A Stack-Based Approach to Query Languages*

Kazimierz Subieta[▷] Catriel Beeri[•] Florian Matthes[◇] Joachim W. Schmidt[◇]

[▷]) Institute of Computer Science
Polish Academy of Sciences
Ordona 21
01-237 Warszawa, Poland
subieta@wars.ipipan.war.pl

[•]) Hebrew University
Institute of Computer Science
Givat Ram,
Jerusalem 91904, Israel
beeri@cs.huji.ac.il

[◇]) University of Hamburg
Department of Computer Science
Vogt-Kölln-Straße 30
D-22527 Hamburg, Germany
matthes@dbis1.informatik.uni-hamburg.de

Abstract

Scoping, naming and binding are central concepts in the definition and understanding of programming languages. With the introduction of sophisticated data models, these issues become important for query languages as well. Additionally, the goal of integrating query and programming languages requires a common basis for their operational semantics. We offer here an approach to the definition of the operational semantics of query languages based on an abstract machine, in which names, their bindings, and scopes defined by query and data structure are central. The machine has own simple data model for its store, and has a stack for dealing with scopes. We argue for the generality of the approach and illustrate it by defining the semantics of many query language primitives. Finally, we briefly consider how assignment and procedures can be neatly added.

1 Introduction

A major theme of database research in the last decade is the relationship between declarative, high level query languages and procedural, general purpose programming languages. The standard solution, namely embedding a query language (typically SQL) in the programming language suffers from problems that have been amply recorded in the literature and collectively called *impedance mismatch*. Alternative approaches that have been proposed and extensively investigated include persistent programming languages and database programming languages such as Pascal/R [13], Galileo [1], DBPL [14, 7], Napier88 [8], Machiavelli [12], Taxis [10], and O_2C [11]. The emergence of object-oriented databases is also due, in part, to the feeling that classical database systems do not provide the functionality desired and required for data intensive application development.

Despite their limited algorithmic expressive power, query languages are one of the major achievements of the database domain [2, 15, 16]. This is not only because they are the means for *ad hoc* interactive querying and updating of a database, but rather because they are the basis for high level, declarative, data independent data definition and manipulation. In data manipulation, declarative queries are much easier to optimize than procedural programs. Queries are also potentially parallelizable. The use of queries and set-oriented updates increases programmers' productivity, and program reliability, readability and modifiability. In data definition, queries are used to express integrity constraints, access restrictions, views, snapshots, database procedures, and active rules. However, despite contiuous advances in research and development of database languages and object-oriented database systems, the ideal of "seamless integration" of a programming language with a query language is still elusive.

A primary goal of this paper is to contribute to the understanding of the relationship between query and general programming languages. A major thread in the development of programming languages concerns facilities for modularity, as expressed, e.g., in procedural abstraction in the form of functions and procedures, and data abstraction in the form of ADT's and objects. The use of such facilities emphasizes the relationships between names used in a program, their scope of definition, and the bindings they receive at compile and run-time. We believe that the study of *names, scopes, and bindings* is also important for the design and use of query languages. Understanding of the relevant issues is a crucial step towards

*Work partially supported by a grant from the German-Israel Foundation for Scientific Research and Development.
Draft, February 24, 1994

the successful integration of query and programming languages. The issue is important even if one is interested in query languages only for the relational model, since a query may use a relation name or an attribute more than once with different meanings. However, in the last decade many query languages for more sophisticated models have been defined and studied. In such models, the meaning of a name depends both on where it is defined and where it is used.

This paper proposes an approach to the definition of the operational semantics of query languages centered around the naming-scoping-binding theme. Specifically, we present an abstract machine that we claim to be suitable for the definition of program data bindings. An important component of the machine, that distinguishes it from other machines, is its store, modelling the database as a collection of related entities associated with names to which they can be bound. The store can also contain volatile objects. It is defined in terms of a simple object model. We argue that the simplicity of the model makes our work applicable to a variety of current data models.

The actual binding of names appearing in queries to store objects is achieved by the use of *environments*, where an environment is a collection of local scopes and rules used for associating names with programming objects. The common approach, which we follow here, is that the scopes are organized in a stack, with the "search from the top" rule. Some extensions to the structure of stacks used in the definition and interpretation of programming languages are necessary to accommodate, e.g., the fact that in a database we have bulk data structures, hence possibly multiple simultaneous bindings for a name. This environment stack, and a procedure for evaluation of queries, are the backbone of our approach. An important aspect of the evaluation is parallelism which is inherent in many query primitives, and important for proper definition of update semantics.

The machine and the semantics of a query language are derived from a language implemented in the system LOQIS [17]. However, majority of constructs that we considered exists also in other query languages. In the paper we illustrate how the machine can be used to provide a precise definition of the semantics of some query constructs. (A more comprehensive presentation can be found in [18]). We also illustrate the design of a query language that is directly influenced by the stack-based discipline of scoping and binding. Finally, we briefly consider updates and procedures, thus supporting our claim that this approach facilitates the integration of declarative and procedural constructs.

The rest of the paper is organized as follows. In Section 2 we discuss the abstract store model and present preliminary formal definitions. In Section 3 we discuss the abstract machine model, introducing details of the operational semantics used for the specification of language constructs. In Section 4 we present and discuss various query constructs. In Section 5 we discuss updates and procedures. Section 6 concludes.

2 An Abstract Store Model

The store is the component of the abstract machine that models the database. The operational semantics of queries is defined to a large extent in terms of accesses and manipulations of its elements. The store model and its relationship to other data models are described below.

2.1 The Goals

The primary goal is to serve as a basis for defining the operational semantics of query languages. The model does not include details of physical data organization, indices, storage hierarchies, and buffer management. We view a database as a large collection of (possibly interrelated) entities, as it can be found in modern persistent object stores [9, 3]. The store may contain both persistent and volatile entities; we strive to treat them uniformly.

Current implemented or proposed data models include a large variety of features: data types such as records and arrays; bulk data types such as sets, bags, lists, maps and trees; object features such as identity, *is-a* relationships, methods and encapsulation. We want our approach to semantics to be applicable to a wide range of query languages. One possible approach to addressing this diversity of concepts is to define a rich model that contains "all" features. We have chosen the opposite approach of a very simple model, where various conceptual models, from value-based to object-based, can easily be mapped to it. Semantics for a specific query language over a given model can be defined by a mapping to our model, and by providing the semantics in our model. Although some features found in existing models have no direct mapping to our model, it covers a large variety of features and can be easily

extended. The approach presented in this paper can also be adapted directly (without translation) as another data model.

The ideas treated in the paper do not require a specific (monomorphic, polymorphic) type discipline nor precise data representation details. Types are very important for enforcing static constraints on the use of queries and procedures; however, they are irrelevant to the goals of this paper, hence are not discussed.

2.2 The Model

Intuitively, a stored database (at a given point of time) consists of a collection of stored entities which we call *store objects*[1]. There are three components in a store object: (1) its location; (2) the name that can be used to denote it; (3) the value stored there, i.e., its content. As we are not interested in physical organization, we represent locations abstractly, using the concept of l-value from programming language semantics. Atomic values, records, components of records, conceptual objects, are all store objects: each one of them resides somewhere in memory and needs location for internal identification, in order to be retrieved, updated or deleted. For us, these locations have no structure or meaning, except that they can be used to access objects; queries do not refer to them or use their values in any way. In the sequel, to emphasize their abstract nature, locations are called identifiers.

Names are among the basic building blocks of queries and, in general, of program expressions. Their bindings are an important component of semantics, and a central topic in this paper. While for general programming languages names used in a program are normally defined in the program itself, most of the names used in queries are defined in the database. That is, we are dealing with persistent data where names are an inherent part.

The contents (i.e., the r-value) can be an atomic or complex value, or a location. As for the names, we assume that each store object contains one value. As seen below, more than one value associated with an object is modelled by creating many objects.

Formally, let I be a set of *identifiers*, N be a set of *names*, and V be a set of atomic values. We make no assumption about V; it may contain numerals, strings, graphics, compiled procedures, and so on. Atomicity of the elements of V means that we do not assume the existence of operations that refer to their parts. We assume $I \cap (V \cup N) = \emptyset$; however, N and V are not required to be disjoint. It is possible, even common, that programs compute values, then use them as names, e.g., integers are used as array indices. Generally, some stored values can be used as names of store objects; this property supports genericity of programming. In many languages, for example Ingres/Windows 4GL [4], LOQIS [17] and F-logic [6], data names are first-class citizens.

A *store object* is a triple $< i, n, \vartheta >$, where $i \in I, n \in N$, and ϑ are its identifier, name, and value, respectively. We say that i identifies this object. The value ϑ can be one of the following:

- An atomic value from V.
- An identifier from I. This identifier, as the value of the store object, serves as a (logical) pointer to another store object.
- A set of objects.

We refer to these three types of objects as *value-objects*, *pointer-objects*, and *set-objects*, respectively. Objects of the first two types are called *atomic*, those of the third type are called *complex*. Nesting of set-objects allows us to represent arbitrarily complex objects with hierarchical structure. Below is an example of a complex object, consisting of three atomic objects.

$$< i_5, EMP, \{< i_{51}, NAME, Smith >, < i_{52}, SAL, 2000 >, < i_{53}, WORKS_IN, i_6 >\} >$$

In the following, "object" always means store object, unless specifically noted otherwise.

A *store* is a set S of store objects, and set R of identifiers of designated *root* objects. We make no assumption that elements of R refer to the top hierarchy level of store objects: sometimes elements of R refer to objects nested in other objects. We assume that S satisfies the obvious constraints: An element $i \in I$ is used in it at most once as an identifier; that is, there is a one-to-one correspondence between the set of identifiers used in a database, and the set of objects in it. If an identifier is used as a pointer, then

[1] These should not be confused with the conceptual objects of object-oriented models.

it is also used as an identifier (referential integrity). We also assume that each root identifier actually identifies an object in S. By definition, objects inaccessible from R (directly or indirectly) do not exist. As we will see later, we build over the set R some structure which we will call *environment stack*. The persistence status of objects is irrelevant in our approach; indeed, there is no essential semantic property that can make distinction between querying persistent and volatile data. (The orthogonality of types, queries and persistence is assumed in many modern DBPLs [1, 7, 8, 12, 14, 17].)

We re-emphasize that the identifiers are *internal* – they are not used in queries or programs, and are not printable. A one-to-one mapping of all identifiers to another collection of identifiers cannot be recognized from the outside. The names are *external* in the sense that they are used in queries – they are part of the users' model.

Example 2.1 The following is an example store. The root objects are $\{i_1, i_5, i_9, i_{13}, i_{17}\}$.

MyDatabase:

$< i_1, \quad EMP, \quad \{< i_2, NAME, Brown >, < i_3, SAL, 2500 >, < i_4, WORKS_IN, i_{13} >\} >$

$< i_5, \quad EMP, \quad \{< i_6, NAME, Smith >, < i_7, SAL, 2000 >, < i_8, WORKS_IN, i_{17} >\} >$

$< i_9, \quad EMP, \quad \{< i_{10}, NAME, Jones >, < i_{11}, SAL, 1500 >, < i_{12}, WORKS_IN, i_{17} >\} >$

$< i_{13}, \quad DEPT, \quad \{< i_{14}, DNAME, Toy >, < i_{15}, LOC, Paris >, < i_{16}, LOC, London >\} >$

$< i_{17}, \quad DEPT, \quad \{< i_{18}, DNAME, Sales >, < i_{19}, LOC, Berlin >\} >$

Note that an object WORKS_IN stores an identifier of a department object. If instead it contained, say, a department name, the example would be value-based. If additionally we disallow multiple LOC objects for a department, we obtain a relational database. □

In examples we also use the schema presented in Fig.1. It has redundancy allowing us to demonstrate different styles of querying, e.g. relational and navigational. Arrows denote pointers, and symbols 1 and n denote a 1:n relationship. For example, EMPLOYS denotes pointer-objects inside DEPT objects; a DEPT objects can contain many EMPLOYS objects. Similarly, LOC and PREV_JOB are multi-valued attributes.

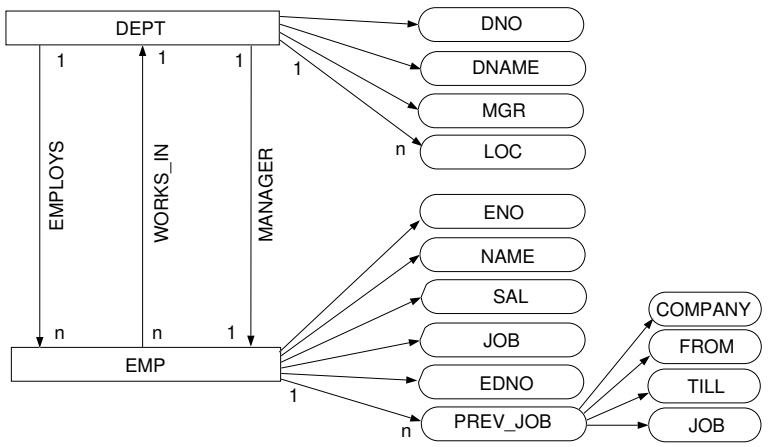

Figure 1: A database schema used in examples

2.3 Discussion

Having defined the model, let us now consider how databases given in one of the common conceptual models can be represented in it. In the relational model, to represent information about employees, we

may use a relation, i.e., a set of tuples, called EMP. Considering the execution of the query **select** SAL **from** EMP, we can see that the name EMP gets *bound to each tuple* of the relation. As we are interested in names in the context of binding, in our model the name EMP is associated not with the relation, but rather with each of its tuples. This holds in general for any named set: in our model the name is associated with each of the elements of the set; see, e.g., LOC of DEPT in MyDatabase.

Speaking in terms of programming languages, a tuple defines a local environment: each attribute is a name that, *in the context of this tuple*, can be bound to the associated object – attribute value. Note that we do not have *tuple* as a data structure in our model – as just seen, a tuple is represented by the set of its named components. Also note that the identifier of the object representing the tuple is internal and has nothing to do with the conceptual model, in which this tuple is a value, not an object. It is just an abstract representation of the fact that the tuple is an identifiable store object. (Many relational systems indeed use tuple-id's internally.)

Names can be numbers. For example,

$$< i_1, A, \{< i_2, 1, Monday >, < i_3, 2, Thuesday >, \ldots, < i_8, 7, Sunday >\} >$$

represents an *array*. Thus, tuples and arrays have essentially the same representation. Indeed, both are essentially local environments, and the differences have to do only with the type and generation of the names that can be used in the environment. As already noted, names used in arrays are also values, and can be calculated during run-time.

We do not impose constraints requiring that objects with the same name should have the same structure. This allows us to represent *variants*, as in the following database fragment:

$$< i_1, EMP, \{< i_2, NAME, Brown >, < i_3, SAL, 2500 >\} >$$
$$< i_6, EMP, \{< i_7, NAME, Smith >, < i_9, IS_STUDENT, TRUE >\} >$$

Similarly, we can represent *null values*, when they mean lack of stored information. Since the definition of a store does not include any concept of format, a null value is represented simply by the absence of a binding. For example, in the database fragment above, the SAL field for Smith is null-valued. We do not associate any semantics with this absence. As far as we are concerned, Smith may have no salary, or his salary may be unknown, or it may be determined in another way, e.g., by using the information that he is a student. The interpretation of the absence of information is part of the semantics of the data, but is *outside* of the scope of our model. Of course, we need the ability to test for the lack of a stored object with a given name, so that the query evaluator can interpret this and apply the appropriate semantics.

To generalize to the nested relational, or complex object models, we need to be able to represent sets, and obviously we have that in the model. We can also represent bags. For example, the object $< i_1, N, \{< i_2, M, 5 >, < i_3, M, 5 >\} >$ represents the bag $\{5, 5\}$. Similarly one can represent lists, trees, and generally, instances of recursively defined data types.

Let us now consider a conceptual object. It has an identity, and an updatable state. The mapping is straightforward – an object is represented as a store object. Object sharing and relationships between objects can be easily mapped by using internal object identifiers as values of pointer sub-objects, as seen in *MyDatabase* above.

In summary, this simple model allows one to represent a variety of data structures and concepts, and their combinations, including tuples, arrays, sets, bags and lists, variants, null values; objects, object sharing, and instances of recursive types. For object-oriented models some simple extensions are needed; we discuss them in [18]. However, we believe that the simplicity of the model is an advantage: it allows us to explain the meaning of query constructs without having to deal with a variety of special cases.

3 An Abstract Machine Model

In this section we present an abstract machine model, on which query language expressions can be evaluated. A component of a state of the machine is a store, as described above. Other components include an environment stack and a result stack for the store of intermediate results. We discuss some commands of the machine, and its facility for parallel execution.

3.1 The Environment Stack

The atomic components of queries are the names and constants. The evaluation of names means binding them to data units, depending on the context. Consider, for example, a query in a relational database that asks for the names of employees satisfying some condition:

select *NAME* **from** *EMP* **where** ...

The evaluation of the query involves finding a set of bindings for *EMP*, then removing some of them which do not satisfy a predicate after *where*. Each binding for *NAME* is then determined in a context associated with one of the bindings to *EMP*; it is bound to the appropriate *NAME* sub-object inside an EMP object.

3.1.1 Stack Organization

While bindings for simple relational queries, such as the one above, seem to be well understood, this is not the case for more complex queries. For example, in a query that selects the salaries of employees that earn more than their managers, *SAL* is used three times, and there is a need to define how each occurrence is bound. The problem is aggravated in models that admit deeply nested structures, possibly, with repeated (sub) attributes. The binding of a name to its meaning may then depend both on the positions of the name in the data structure, and its use in the query.

To deal with this issue, we propose a well known concept of *environment*. Abstractly, an environment is an association of names and objects. It is constructed incrementally, by accumulating local scopes and the scope added last is the first to be visited in a search. Thus, the environment is subdivided into parts, called *sections*, which form the *environment stack*, denoted *ES*.

We assume that each section of *ES* stores identifiers of store objects. Given an identifier, the complete object can be retrieved. In particular, the name stored in the object can be retrieved, to be compared with the name for which a binding is requested. Thus, storing identifiers suffices for establishing bindings.

In a (run-time) stack in programming languages a stack section typically contains local data such as parameters and local variables of a procedure. Such local data *lives* on the stack. In contrast, we are dealing here with global data, that live in the database, and continue to exist not only when a stack section is removed, but also when query evaluation has terminated and the complete stack has disappeared. Thus the objects themselves will be stored in the store, and their identifiers stored in the appropriate stack section(s). This scheme allows the same object to be referenced from several sections (in contrast to the run-time stack where an object occurs in precisely one section). The semantics we are going to deal with depends on this property. Note that in a run-time stack a name occuring in a program is bound to one object. In contrast, we allow a section in *ES* to associate several objects with a name, as e.g. for the EMP name.

In summary, our proposal generalizes the environment stack as used in programming languages in two major ways: (1) The stack contains identifiers of both persistent and volatile store objects, which exist outside the stack; (2) Binding is a multi-valued, rather than a single-valued. This is intimately related to the set orientation and parallelism inherent in query semantics.

In an implementation based on our semantics many optimizations are possible, such as storing names with the identifiers in sections, avoiding long lists of identifiers of objects having the same name, and performing some bindings during compilation. Some of these optimisations are implemented in LOQIS; they are not discussed here.

We assume that at the beginning of the evaluation of a query the environment stack consists of one section containing identifiers of all persistent *root* database objects, such as the *EMP* and *DEPT* objects in *MyDatabase*.

3.1.2 Binding via Stack Operations

Binding a name n is performed by a search for one or more objects associated with that name in the environment stack. The search follows *scope rules*. The simplest rule is to strictly follow the structure of the stack, from the top down towards the bottom. The search terminates when it finds a section that contains one or more bindings for the given name. In that case, the set of *all* objects associated with the name in that section is returned as the result of the search. The result we denote by $search(n)$. If no section with bindings for the given name exists, the search terminates at the bottom of the stack, returning an empty set. This simple search strategy will be changed to accommodate more sophisticated scope rules, e.g., skipping irrelevant stack sections for binding names used inside a procedure.

Changes to the environment have to update the stack by adding or removing sections at its top, using the traditional operations *push* and *pop*. These operations correspond to opening a new scope and closing a scope, respectively.

In programming languages, opening a new scope corresponds to an activation of a block or a procedure. In a query language, it corresponds also to the evaluation of a query component in a context determined by another component. In the general case, this operation depends on both the structure of the query and the structure of the data. Consider, for example, the query $DEPT.LOC$. Each possible binding for LOC is determined by a binding for $DEPT$. That is, first $DEPT$ is bound, in general to many identifiers $i_1, i_2, ..., i_k$ (one for each department). Each identifier i_j identifies a department object, and this object defines a scope in which names such as DNO, DNAME, MGR, LOC are associated with the objects that make up the department object. The semantics of the dot operation is that bindings for LOC are determined in each of these scopes.

Let $o =< i, n, v >$ be an object, with identifier i. We denote by $nested(i)$ the following set of identifiers: If o is a set object, then $nested(i)$ is the set v; if o is a pointer object $< i, n, j >$, then $nested(i) = \{j\}$; if o is a value object, $v \in V$, then $nested(i) = \emptyset$. For uniformity, we extend the function $nested$ for values $v \in V$: $nested(v) = \emptyset$. Then we extend the function to sequences of identifiers and values by defining the result to be the union of the value of the function on the components. For example (c.f. *MyDatabase*), $nested(i_1) = \{i_2, i_3, i_4\}$, $nested(i_4) = \{i_{13}\}$, $nested(i_3) = nested(1800) = \emptyset$, and $nested(< i_1, i_{13} >) = \{i_2, i_3, i_4, i_{14}, i_{15}, i_{16}\}$.

Going back to the evaluation of $DEPT.LOC$, we see that for a given binding i for $DEPT$, the scope in which it makes sense to look for a binding for LOC is $nested(i)$. If this set is pushed as a new scope on the stack then the search for bindings for LOC will find the objects representing the location(s) of the given department, as required. As will be shown, this approach is a core of the definiton of many query operators, including *where*, dot, join, and quantifiers.

3.2 The Query Result Stack

In addition to the environment stack ES, we also need a place for intermediate query results. These are kept on another stack, called the *query result stack* ($QRES$). In particular, at the end of evaluation its top holds the answer to the query. We assume that an intermediate result is always a *table*, where a table is a multi-set (bag) of *rows*, all of the same width. Rows may contain atomic values or identifiers, i.e. their elements belong to $V \cup I$. For uniformity, we represent single values or identifiers as tables having one row and one column (1×1 tables), and do not distinguish between such a table and the element in it. An example table, referring to *MyDatabase*, is presented below; it may represent an answer to the query "Get employee names, their department's names, and 10% of their salary".

i_2	i_{14}	250
i_6	i_{18}	200
i_{10}	i_{18}	150

A table is not in the form we want to see for printed query results. We assume, however, that the visualisation of it, if necessary, can be done by a some procedure (e.g. *display*) taking the table as an argument.

3.3 Set Orientation and Parallelism

The query $DEPT.LOC$ illustrates an important characteristic of queries, namely parallelism. The bindings for LOC are to be performed *for each* binding found for $DEPT$. In principle, choosing a binding for $DEPT$ and then searching for bindings for LOC can be done in any order, even in parallel. For queries, there is no difference between a parallel or a sequential execution in some arbitrary order; however, there is a significant difference when updates are considered. Consider, for example, the update 'increase the salary of each employee by 10% if smaller than the average of the employees' salaries.' In a parallel execution, the average is the same for all updates. In a sequential execution, each update changes the average and thus may influence the next update. To precisely define the semantics of a bulk updates, parallelism is a very useful notion, even if the real implementation will be sequential.

We assume that our machine has sequencing, denoted by ';', parallel loops denoted by *foreach i in C do in parallel...end foreach*, and regular sequential loops denoted similarly, without "in parallel'.' The degree of parallelism is determined by a specified collection: each element i in the collection corresponds

to, or generates an evaluation stream. The element i is available for use, as a parameter, in the stream. We assume that when a parallel loop is started, a separate processor is allocated to each stream. Each processor has its own stacks: $QRES$ is initially empty, and ES is a copy of the common stack. Each branch of the parallel loop continues as a process running one of these processors, and the processors execute concurrently and asynchronously. The streams merge into one stream when they terminate. The store is shared among the parallel streams.

There are issues that we need to worry about in this idealized parallel execution. One is how these streams are merged when they terminate. We assume that when a branch terminates, ES returns to its state at the time just before the loop started; thus, at the end environment stacks are identical for all streams. Considering QRES, when a parallel stream executes, it does not change any part of the stack below the level at the starting time, and that at the end the local stack contains one cell storing the partial result. When the parallel branches merge, these tables are merged, forming one table at the top of the common $QRES$. The merge function depends in general on on the query operator being actually evaluated. æ

4 The Language SBQL

In this section we illustrate how the semantics of query language primitives, and hence also complete query languages, can be defined on our machine. We sketch here the definition of an untyped language, called SBQL (Stack-Based Query Language). For lack of space, we present only some of the constructs. A few update and procedural constructs are presented in the next section.

4.1 The General Framework

The structure of the language is simple: Any constant or name is an *atomic* query. From such queries we build complex queries by applying unary or binary operators. For example, EMP, SAL, 1800 and 2 are queries, from which we can build such queries as EMP *where* $(SAL > 1800)$ and $2 * 2$. With the exception of typing constraints (e.g., one cannot multiply two sets of strings), we assume full orthogonality of operators. Parentheses are allowed, precedence rules are not discussed here.

To define the semantics of the language, we introduce a recursive procedure *eval* that maps a syntactically correct query and a machine state to a result. The state consists of a store and a state of the environment stack. The procedure is defined by cases, one for each operator. It may change the environment stack; however, the state of the stack after evaluation is always the same as before evaluation. The result of a query is left as a new cell at the top of $QRES$.

4.2 Atomic Queries

We start with atomic queries, namely constants and names. We assume that constants that denote some elements of V are available in the language. For the query c, where c is a constant, $eval(c)$ leaves the 1×1 table $\{c\}$ at the top of $QRES$. For the query n,where n is a name, $eval(n)$ searches for bindings for n is ES, and leaves the result, $search(n)$, at the top of $QRES$. For example, the query EMP, applied to $MyDatabase$, returns a single-column table containing identifiers i_1, i_5, i_9.

4.3 Compound Queries: Algebraic Operators

An operator is called *algebraic* if its effect is defined purely in terms of its argument(s), and is independent of the environment. By convention, such an operator takes its argument(s) from $QRES$ (popping the cells that contain them) and pushes its result there. Examples of algebraic operators include operations on atomic values such as arithmetic operations and comparisons, string operations, boolean operations, and equality tests of identifiers. They also include aggregates applied to collections, and set operations such as set union, product, tests for containment, emptiness, and so on.

The semantics of a binary algebraic operator Θ is defines as follows:

case *query* is $q_1 \Theta q_2$ (Θ algebraic):
  begin
    $eval(q_1)$;

$eval(q_2)$;
$apply(\Theta)$
end

By our assumption on the state of ES before and after a call to *eval*, both queries are evaluated in the same environment. We assume that *apply* is defined without any use of the environment. The semantics of operations such as set union or product may be defined using the same pattern. Dereferencing may be needed, dependening on the operator being applied; for example, $SAL > 2500$ requires dereferencing of an identifier returned by SAL. For the space limit we ommit further discussion.

4.4 Compound Queries: non-Algebraic Operators

The general syntax for application of a non-algebraic operators is $q_1 \Theta q_2$, where q_1 and q_2 are (arbitrarily complex) queries, and Θ is an operator. The general pattern that defines a part of the *eval* procedure for queries with a non-algebraic operator is as follows.

case query is $q_1 \Theta q_2$ (Θ non-algebraic):
begin
$eval(q_1)$;
for each row r in $top(QRES)$ do in parallel
push($ES, nested(r)$); (* Open a new scope on ES *)
$eval(q_2)$;
$partial_result := combine(r, top(QRES), \Theta)$;
pop(ES); pop($QRES$); (* Restore the state of local processors *)
end for each;
$merged_result := merge(partial_results, \Theta)$;
pop($QRES$); (* Remove from QRES the table created by q_1 *)
push($QRES, merged_result$);
end

Before the evaluation of q_2, many parallel processors are activated, one for each row r from the table returned by q_1. Each processor opens an own new scope $nested(r)$, then evaluates q_2. A partial result returned by it depends on the result returned by q_2, r and Θ. Finally, partial results are merged; the merging function depends on Θ.

Note that the evaluation is not symmetric in the two subqueries, since q_2 is evaluated in environments determined by the result of q_1. In the rest of the section we define and illustrate various non-algebraic operations that exploit these environment stack dependencies.

4.5 Selection

The syntax for the selection operation is q_1 *where* q_2, where q_1 is any query, and q_2 is a boolean-valued query, i.e., a condition. The semantics is defined by the scheme above, where the function *combine* is defined by:

$$combine(r, top(QRES), where) \equiv \textbf{if}\ top(QRES) = TRUE\ \textbf{then}\ \text{return}\ \{r\}\ \textbf{else}\ \text{return}\ \emptyset$$

The *merge* function is the union of partial results.

Example 4.1 (C.f. *MyDatabase*).
Consider EMP *where* $(SAL > 1800)$ expressing the query"List employees earning more than 1800". We follow its execution. Initially, the environment stack contains one section containing identifiers of all root database records:

$$\boxed{i_{1(EMP)}, i_{5(EMP)}, i_{9(EMP)}, i_{13(DEPT)}, i_{17(DEPT)}}$$

First, *eval* is called with the atomic query EMP. The name EMP is bound to three objects, whose identifiers are put in a single-column table in $QRES$:

i_1
i_5
i_9

Each row is processed by the body of *for each*. For the row containing i_1, we have $nested(i_1) = \{i_2, i_3, i_4\}$. This set is pushed onto ES:

$i_{2(NAME)}, i_{3(SAL)}, i_{4(WORKS_IN)}$
$i_{1(EMP)}, i_{5(EMP)}, i_{9(EMP)}, i_{13(DEPT)}, i_{17(DEPT)}$

Now $SAL > 1800$ is evaluated. The predicate $>$ is algebraic, so subqueries SAL and 1800 are evaluated with the same environment stack. The evaluation goes through the following states of local $QRES$ (which is assigned to a processor processing i_1):

$eval(SAL)$:

i_3

$eval(1800)$:

1800
i_3

Dereferencing:

1800
2500

Comparison:

$TRUE$

Since the predicate returns $TRUE$, the row i_1 is included into the result.

The same action is performed concurrently for the rows i_5, for which the predicate is also $TRUE$, and i_9, for which the predicate returns $FALSE$. At the end, upon exit from the *for each*, the environment stack will be the same as at the beginning. Merging individual results of parallel streams is collected at the top of $QRES$ as a table $\{i_1, i_5\}$. □

4.6 Joins

Using the product and the selection, we can define joins. Consider the query

$$(DEPT \times EMP)\ where\ (DNO = EDNO).$$

As described above, the query $DEPT \times EMP$ leaves on $QRES$ a two-column table where each row is a pair of identifiers of $DEPT$ and EMP objects. A stream in the evaluation of the *where* clause will push onto ES $nested(\{i, j\})$ for one such pair, then evaluate the condition. The top section now contains identifiers of objects whose names are the attributes of the $DEPT$ and EMP collections. The condition $DNO = EDNO$ is evaluated on attributes of the two objects. The final result is a set of pairs of identifiers, one from each collection, that identify pairs of objects for which the condition is true.

4.7 Projection, Navigation, Path Expressions

We need the ability to retrieve subobjects of previously identified objects. This is provided by the 'dot' operator. When used more than once, it allows navigation in an object graph using path expressions. The syntax is $q_1.q_2$, where q_1 and q_2 are queries. To avoid parenthesis in long path expressions, we assume association to the left. The definition of the semantics is very similar to the semantics of *where*, except that the *combine* function simply returns the table returned by q_2.

Example 4.2

$$EMP.SAL$$

returns a single-column table, in which each row is the identifier of a SAL object nested inside some EMP object, for example, i_3.

$$(EMP\ where\ NAME = "Smith").WORKS_IN.$$

The evaluation of the first part terminates with ES containing only the initial section, and $QRES$ containing the table $\{i_5\}$. Now, to evaluate the '$.WORKS_IN$' part, we push on ES the section $nested(i_5) = \{i_6, i_7, i_8\}$. The evaluation terminates with $\{i_8\}$ at the top of $QRES$. □

To find the name of the department where Smith works, we have to write

$$(EMP\ where\ NAME = "Smith").WORKS_IN.DEPT.DNAME$$

An evaluation strategy requires $DEPT$ in the path expression to be able to access $DNAME$. This is a viable option, however, it makes path expressions longer. Most languages for object-oriented databases have chosen to interpret the "dot" operation so that $EMP.WORKS_IN.DNAME$ is a legal query.

To follow this tendency, we could introduce a new function *nested*, which for a pointer object returns its r-value. However, this will make it difficult to treat updates. How do we define assignment of a new $DEPT$ identifier to the $WORKS_IN$ attribute of Smith? We need the query that returns the l-value for the assignment, which in this case is i_8, but with the currently considered option the query returns the r-value, i_{17}. We can also consider elliptic syntax, which does not change semantics, but allows us in some contexts to omit some elements of a full query. These issues are not considered in this paper.

Example 4.3

$$EMP\ where\ SAL > ((EMP\ where\ NAME = "Smith").SAL)$$

This query combines nested *where* clauses with 'dot'. It expresses the query "List employees earning more than Smith". Note that the binding of the second occurence of EMP is performed on ES containing two sections, e.g.

$i_{2(NAME)}, i_{3(SAL)}, i_{4(WORKS_IN)}$
$i_{1(EMP)}, i_{5(EMP)}, i_{9(EMP)}, i_{13(DEPT)}, i_{17(DEPT)}$

and will again return $\{i_1, i_5, i_9\}$, and binding of the second occurence of SAL is performed on the stack containing three sections, e.g.

$i_{6(NAME)}, i_{7(SAL)}, i_{8(WORKS_IN)}$
$i_{2(NAME)}, i_{3(SAL)}, i_{4(WORKS_IN)}$
$i_{1(EMP)}, i_{5(EMP)}, i_{9(EMP)}, i_{13(DEPT)}, i_{17(DEPT)}$

and will return (in this case) i_7. □

It is easy to see that 'dot' is an associative operation: $(q_1.q_2).q_3$ is equivalent to $q_1.(q_2.q_3)$; thus, parentheses are not needed.

Example 4.4

$$(EMP\ where\ NAME = "Smith").WORKS_IN.DEPT.EMPLOYS.EMP.(NAME \times JOB)$$

returns a table containing (identifiers of) the name and job of the employees working in the same department as Smith. This query illustrates the use of product for constructing multi-component target list, which may be important in the context of multivalued attributes. □

Path expressions for navigation in object databases have been proposed in many papers, e.g. [5]. We believe that our approach is very general, fully orthogonal to other query operators, and semantically clean.

4.8 Navigational Join

The 'dot' operation allows one to 'walk' on a path, but the result is always the end point of the path. Another interesting operation is *navigational join*, that forms pairs containing the start and end points of a path. For example, we might want to create a set of pairs of employees and the departments they work in. In relational systems such an operator can be defined as a product followed by a selection, but in object-oriented systems it is often more natural to express it directly as a navigation through a pointer-valued attribute.

The syntax is $q_1 \bowtie q_2$. Let $\otimes$ denote the cartesian product generalized to bags. The semantics of the operation is defined by the same scheme as above, except for the *combine* function, which is defined as follows:

$$combine(r, top(QRES), \bowtie) \equiv return(\{r\} \otimes top(QRES))$$

Thus, each row r returned by q_1 is combined with each row returned by q_2 for this r. As previously, the final result is a union of partial results.

Example 4.5

$$EMP \bowtie (WORKS_IN.DEPT)$$

returns a two-column table, where each row contains the identifiers of an employee and the identifier of his department.

$$EMP \bowtie (DEPT\ where\ EDNO = DNO)$$

is a relational variant of the previous example. It shows that the construct is general. In $q_1 \bowtie q_2$, the second component q_2 can be any query, in particular, a function of the elements returned by q_1.

$$EMP \bowtie (WORKS_IN.DEPT.(DNAME \times LOC))$$

returns a three column table, where identifiers of EMP are associated with identifiers of $DNAME$ and LOC of the department in which the employee works.

$$DEPT \bowtie avg(EMPLOYS.EMP.SAL)$$

returns a table of pairs consisting of a department identifier and the average salary of its employees. In SQL this query requires the *group by* operator. In our approach we achieve the same goal using clean functional semantics and orthogonality. □

4.9 Quantifiers

The syntax is $Qq_1(q_2)$, where Q is either $\forall$ or $\exists$, q_1 is an arbitrary query, and q_2 is an arbitrary boolean-value condition. Thus, Q is regarded as a binary operation, similar to previous non-algebraic operations. The semantics is also defined similarly. The differences are in the *combine* function, and in how the results of the parallel evaluation streams are merged. The *combine* function simply returns the truth value returned by q_2. For $\forall$ the result is $TRUE$ if q_1 returns an empty table, or the boolean conjunction of all returned truth values otherwise; for $\exists$ the result is $FALSE$ in the first case, or the booloean alternative of partial results otherwise.

Example 4.6 Departments where all programmers used to work for IBM:

$$DEPT\ where\ \forall\ (EMPLOYS.EMP\ where\ JOB = "programmer")$$
$$(\ \exists\ PREV_JOB\ (COMPANY = "IBM"))$$

□

Usually quantifiers are associated with auxiliary variables. The next sub-section addresses this problem.

4.10 Correlation Variables and Synonyms

Most logical calculi rely strongly on the use of variables; so do many query languages. However, the concept of variable as used in these languages is quite different from its standard use in programming languages as a cell used to hold an updatable value. In calculi and query languages, variables are used as temporary names: either to provide synonyms for stored relations (to avoid name conflict), or as iterators over sets. The SQL phrase *from EMP e*, introduces e for a similiar purpose.

Semantics of such auxiliary names is not always obvious. An example is the DBPL construct [14]:

$$FOR\ EACH\ x\ IN\ EMP : x.JOB = \text{“}clerk\text{''}\ DO\ x.SAL := 3000;\ END;$$

where x is used both as a calculus variable and as a programming variable. Achieving all goals of auxiliary naming, (i.e. local synonyms for objects with statically determined scope, naming of a query result or its parts, iterators/cursors, updating via auxiliary names, easy and clean semantics, avoiding semantic anomalies) is difficult. The proposal below (implemented and tested in LOQIS) is a solution that in our opinion is very simple, universal, and free of many disadvantages of other solutions.

The basic idea is that an auxiliary name can be used to refer object temporarily, i.e., only in the context of a subquery. This blends naturally with our scoping-based approach. We distinguish two kinds of occurrences of auxiliary names in a query: declaration and use. The syntax for the declaration is $n \leftarrow q$, where q is an arbitrary query. The declaration associates the name n with each row in the result of q. For row r returned by q, $n(r)$ denotes the corresponding row returned by $n \leftarrow q$. We allow also such named rows to appear in ES. To be consistent with our previous definitions, we have to improve the function *nested*: for arguments $n(r)$ it is an identity function. This means that during opening a new scope such elements are copied to ES without changes. The binding works as usual, but with a

small difference: when name n occured in a query is bound to the object $n(r)$ on ES, only r is pushed on $QRES$; name n is not propagated to the result.

Example 4.7 (C.f. MyDatabase.)

$$(x \leftarrow EMP)\ where\ (x.SAL > 1800)$$

As before, the atomic query EMP returns a single-column table $\{i_1, i_5, i_9\}$. Application of $x \leftarrow$ changes this table into $\{x(i_1), x(i_5), x(i_9)\}$. Assume $x(i_1)$ is processed by the *where* operator. This means the application of *nested*, which pushes $x(i_1)$ without changes on ES. Name x in the predicate after *where* is bound to this element, but only i_1 is send to $QRES$; name x is not propagated. The dot operator in a subquery $x.SAL$ pushes $nested(i_1) = \{i_2, i_3, i_4\}$ on the top of ES; thus SAL can be bound, as usual, to i_3. Then, the dereferencing and comparison operators will return $TRUE$. In the result, the whole query will return a single-column table $\{x(i_1), x(i_5)\}$.

The reader can verify that the SQL-style query

$$(((x \leftarrow (EMP\ where\ JOB = "clerk")) \times (y \leftarrow EMP) \times (z \leftarrow DEPT))$$
$$where\ (x.SAL > y.SAL \wedge x.EDNO = z.DNO \wedge z.MGR = y.ENO)).x$$

returns identifiers of clerks earning more than their managers. Besides the SQL style, this general mechanism of auxiliary names allows for many other styles (in particular, the domain calculus style). □

5 Updates and Procedures

In this section we briefly consider how bulk updates and procedures can be integrated into the language,

5.1 Assignment

The semantics of an assignment $l := r$ assumes that the left-hand side evaluates to an l-value, and the right-hand side evaluates to an r-value; the r-value is assigned as the new value associated with the l-value.

Our approach to assignment as a general operation generalizes the *update* statement of SQL. The general syntax is $q_1 := q_2$. It can be combined with previous queries in the form $q.(q_1 := q_2)$. To understand the issues in using such an operation, consider

$$(EMP\ where\ SAL > 1800).(WORKS_IN := (DEPT\ where\ DNAME = "Toy"))$$

The first part of the query terminates with $\{i_1, i_5\}$ on the top of $QRES$. The '.' evaluation starts streams, pushing the corresponding scopes on ES. Consider the stream for i_5. The first argument of the assignment evaluates to an l-value, i.e., to i_8. The second argument evaluates to the identifier i_{13}. The assignment updates the object $< i_8, WORKS_IN, i_{17} >$ to $< i_8, WORKS_IN, i_{13} >$. Note that it is crucial here that $WORKS_IN$ returns a l-value for a $WORKS_IN$ objects (i_8), not its r-value (i_{17}).

The example illustrates an issue that needs to be resolved for bulk updates to be well defined. To avoid semantic ambiguity we assume that each of parallel streams prepares only an intention list of updates to be performed, relying on a state before the updating, common to all parallel streams. When streams are merged, their lists are merged. During merging inconsistences can be discovered, e.g. more than one stream attempted to update the same object. To take into account a more complex case of nested updates (i.e. when the update is of the form $q.p$, where q is a query and p is an arbitrary updating program) we must assume the transactional semantics: each parallel stream can see only own updates, and any inconsistency or conflict between parallel stream reverts the database to the initial state.

We can easily do multiple updates in a single stream (';' denotes sequencing):

Example 5.1 $(EMP\ where\ JOB = "clerk").(SAL := SAL + 100;\ JOB := "officer")$ □

5.2 Procedures

Since the semantics of procedures is also defined by using a stack, we can easily incorporate procedures into our framework. There are, nevertheless, many issues to be considered, e.g. methods of parameter transmission. For lack of space, we only illustrate the essential idea. A procedure can introduce local

objects, which are removed when the procedure is terminated. Parameters are treated similarly. We assume static scoping, i.e., if p_1 calls p_2 then local objects of p_1 are not visible from the inside of p_2. Local objects can be easily implemented by storing them as volatile named objects, and storing the identifiers on a new cell of ES. When the cell containing an identifier is popped, the object is gone. Output from functional procedures is pushed onto $QRES$.

A procedure declaration has the usual format, with a header that contains a formal parameter list. A call has the form

$<$*procedure name*$>$ $(p_1 : q_1; \ldots)$,

where p_i is a name of the i'th formal parameter, and q_i is a query that provides a value for it. Local objects (i.e., variables) are declared by

create local $<$ *specification of the object* $>$

Example 5.2 A functional procedure 'poor' has a list of jobs as a parameter. It returns pointers to names, salaries, and department names of employees, who do one of the specified jobs and earn less than the average.

procedure $poor(JOBS)$
  begin
    create local $AVERAGE(avg(EMP.SAL))$;
    (* Creating a set of pointer objects pointing poor EMPs *)
    create local pointers $POOR(\ EMP\ where(JOB \in JOBS \wedge SAL < AVERAGE))$;
    return $POOR.EMP.((N \leftarrow NAME) \times (S \leftarrow SAL) \times (D \leftarrow (WORKS_IN.DNAME)))$;
  end *poor*;

A query that retrieves names of poor clerks and programmers from the department "*Sales*":

$(poor(JOBS : \{"clerk", "programmer"\})\ where\ (D = "Sales")).N$

Increase salaries of poor programmers from the department "*Sales*" by 100:

$(poor(JOBS : "programmer")\ where\ (D = "Sales")).(S := S + 100)$ □

Example 5.3 Define a view $MyView(Dname, AvgSal, Mgr(Name, Sal))$ containing information about department names, average salaries, and manager names and salaries for departments located in Paris.

procedure $MyView()$
  begin
    $return(DEPT\ where\ "Paris" \in LOC).($
      $(Dname \leftarrow DNAME)\times$
      $(AvgSal \leftarrow avg(EMPLOYS.EMP.SAL))\times$
      $(Mgr \leftarrow (MANAGER.EMP.((Name \leftarrow NAME) \times (Sal \leftarrow SAL)))))$
  end $MyView$;

A query on the view: Give manager name for the department with the highest average salary:

$(MyView\ where\ AvgSal = max(MyView.AvgSal)).Mgr.Name$

An update through the view: Increase by 200 the salary of the manager of the Sales department:

$(MyView\ where\ Dname = "Sales").Mgr.(Sal := Sal + 200)$ □

6 Conclusions

We have presented an approach to query languages based on a modification of concepts that are well known in programming languages. It makes possible to build query languages for variety of data models,

including relational and object-oriented models. The approach supports pragmatic universality, orthogonality, and precision of specification of semantics. We have shown that a modified stack-based mechanism can be used to define and process declarative queries, and that it makes possible seamless integration of query constructs with imperative constructs and programming abstractions.

For space limit, we do not discuss many issues relevant to this approach, in particular, static polymorphic typing, object orientation, general transitive closures, ordering, processing of null values and variants, various procedural constructs, methods of parameter transmission in procedures, optimization, active rules and event-driven programming. Many of them are implemented in LOQIS and they will be subjects of subsequent papers.

References

[1] A. Albano, L. Cardelli, R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. ACM Transactions on Database Systems, Vol.10, No 2, 1985, pp.230-260

[2] C. Beeri. Formal Models for Object-Oriented Databases. Proc. 1st DOOD Conf., Kyoto, pp.370-395, 1989.

[3] A.I. Brown, R. Morrison. A Generic Persistent Object Store. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/39, 1992

[4] Language Reference Manual for INGRES/Windows 4GL for the UNIX and VMS Operating Systems. INGRES Release 6, Ingres Corporation, August 1990.

[5] M. Kifer, W. Kim, Y. Sagiv. Querying Object-Oriented Databases. Proc. ACM SIGMOD Conf. pp.393-402, 1992.

[6] Kifer, M. and G. Lausen F-Logic: A higher-order language for reasoning about objects, inheritance, and scheme. *Proc. SIGMOD Conf.*, June 1989, pp. 134–146.

[7] F. Matthes, A. Rudloff, J.W. Schmidt, K. Subieta. The Database Programming Language DBPL, User and System Manual. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/47, 1992

[8] R. Morrison, F. Brown, R. Connor, A. Dearle. The Napier88 Reference Manual. Universities of St Andrews and Glasgow, Departments of Comp. Science, Persistent Programming Report 77, July 1989.

[9] E.J. Moss. Design of the Mneme Persistent Object Store. ACM Transactions on Information Systems, Vol.8, No.2, April 1990, pp.103-139

[10] J. Mylopoulos, P.A. Bernstein, H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications. ACM Transactions on Database Systems, Vol.5, No 2, 1980, pp.185-207

[11] The O_2 User Manual, Version 4.1. newblock O_2 Technology, Versailles, France, October 1992

[12] A. Ohori, P. Buneman, V. Breazu-Tannen. Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. Proc. of ACM SIGMOD 89 Conf., 1989, pp.46-57

[13] J.W. Schmidt. Some high level language constructs for data of type relation. ACM Transactions on Database Systems, Vol.2, No 3, 1977, pp.247-261

[14] J.W. Schmidt, F Matthes. The Database Programming Language DBPL, Rationale and Report. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/46, 1992

[15] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. IEEE Transactions on Knowledge and Data Engineering, 2:1, pp.125-142, 1990.

[16] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech: The Committee for Advanced DBMS Function. Third-Generation Data Base System Manifesto. ACM SIGMOD Record 19(3), pp.31-44, 1990.

[17] K. Subieta. LOQIS: The Object-Oriented Database Programming System Proc.1st Intl. East/West Database Workshop on Next Generation Information System Technology, Kiew, USSR 1990 Springer Lecture Notes in Computer Science, Vol.504, pp.403-421, 1991.

[18] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt. A Stack-Based Approach to Query Languages. Institute of Computer Science Polish Academy of Sciences, Report 738, Warszawa, December 1993.