinary" databases which do not contain any procedural data. By considering $\mathscr{R}\mathscr{A}$ as a query language working on lexical-free databases only, we can study reflection purely as a language mechanism for expressing standard, classical queries.

We observe:

PROPOSITION 4.18. *For any finite set C of lexical symbols, $\mathscr{R}\mathscr{A}$ expresses precisely the PTIME queries on C-lexical-free databases.*

Proof. When restricting attention to C-lexical-free databases only, every $\mathscr{R}\mathscr{A}$ program is automatically lexically constrained, since it can only introduce a constant number of extra lexical symbols through its constant relations. Hence, the arguments developed in the proofs of Theorems 4.9 and 4.14 remain valid when considering lexical-free databases rather than lexically constrained programs. ▮

4.4. *Recursive Reflection and Unbounded Looping*

The type of reflection allowed in $\mathscr{R}\mathscr{A}$ is non-recursive. More specifically, up to now, the **eval** operator is only defined to work on (program relations containing) $\mathscr{A}$-programs, not $\mathscr{R}\mathscr{A}$-programs in turn. In this section, we drop this restriction, allowing eval to interpret programs containing **eval**-statements. We will call the resulting language the *recursive reflective algebra*, denoted $\mathscr{R}^2\mathscr{A}$.

Strictly speaking, the syntax of $\mathscr{R}\mathscr{A}$ and $\mathscr{R}^2\mathscr{A}$ can be the same. Their difference would result from associating different semantics to the **eval** operator. However, for clarity, it will be advantagous to syntacticaly differentiate between $\mathscr{R}\mathscr{A}$ and $\mathscr{R}^2\mathscr{A}$. This leads us to the following definition of the recursive **eval** operator **reval**.

DEFINITION 4.19. A term of $\mathscr{R}^2\mathscr{A}$ is either a term of $\mathscr{R}\mathscr{A}$, or an expression of the form "**reval**(X)", where X is a variable such that sch(X) is a program scheme. The semantics of the recursive evaluation operator is the following. If r is a program relation containing an $\mathscr{A}$ program, then **reval**(r) has the same effect as **eval**(r). If r is a program relation containing an $\mathscr{R}\mathscr{A}$ program P then **reval**(r) executes P and evaluates to P's final result. And similarly, if r is a program relation containing an $\mathscr{R}^2\mathscr{A}$ program P then **reval**(r) (recursively) executes P and evaluates to P's final result. Finally, if r does not contain a syntactically correct $\mathscr{R}^2\mathscr{A}$ program then **reval**(r) yields the empty relation by default.

Remark 4.20. The notion of lexically-constrained $\mathscr{R}\mathscr{A}$ program can be extended to $\mathscr{R}^2\mathscr{A}$ in a straightforward way.

We will show that the expressive power of $\mathscr{R}^2\mathscr{A}$ is closely related to that of $\mathscr{W}\mathscr{A}$. This relationship will allow us to derive results about the data complexity of $\mathscr{R}^2\mathscr{A}$.

THEOREM 4.21. *The $\mathscr{W}\mathscr{A}$ statement* **while** $X \neq \varnothing$ **do** P **od** *can be simulated by a lexically constrained program in $\mathscr{R}^2\mathscr{A}$.*

Proof. Let W_P be a program relation containing the following $\mathscr{R}^2\mathscr{A}$ program:

$$P;$$
$$Call_{P'} := \textbf{reval}(P')$$

Furthermore, let P' be a program relation containing the following $\mathscr{R}^2\mathscr{A}$ program:

$$X_{test} := X \bowtie W_P;$$
$$W_P := \hat{\pi}_{\tau(X)}(X_{test});$$
$$Call_P := \textbf{reval}(W_P)$$

The second statement is an abbreviation for a sequence of statements which project out, one by one, the attributes of $\tau(X)$.

If $X \neq \varnothing$ at the start of an execution of the program stored in P', then the contents of W_P after the second statement will be identical to the contents of W_P at the start. In that case, the third statement executes P (once) and subsequently reevaluates the program stored in P'. (Note that X can be side-effected by the execution of the program P.)

If $X = \varnothing$ at the start of the program, however, then the contents of W_P will be erased. As a consequence, the third statement will have no effect.

Hence, the $\mathscr{R}^2\mathscr{A}$ program:

$$Initialize(W_P);$$
$$Initialize(P');$$
$$X_{while} := \textbf{reval}(P').$$

correctly simulates the while-loop statement **while** $X \neq \varnothing$ **do** P **od**. The initializations of W_P and P' can be done by lexically constrained $\mathscr{R}\mathscr{A}$ programs. ▮

A notable feature of the program relations used in the above proof is that their sole use of reflection is by **reval** statements occurring in *tail-recursive position*. We now show the following converse to Theorem 4.21:

THEOREM 4.22. *Let P be a lexically-constrained $\mathscr{R}^2\mathscr{A}$ program in which all program relations used during the execution of P are either $\mathscr{A}$-programs or tail-recursive. Then P can be simulated by an $\mathscr{WA}$ program.*

Proof. Consider a statement of the form "$X_2 :=$ **eval**(X_1)" or "$X_2 :=$ **reval**(X_1)" in P. When simulating this statement, we know that X_1 has the form of an $\mathscr{A}$-program, possibly followed by a statement of the form "$X_4 :=$ **reval**(X_3)". Hence, we can use a **while**-loop with a structure similar to the **for**-loop used in the proof of Theorem 4.14. We only indicate the differences:

$X := X_1$;
while $X \neq \varnothing$ **do**
  ⋮
  if $t(op) =$ '**reval**' **then**
    if $t(arg\text{-}1) =$ 'X_3' **then**
      $X := X_3$
    else…
  else…
od

Note that, due the lexically constrained and tail-recursive nature of the **reval**-calls, the return values of these calls can be ignored without loss of generality. ∎

As a corollary to Lemma 4.8 and Theorems 4.21 and 4.22, we obtain:

COROLLARY 4.23. *The class of lexically constrained, tail-recursive $\mathscr{R}^2\mathscr{A}$ programs coincides with the class of PSPACE queries. Alternatively, on lexical-free databases, tail-recursive $\mathscr{R}^2\mathscr{A}$ expresses precisely the PSPACE queries.*

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. We hope they can also contribute to the ongoing discussion among database practitioners as to whether the separation of data and programs, one of the original ideas upon which current database systems have been built, is still tenable.

An interesting open problem (suggested by C. Beeri) is the question of what becomes of the notion of *computable query* [6] in the presence of procedural data.

From a more practical perspective, it would be useful to design syntactic variants of $\mathscr{RA}$ (or other reflective database languages) that are more user friendly and/or amenable to static type checking, issues which we have ignored in this paper. In this respect it will probably be advantageous to move to an object-oriented data model.

ACKNOWLEDGMENTS

We are indebted to Victor Vianu, who suggested to us a possible connection between recursive reflection and unbounded looping, and to Manoj Jain, who contributed to the proof of Theorem 4.21. We also thank the anonymous referees, for their thoroughness and helpful comments, and Antonio Badia, Anurag Mendhekar, and Ed Robertson, for helpful discussions with the second author.

Received November 20, 1995

REFERENCES

1. S. Abiteboul and P. Kanellakis, Object identity as a query language primitive, *in* [8, pp. 159–173].
2. S. Abiteboul and V. Vianu, Datalog extensions for database queries and updates, *J. Comput. System Sci.* **43**, No. 1 (1991), 62–124.
3. H. Abramson and M. H. Rogers, (Eds.), "Meta-Programming in Logic Programming," MIT Press, Cambridge, 1989.
4. A. Basu and R. Ahad, Using a relational database to support explanation in a knowledge-based system, *IEEE Trans. Knowledge Data Eng.* **4**, No. 3 (1992), 572–581.
5. A. Chandra, Programming primitives for database languages, *in* "Proceedings 8th ACM Symposium on Principles of Programming Languages, ACM Press, New York, 1981," pp. 50–62.
6. A. Chandra and D. Harel, Computable queries for relational database systems, *J. Comput. System Sci.* **21**, No. 2 (1980), 115–178.
7. W. Chen, M. Kifer, and D. W. Warrens, HiLog: A foundation for higher-order logic programming, *J. Logic Programming* **15**, No. 3 (1993), 187–230.
8. J. Clifford, B. Lindsay, and D. Maier, (Eds), "Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data," SIGMOD Record, Vol. 18:2, ACM Press, New York, 1989.
9. H. Garcia-Molina and H.V. Jagadish, (Eds), "Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data," SIGMOD Record, Vol. 19:2, ACM Press, New York, 1990.
10. R. Hull and J. Su, On accessing object-oriented databases: Expressive power, complexity, and restrictions, *in* [8, pp. 147–158].
11. R. Hull and M. Yoshikawa, ILOG: Declarative creation and manipulation of object identifiers, *in* "Proceedings of the 16th International Conference on Very Large Data Bases" (D. McLeod, R. Sacks-Davis, and H. Schek, Eds.), Morgan Kaufmann, Los Altos, CA, 1990.
12. N. Immerman, Relational queries computable in polynomial time, *Inform. and Control* **68** (1986), 86–104.
13. G. Kiczales, J. des Rivières, and D. Bobrow, "The Art of the Meta-object Protocol," MIT Press, Cambridge, MA, 1991.
14. M. Kifer, G. Lausen, and J. Wu, Logical foundations of object-oriented and frame-based languages, *J. Assoc. Comput. Machin.* **42**, No. 4 (1995), 741–843.
15. P. Maes and D. Nardi, (Eds.), "Meta-Level Architectures and Reflections," North-Holland, Amsterdam, 1988.
16. R. Muller, M-LISP: A representation-independent dialect of LISP with reduction semantics, *ACM Trans. Programming Languages and System* **14**, No. 4 (1992), 589–616.
17. J. Paredaens, P. De Bra, M. Gyssens, and D. Van Gucht, The Structure of the relational database Model, *in* "EATCS Monographs on

Theoretical Computer Science," Vol. 17, Springer-Verlag, New York/Berlin, 1989.
18. F. Pfenning and P. Lee, Metacircularity in the polymorphic λ-calculus, *Theoret. Computer Sci.* **89**, No. 1 (1991), 137–159.
19. K. Ross, Relations with relation names as arguments: Algebra and calculus, *in* "Proceedings 11th ACM Symposium on Principles of Database Systems, 1992," pp. 346–353.
20. L. V. Saxton, D. Van Gucht, and M. Gandhi, Universal queries for relational query languages, unpublished results.
21. SIGMOD, Session on triggers and derived data, *in* [8].
22. SIGMOD, Session on rule processing systems, *in* [9].
23. B. C. Smith, "Reflection and Semantics in a Procedural Language," Tech. Rep. 272, MIT LCS, 1982.
24. B. C. Smith, Reflection and semantics in LISP, *in* "Proceedings 11th ACM Symposium on Principles of Programming Languages, 1984," pp. 23–35.
25. D. Stemple *et al.*, Exceeding the limits of polymorphism in database programming languages, *in* "Advances in Database Technology—EDBT'90" (F. Bancilhon, C. Thanos, and D. Tsichritzis, Eds), Lecture Notes in Computer Science, Vol. 416, pp. 269–285, Springer-Verlag, New York/Berlin, 1990.
26. D. Stemple *et al.*, "Type-Safe Linguistic Reflection: A generator technology," Res. Rep. CS/92/6, Univ. of St. Andrews, 1992.
27. M. Stonebraker *et al.*, QUEL as a data type, *in* "Proceedings of SIGMOD 84 Annual Meeting" (B. Yormark, Ed.), SIGMOD Record Vol. 14:2, pp. 208–214, ACM Press, New York, 1984.
28. M. Stonebraker *et al.*, Extending a database system with procedures *ACM Trans. Database Systems* **12**, No. 3 (1987), 350–376.
29. M. Stonebraker *et al.*, On rules, procedures, caching and views in data base systems, *in* [9, pp. 281–290].
30. J. Ullman, "Principles of Database and Knowledge-Base Systems," Vol. I, Computer Science Press, 1988.
31. K. Vadaparty, On the power of rule-based languages with sets, *in* "Proceedings 10th ACM Symposium on Principles of Database Systems, 1991," pp. 26–35.
32. M. Vardi, The complexity of relational query languages, *in* "14th ACM Symposium on Theory of Computing, 1982," pp. 137–146.
33. M. Wand and D. Friedman, The mystery of the tower revealed: A non-recursive description of the reflective tower, *LISP Symbolic Comput.* **1** (1988), 12–38.
34. S. B. Zdonik and D. Maier, (Eds), "Readings in Object-Oriented Database Systems," Morgan Kaufmann, Los Altos, CA, 1989.