Procedures in Object-Oriented Query Languages

Kazimierz Subieta Yahiko Kambayashi
Integrated Media Environment Experimental Lab.
Faculty of Engineering
Kyoto University, Sakyo, Kyoto 606-01, Japan
{subieta,yahiko}@kuis.kyoto-u.ac.jp

Jacek Leszczyłowski
Institute of Computer Science
Polish Academy of Sciences
Ordona 21, 01-237 Warszawa, Poland
jacek@ipipan.waw.pl

Abstract

We follow the stack-based approach to query languages which is a new formal and intellectual paradigm for integrating querying and programming for object-oriented databases. Queries are considered generalized programming expressions which may be used within macroscopic imperative statements, such as creating, updating, inserting, and deleting data objects. Queries may be also used as procedures' parameters, as well as determine the output from functional procedures (SQL-like views). The semantics, including generalized query operators (selection, projection, navigation, join, quantifiers, etc.), is defined in terms of operations on two stacks. The *environment stack* deals with the scope control and binding names. The *result stack* stores intermediate and final query results. We discuss definitions of object-oriented concepts and present variants of parameter passing methods. Finally, we indicate a potential of the approach for query optimization based on rewriting.

1 Introduction

In the stack-based approach to object-oriented query languages (QLs) [SBMS93, SBMS94] we reconstruct QLs' concepts from the point of view of programming languages (PLs).The most fundamental property of QLs integrated with PLs concerns the presence of *conceptual* operators that encapsulate iterations. Conceptual operators have the intuitive meaning for the programmer, allowing her/him to write, read and understand programs fluently. On the other hand, the operators should have formal and simple semantics. Projection, selection, join, quantifiers are examples of relational query operators that encapsulate iterations and are at the same time conceptual, formal and simple. For more sophisticated data models, in particular, for object-oriented models, sufficiently general and semantically clean definitions of such operators cause difficulties. These are due to the complexity and variety of data structures that have to be queried, and the necessity to integrate queries with various data/PL functionalities, such as macroscopic imperative statements, views, integrity constraints, (database) procedures, active rules, modules, methods, classes, and types.

In this paper we present a new conceptual framework (a formal ideological platform) aiming the mentioned above issues. The model that we propose is abstract and universal, but simultaneously, precise enough to deal with vital semantic aspects of practical object-oriented QLs/PLs. As one can expect, it subsumes corresponding features of less sophisticated data models, in particular, of relational, nested-relational and functional ones.

The main syntactic decision of our approach is the unification of PL expressions and queries; queries remain as the only kind of PL expressions. Concerning semantics, we focus on the naming-scoping-binding issues. Each name occuring in a query is *bound* to runtime program entities (persistent data, procedures, actual parameters of procedures, local procedure objects, etc.) according to the actual *scope* for the name. The common approach which we follow here is that the scopes are organized in an environment stack with the "search from the top" rule. Some extensions to the

Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995

structure of stacks used in PLs are necessary to accommodate (in particular) the fact that in a database we have persistent and bulk data structures. Hence the stack contains data identifiers rather than data themselves (i.e., we separate the stack from a store of objects), and possibly multiple objects can be simultaneously bound to a name, which makes the macroscopic processing possible. The operational semantics of queries and procedures is defined in terms of operations on two stacks: the environment stack and the stack of query results.

Basic assumptions of our approach to the integration of queries and procedures are the following:

- ▷ Queries, as generalized programming expressions, are used as parameters of procedures.
- ▷ The body of a procedure is a sequence of imperative statements that use queries in the *many-data-at-a-time* manner (c.f. the SQL *update* statement).
- ▷ The output from a functional procedure is determined by a query. The output may be accessed through auxiliary names, as in SQL views.
- ▷ Procedures may call procedures, in particular, can be recursive.
- ▷ Procedures may declare/create local objects that are unvisible from outside of the procedure body. To enable recursion, local objects are assigned to a run-time procedure invocation rather than to a procedure code.
- ▷ We allow three methods of updating data objects in the *many-data-at-a-time* manner with the use of a procedure: (1) updating via parameters, (2) updating via side effects, and (3) updating via objects' references returned by a functional procedure (the method corresponds to view updating).
- ▷ We deal with scoping rules that take into account modularization, encapsulation, inheritance, and procedures local to a class, i.e. methods.

Majority of constructs described in this paper are implemented in the experimental system LOQIS [SMA90], but we discuss here principles that are independent of the implementation. We dedicate this work to the developers of industrial standards, such as ODMG-93 [Catt94] and SQL-3 [ANSI94]. In our opinion, some drawbacks of these proposals could be eliminated by following principles that stem from the PL state-of-the-art. We present consequences of these principles for a general and disciplined semantic model which is relevant to these proposals.

Why the stack-based approach?

The ideas presented in this paper contribute to a very hot research and technological area, thus have to compete with a tremendous amount of proposals: formal and informal, implemented and speculated[1]. The comparison of our approach with informal ones is a bit difficult, as the discussion has to involve a taste, belief and backgrounds of competing parties, which are incompatible as a rule. We can only indicate lack of some kinds of queries and/or data structures in a particular approach, which are available in ours[2] (for example, *"For each department having more than 50 programmers return the name and the dispersion of salaries"*, assuming the aggregate function *dispersion* is not available, and the request should be formulated as a single query addressing a relational or an object database). Besides the universality, our approach is regular, minimal and semantically clean, which is not the case of informal proposals as a rule. Concerning formal approaches, we argue that currently known ideas (nested relational algebras, object algebras, relational calculi, predicate logic, F-logic, comprehensions, etc.) **must** lead to semantic difficulties at least for one of the following reasons:

- ▷ Lack of uniform, orthogonal treatment of transient, persistent, individual and bulk data;
- ▷ Lack of consistent and universal approach to updating, and in general, to the integration with imperative statements;
- ▷ Absence of scoping rules for names used in queries/programs. The control over scopes for names is necessary to deal with (recursive) procedures, modules, program blocks, classes, methods, procedure parameters, etc.;
- ▷ Inability to deal with object sharing, encapsulation, class hierarchy and inheritance;
- ▷ No clean semantics of auxiliary names used in queries, such as tuple variables (SQL "synonyms"), domain variables, variables bound by quantifiers, cursors in *for each* statements, and new names used in (SQL-like) views;
- ▷ Lack of object identifiers as a QL primitive, which makes impossible to deal with important practical functionalities (for instance, *call-by-reference*);
- ▷ Absence of views and view updating;
- ▷ Absence of null values and variants in data structures and handling them in queries;
- ▷ Absence of ordering and queries based on the ordering (*"Get 20 best-paid employees"*);
- ▷ Lack of compositionality - big syntactic/semantic patterns of QLs' constructs, making minimality, orthogonality and universality of a language very problematic (c.f. the famous "query" $2+2$).

The comprehension syntax, for example, falls into

[1] They are too numerous to cite even the most important.

[2] The reverse statement is the task of our opponents. So far we didn't discover a case showing that our approach has some inherent limitations in comparison to another approach.

all of the mentioned problems, F-logic is not dealing with updates, views, scopes, null values, and ordering; etc. Our approach allows us to deal with all these issues in a consistent framework, which can be explained on few pages of this paper. With respect to other approaches, the level of universality of our approach is incomparably higher; this concerns both data structures and QL/PL functionalities. We also argue that our approach contributes a lot to the understanding the true nature and consequences of semantic decisions in practical query/programming languages that are actually used, implemented or designed.

For the reader competent in the PLs' state-of-the-art it should be evident that a serious approach to the integration of queries and procedures cannot avoid the concept of an environment stack (or an equivalent concept). Our original contribution is that we use the same stack as a part of an abstract, universal mechanism for defining all query operators, such as selection, projection, join and quantifiers.

The rest of the paper is organized as follows. In Section 2 we present an abstract store model. In Section 3 we describe the environment stack. In Section 4 we outline the operational semantics of the query language SBQL; it is a formalized variant of many SQL-like QLs. In Section 5 we present imperative constructs based on queries. In Section 6 we present our view on object-oriented concepts. In Section 7 we introduce procedures and discuss parameter passing. In Section 8 we present an example of optimization based on rewriting. Section 9 contains the conclusion.

2 An Abstract Store Model

Currently 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. Instead of addressing all this diversity we have chosen the opposite approach, developing a very simple model, powerful and general enough to express various models.

Our store consists of *store objects* (*objects* for short) which can be volatile or persistent. There are three components of an object:

- its unique *internal identifier*; it cannot be directly written in queries and is not printable;
- the *external name* invented by the programmmer or database designer that can be used to access the object from a program;
- the *content* of the object which can be a value, a pointer, or a set of objects.

Formally, let I be a set of *identifiers*, N be a set of *names*, and V be a set of *atomic values*. Atomicity of the elements of V means that they are considered to have no components. We assume $I \cap (V \cup N) = \emptyset$; N and V are not required to be disjoint. A *store object* is a triple $<i, n, v>$, or $<i, n, i_1>$, or $<i, n, T>$ where $i, i_1 \in I, n \in N$, and T is a set of store objects. i is an internal identifier of the objects, n is an external name, and v, i_1 and T are contents of the objects. We say that identifier i *identifies* an object. We refer to these three types of objects as *value objects*, *pointer objects*, and *complex objects*. A *store* is a set S of store objects, and set R of identifiers of designated *root* objects.

Note that complex objects can represent arbitrary hierarchical data structures, and pointer objects can be used to represent object sharing. To deal with encapsulation, classes and inheritance some extensions are necessary; we present them later. Types are (almost) orthogonal to the topics considered in this paper thus we do not deal with them. Note that to deal with bulk data we do not impose uniqueness of external names of objects at any level of data hierarchy. We have shown in [SBMS93, SBMS94] that this simple model allows one to represent a variety of data structures and concepts, including tuples, (nested) relations, arrays, sets, bags, null values, variants, instances of recursive types, etc.

The following is an example store. The root objects of the store are identified by $\{i_1, i_5, i_9, i_{13}, i_{17}\}$.

TinyDatabase:

$< i_1,$	*EMP*, {	$<i_2$, *NAME, Brown*>,	
		$<i_3$, *SAL*, 2500>,	
		$<i_4$, *WORKS_IN*, $i_{13}>$	} >
$< i_5,$	*EMP*, {	$<i_6$, *NAME, Smith*>,	
		$<i_7$, *SAL*, 2000>,	
		$<i_8$, *WORKS_IN*, $i_{17}>$	} >
$< i_9,$	*EMP*, {	$<i_{10}$, *NAME, Jones*>,	
		$<i_{11}$, *SAL*, 1500>,	
		$<i_{12}$, *WORKS_IN*, $i_{17}>$	} >
$< i_{13},$	*DEPT*, {	$<i_{14}$, *DNAME, Toys*>,	
		$<i_{15}$, *LOC, Paris*>,	
		$<i_{16}$, *LOC, London*>	} >
$< i_{17},$	*DEPT*, {	$<i_{18}$, *DNAME, Sales*>,	
		$<i_{19}$, *LOC, Berlin*>	} >

In examples we also use the schema presented in Fig.1. Objects are represented by ovals, labelled by objects' external names. Relationships are represented by pointer objects stored inside higher level objects, as in ODMG-93, or by value objects (attributes), as in the relational model. This creates redundancy allowing us to demonstrate different styles of querying, e.g., relational and navigational. The relationship expressed by *EMPLOYS* pointers is opposite to the one expressed by *WORKS_IN* pointers. The atribute *MGR* stores the same string as the *ENO* attribute identified by the content of the pointer *DMGR*. Similarly, the attribute

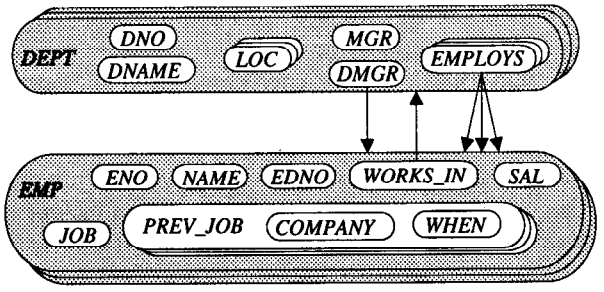

Figure 1: A database schema used in examples

EDNO contains the same value as the attribute *DNO* of the object identified by the pointer *WORKS_IN*. Note that a *DEPT* object contains many pointer sub-objects *EMPLOYS*, and *LOC* and *PREV_JOB* are multi-valued attributes; the latter is a complex one.

3 The Environment Stack

The evaluation of names occuring in queries means *binding* them to run-time data or program units. As usual, we assume that locality of binding scopes is controlled via the environment stack (*ES*). Bulk and persistent data, and some features of queries require changing the construction of the stack used in classical PLs. Our environment stack consists of sections, where each section consists of pairs $n(i)$ and $n(v)$, where $n \in N, i \in I, v \in V$; such pairs are called *binders*. Note that in the binder $n(i)$ name n may be different from the external name of the object identified by i. As will be shown, this feature also allows us to explain parameter passing methods and renaming data objects in views. The same binder may be present in several stack sections and the same object may be accessed through different names (possibly from the same section). We allow many binders in one section to contain the same name; this corresponds to the fact that the external names of objects in a store are not unique. As a consequence, binding a name occuring in a program can be multi-valued.

We assume that at the beginning of the evaluation of a query/program the environment stack consists of one section containing binders of all root database objects determined by R, such as the *EMP* and *DEPT* objects in the *TinyDatabase*.

Binding a name n is performed by a search in the environment stack for one or more binders $n(\vartheta)$. The search follows *scoping rules*. The simplest rule is to follow strictly 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 such binders. In that case, the bag of **all** such elements is taken as a temporary result of the search. The final binding result is received by removing name n

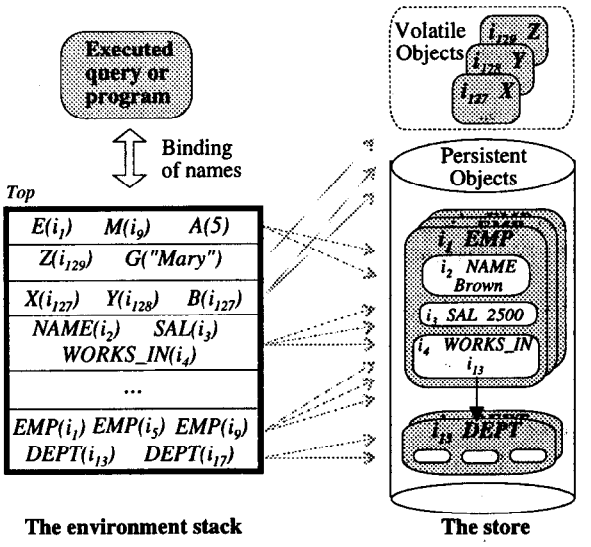

Figure 2: A snapshot of *ES* and of the store

from each binder that belongs to the temporary result: only elements ϑ are left. For example (c.f. Fig.2), if the stack contains the section { $EMP(i_1)$, $EMP(i_5)$, $EMP(i_9)$, $DEPT(i_{13})$, $DEPT(i_{17})$ } and we want to bind *EMP*, then the result of the binding is $\{i_1, i_5, i_9\}$. Similarly, the result of binding *SAL* is $\{i_3\}$. If no section with binders for a given name exists, the empty bag is returned. This simple binding strategy is modified for procedures to accommodate skipping irrelevant stack sections.

Changes to the environment have to update the stack by adding or removing sections at its top, using the standard operations *push* and *pop*.

In PLs, adding a new section (so-called, an *activation record*) corresponds to an activation of a block or a procedure. In a QL, as will be shown, it corresponds also to the evaluation of a query component in a context determined by another component. To formalize opening a new scope in the context of query operators we introduce a function *nested*. Given an identifier i of a complex object, *nested*(i) returns binders to objects at the lower data hierarchy level than the object identified by i. The function has store S as an implicit argument. We generalize the function for elements $v \in V$, binders $n(\vartheta)$, and sequences of identifiers, values and binders. The definitions follow.

- ▷ Let $<i, n, \{<i_1, n_1, \vartheta_1>, ..., <i_k, n_k, \vartheta_k>\}>$ be a complex object with identifier i. In this case
  $nested(i) = \{ n_1(i_1), ..., n_k(i_k) \}$;
- ▷ For a binder the function returns a one-element set with the binder: $nested(\ n(\vartheta)\ ) = \{\ n(\vartheta)\ \}$;
- ▷ If S contains a pointer object $<i_1, n, i_2>$ and an arbitrary object $<i_2, m, \vartheta>$ then
  $nested(i_1) = \{m(i_2)\}$;

▷ For a value $v \in V$ and for a value object $<i, n, v>$ the function returns the empty set:
$nested(v) = nested(i) = \emptyset$;

▷ For a sequence of identifiers, binders and values the function returns the union of the function results for components.

Below we present example applications of the function (c.f. *TinyDatabase*):

$nested(i_1) = \{NAME(i_2), SAL(i_3), WORKS\_IN(i_4)\}$;
$nested(E(i_1)) = \{E(i_1)\}$;
$nested(i_4) = \{DEPT(i_{13})\}$;
$nested(i_3) = nested(1800) = \emptyset$;
$nested(<1800, i_1, i_{13}, A(5)>) =$
$\{NAME(i_2), SAL(i_3), WORKS\_IN(i_4),$
$DNAME(i_{14}), LOC(i_{15}), LOC(i_{16}), A(5)\}$.

Consider the simple query *EMP.SAL*. If the binding of the name *EMP* returns (among others) an identifier i_1, then the scope in which it makes sense to bind the name *SAL* is $nested(i_1)$. If this set is pushed as a new scope onto the stack then the search for bindings for *SAL* will find the object representing the salary of the given employee, as required. This approach is a core of the definiton of query operators, including selection, projection/navigation, join, and quantifiers.

In PLs bindings are mostly performed during compilation; this technique is referred to as *static binding*. Static binding is desirable because of performance, but it requires some program features (types, declarations, variable names, etc.) to be second-class citizens in a PL (i.e., they cannot be manipulated during run-time). In database PLs some bindings must be dynamic because of the dynamic nature of database operations. For uniformity, in this paper we assume that all program features are first-class and all bindings are dynamic. This assumption does not exclude static binding if some program features are supposed to be second-class.

4 The Language SBQL

We sketch the definition of an untyped language called SBQL (Stack-Based Query Language); more comprehensive description can be found in [SBMS93]. SBQL described in this section is a purely retrieval language. The syntax of SBQL is very simple. Any literal or name is an atomic query. From simpler queries we form compound ones using unary or binary operators and parentheses. For example, 2, *EMP*, *SAL* and 1800 are atomic queries, from which we can build queries $2+2$ and *EMP where* ($SAL > 1800$). Note a very untypical property: in the latter compound query *SAL* as well as ($SAL > 1800$) are sub-queries in their own semantic rights, because they are evaluated *relatively* to the state of *ES*. This makes our approach very different from other approaches (for example, from SQL), based on big syntactic and semantic constructs. We argue that this relativity and compositionality much increases the level of orthogonality, minimality and universality. With the exception of typing constraints (e.g., one cannot multiply two strings), we assume full orthogonality of operators. Sometimes we omit parentheses assuming some obvious precedence rules.

4.1 Assumptions Concerning Semantics

Intermediate and final query results are kept at a stack, called the *query result stack* (*QRES*). The stack is a generalization of the well-known arithmetic stack necessary to perform parenthesized arithmetic expressions[3]. In this paper we assume that an element of *QRES* is a *table*, where a table is a bag of *rows*, all of the same width; sometimes we assume that a table is a sequence of rows. Rows may contain atomic values, identifiers and binders. For uniformity, we represent single values or identifiers as tables having one row and one column (1×1 tables), and do not make distinction between such a table and an element inside it (thus we apply to some 1×1 tables operators such as $+, *, <, \wedge$, etc.). An example table, referring to the *TinyDatabase*, is presented below; it may represent the result of the query "Get identifiers of employee names, identifiers of their department names giving them auxiliary name N, and 10% of their salary".

i_2	$N(i_{14})$	250
i_6	$N(i_{18})$	200
i_{10}	$N(i_{18})$	150

When formulating queries in English, we will usually omit the phrase "...identifier(s) of...", assuming that queries return identifiers of the mentioned objects rather than values being their content.

To define the operational 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 an environment stack. The procedure is defined by cases, one for each operator. It may change *ES*; 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*.

For the query l, where l is a literal denoting value $\bar{l} \in V$, $eval(l)$ pushes the 1×1 table $\{<\bar{l}>\}$ onto the top of *QRES*. For the query n, where n is a name, $eval(n)$ inspects *ES*, and pushes the result of the binding onto the top of *QRES* as a single-column table of identifiers and values. The results of such simple queries (as well

[3] In the denotational setting *QRES* can be easily hidden in recursive semantic clauses. This observation *does not* concern *ES*: any formal approach must deal with it explicitly.

as the results of complex queries) can be combined by application of operators, which we subdivide into *algebraic* and *non-algebraic*.

4.2 Algebraic Operators

An operator is called *algebraic* if its semantics is defined purely in terms of operations on *QRES* and does not involve *ES*. That is, if Δ is a symbol denoting a binary algebraic operator $\overline{\Delta}$ and $res(q)$ is the result of evaluation of q, then $res(q_1 \Delta q_2) = res(q_1)\ \overline{\Delta}\ res(q_2)$. A corresponding part of the *eval* procedure may look as follows:

```
procedure eval(query)
begin
    .....
case query is q1Δq2: (* Δ is algebraic *)
begin
  q1result, q2result: table;
    (* Push the results of q1, q2 onto QRES *)
  eval(q1); eval(q2);
    (* Apply Δ̄ to both results *)
  q2result := top(QRES); pop(QRES);
  q1result := top(QRES); pop(QRES);
  push( QRES, q1result Δ̄ q2result );
end
    .....
end eval;
```

Both component queries are evaluated in the same environment (assuming no side effects of functional procedures called in q_1), and the result of *q1result* $\overline{\Delta}$ *q2result* is defined without any use of *ES*. Semantics of unary, ternary, etc. operators can be defined similarly.

Typical classes of the algebraic operators include: numerical comparisons, operators and functions: $<, \leq, =, \neq, >, \geq$, $+, -, \star, /, \star\star$, *sin, log, sqrt*, etc.; string comparisons, operators and functions; boolean *and, or, not* (denoted here $\wedge, \vee$ and $\neg$, respectively); comparison of identifiers for equality/non-equality; coercion operators, e.g. changing a string into an integer; dereferencing operator (mostly implicit); aggregate arithmetic functions *sum, min, max, avg*, etc.; a function removing duplicate rows (c.f. SQL); function *exists* mapping a non-empty table to *TRUE* and empty one to *FALSE*; function *count* mapping a table into the number of rows; set, bag and sequences operators and comparisons: cartesian product (denoted here by $\times$), union (denoted here by $\cup$), intersection, difference, is-equal-set, $\subseteq$, $\in$, etc. In this paper we do not deal with (usually obvious) definitions of these operators nor with their pragmatic meaning in QLs.

An important (unary) algebraic operator is the one introducing an auxiliary name. The syntax is $n \leftarrow q$, where $n \in N$ is an auxiliary name, and q is a query returning a single-column table containing values or identifiers. Let q return the table

$\{ <\vartheta_1>, <\vartheta_2>, ..., <\vartheta_k> \}$, where $\vartheta_i \in V \cup I$.

Then $n \leftarrow q$ returns the table of binders

$\{ <n(\vartheta_1)>, <n(\vartheta_2)>, ..., <n(\vartheta_k)> \}$.

Together with our scoping and binding rules this simple operator constitutes a surprisingly powerful facility, which covers many QLs' features, such as tuple and domain calculus variables, variables bound by quantifiers, cursors in *for each* statements, and renaming data objects in views.

4.3 Non-Algebraic Operators

Non-algebraic operators are *where*, dot, join, quantifiers, ordering, etc. We emphasize that the operators we describe are more general than the standard ones. Their semantics cannot be expressed in the spirit of the relational algebra. The syntax for application of a non-algebraic operator is $q_1 \Theta q_2$, where q_1 and q_2 are (arbitrarily complex) queries, and Θ is an operator (the syntax for quantifiers has a prefix form). The general pattern that defines a part of the *eval* procedure is as follows.

```
procedure eval(query)
    .....
case query is q1Θq2: (* Θ is non-algebraic *)
begin
  partial_results: array of table;
  final_result: table; i: integer; i := 1;
  eval(q1); (* Push the result of q1 onto QRES *)
  for each row r in top(QRES) do
    push(ES, nested(r)); (* Open a new scope *)
    eval(q2); (* Push the result of q2 onto QRES *)
    partial_results[i] := combine_Θ(r,top(QRES));
    pop(ES); pop(QRES); i := i + 1;
  end for each;
  final_result := merge_Θ(partial_results);
  pop(QRES); (* Remove the table created by q1 *)
  push(QRES, final_result);
end
    .....
```

For each row r of the table returned by q_1 the procedure opens a new scope *nested*(r) on *ES*, then evaluates q_2. A partial result returned for this r is a combination of r and of the result returned by q_2. Finally, partial results are merged into the final result, which is pushed onto *QRES*. (This pattern should be a bit modified for ordering and transitive closure).

According to the above pattern we can define classical and new operators that may be useful in QLs, in particular, the following:

Selection: the syntax is q_1 *where* q_2, where q_1 is

any query, and q_2 is a boolean-valued query. If for a particular r returned by q_1 the query q_2 returns *TRUE* then r is an element of the final result table; otherwise it is skipped.

Projection, navigation, path expressions: the syntax is $q_1.q_2$. The final result table is the union of tables returned by q_2 for every row r returned by q_1. Path expressions are recently extensively discussed in the literature; we emphasize that in our approach path expressions are a "side effect" obtained by nesting queries containing the binary dot operator, e.g., $q_1.q_2.q_3.q_4$ is understood as $((q_1.q_2).q_3).q_4$. Note that after a dot we allow not only a name (as in typical path expressions), but an arbitrary query. As will be shown in examples, such an assumption is practical and general.

Navigational join: the syntax is $q_1 \bowtie q_2$. A partial result for a particular r returned by q_1 is a table obtained by a concatenation of the row r with each row returned by q_2 for this r. The final result is the union of partial results.

Quantifiers: the syntax is $\forall q_1(q_2)$ and $\exists q_1(q_2)$, where q_2 is a boolean-valued query. For $\forall$ the final result is *FALSE* iff q_2 returns *FALSE* for at least one row r returned by q_1; otherwise the final result is *TRUE*. For $\exists$ we apply a dual definition.

Ordering: the syntax is q_1 *order_by* q_2. First, $q_1 \bowtie q_2$ is calculated; then the result is coerced to a sequence of rows and sorted according to the columns returned by q_2. Then, these columns are removed from the final result.

Transitive closure: the syntax is q_1 *closed_by* q_2. First q_1 is evaluated and the result is stored at a dynamic table T. Then, for each row r in the table T the query q_2 is evaluated, and the result of the evaluation augments the table T. In this way, rows returned by q_2 will be again processed by q_2. The result can be represented as the least fixed-point of the equation $T = q_1 \cup T.q_2$ which, if exists, is $T = q_1 \cup q_1.q_2 \cup q_1.q_2.q_2 \cup q_1.q_2.q_2.q_2 \cup \ldots$. For space limit we do not present examples which use this operator; see [SMA90, SBMS93].

4.4 Examples of Queries

We can show in examples (see [SMA90, SBMS93, SBMS94]) that the retrieval power of SBQL is considerably higher than the retrieval power of all QLs that we are aware of. Below we present some queries in SBQL (C.f. Fig.1); all are operational in LOQIS.

1. Return an identifier of the *WORKS_IN* pointer within the Smith's object:

(EMP where NAME = "Smith").WORKS_IN

2. Name of the Smith's department:

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

3. Employees of the Toys department earning more than 1800:

EMP where SAL $>$ 1800 $\wedge$
(WORKS_IN.DEPT.DNAME) = "Toys"

4. Employees earning more than Smith:

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

5. Join *EMP* and *DEPT*, the relational style:

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

or

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

6. As above, the navigational style:

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

7. For each employee, return name and location(s) of her/his department:

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

8. For each department return the average salary of its employees (see Fig.3 and the discussion below):

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

9. For each *DEPT* having more than 50 programmers return name and the dispersion of salaries, the navigational style (to receive the relational style change *EMPLOYS.EMP* into *(EMP where EDNO = DNO)*):

((DEPT where count(EMPLOYS.EMP
where JOB = "programmer") $>$ 50)
$\bowtie$ *(a* $\leftarrow$ *avg(EMPLOYS.EMP.SAL))).*
(DNAME $\times$ *sqrt(sum(EMPLOYS.EMP.*
*((SAL − a) * (SAL − a)))/*
(count(EMPLOYS.EMP) − 1)))

10. Departments where all programmers used to work for IBM:

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

11. (Integrity constraint) No department has an employee earning more than his manager.

$\forall x \leftarrow$*DEPT* $(\neg \exists\ y \leftarrow (x.EMPLOYS.EMP)$
$(x.DMGR.EMP.SAL < y.SAL))$

12. Names of clerks earning more than their managers, the SQL style:

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

13. As above, the domain calculus style:

$(EMP.((xn \leftarrow NAME) \times (xs \leftarrow SAL) \times$
$(xj \leftarrow JOB) \times (xd \leftarrow EDNO))) \times$
$(EMP.((ys \leftarrow SAL) \times (ye \leftarrow ENO))) \times$
$(DEPT.((zd \leftarrow DNO) \times (zm \leftarrow MGR)))$
$where\ (xj = "clerk" \wedge xs > ys \wedge$
$xd = zd \wedge zm = ye)).\ xn$

14. As above, the navigational style:
$(EMP\ where\ JOB = "clerk" \wedge SAL >$
$(WORKS\_IN.DEPT.DMGR.EMP.SAL)).NAME$

15. Departments ordered by the number of employees, in descending order:
$DEPT\ order\_by\ (100000 - count(EMPLOYS))$

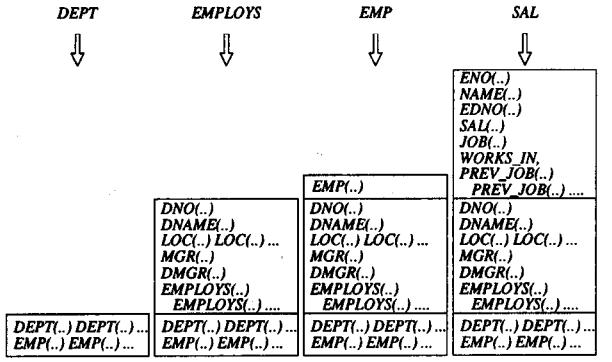

Figure 3: States of ES during binding names in $DEPT \bowtie avg((EMPLOYS.EMP).SAL)$

We present some of the intuitions concerning semantics on the query from example 8, Fig.3. At the beginning ES contains one section with $DEPT$ and EMP binders. (In LOQIS we implemented simple optimizations to avoid storing and processing long lists of binders.) Name $DEPT$ returns all $DEPT$ identifiers. The operator $\bowtie$ scans them, in each loop pushing onto ES binders to all sub-objects of some $DEPT$ object, in particular, $EMPLOYS$ binders. Name $EMPLOYS$ returns their identifiers. The dot after $EMPLOYS$ scans them, pushing onto ES in each loop a section with a single binder EMP; thus name EMP returns a single identifier of EMP. These identifiers are merged into a table. The last dot scans it, in each loop pushing onto ES binders to sub-objects of some EMP object, in particular a SAL binder. Name SAL returns a single identifier to a value object SAL. These identifiers are merged into a table; it consists of identifiers of SAL objects of all employees working in a particular department. Function avg (after automatic dereferencing) counts the average value of their content. According to the semantics of $\bowtie$, the final result is a two-column table, where the first column contains identifiers of all departments, and the second one contains the average salaries in these departments.

5 Imperative Statements

We introduce several constructs which make possible to change a state. The statements are build from queries. Presented definitions have illustrative purposes only. Besides, we use some self-explanatory functions and standard control statements.

We use here statements of the form $q.p$, where q is a query, and p is a a sequence of imperative statements. The construct is a natural overloading of the dot operator. The program p is executed for each row r returned by q with a new scope $nested(r)$ pushed onto ES. (The construct is frequently expressed as *for each q do p.*)

5.1 Creating and Deleting Objects

We assume for uniformity that both database objects and local procedure objects are dynamically created and deleted. After creating a new root persistent object (at the top object hierarchy level) its binder is inserted into a bottom stack section. Local procedure objects are created in a store outside the stack, and for root objects their binders are inserted in an ES section that is created for this procedure invocation. They are automatically removed when the procedure is terminated and the stack is popped. Any object can be removed by an explicit *delete* command, which for root objects has also the effect of removing the corresponding binder from an appropriate stack section.

The syntax for creating and deleting objects is *create persistent <object description>*, *create local <object description>*, and *delete q*, where q is a query returning a single-column table of identifiers; all objects with these identifiers are deleted. The *<object description>* is recursively defined as: $n(q_1)$, $n(\uparrow q_2)$ and $n(<object\ description>,...)$, where $n \in N$, q_1 returns a single-column table of values (dereferencing is automatically applied) and q_2 returns a single-column table of identifiers. In the first case, if q_1 returns a table $\{<v_1>,<v_2>,...,<v_k>\}$ then k new value objects $<i_{new_j}, n, v_j>$ are created. In the second case, if q_2 returns a table $\{<i_1>,<i_2>,...,<i_k>\}$ then k new pointer objects $<i_{new_j}, n, i_j>$ are created. In the third case a new complex object with name n is created, together with its sub-objects, all with new identifiers. In LOQIS we also implemented a case when q_1 returns identifiers of complex objects.

create persistent EMP(
NAME("Clark") SAL(avg(EMP.SAL))
WORKS_IN(↑DEPT where DNAME="Toys"));

Delete the London location of the department Toys:
delete (DEPT where DNAME = "Toys").
((x ← LOC) where x = "London").x;

5.2 Assignments (Updates)

For assignments we use the typical syntax $q_1 := q_2$, where query q_1 has to return an identifier (l-value) and query q_2 has to return a value (r-value). The value is assigned as a new value of the object identified by the identifier. As usual in PLs, r-value can be a pointer. We distinguish syntactically the assignment of a content of an object and the assignment of a pointer by using the syntax $q_1 := \uparrow q_2$ for the latter case. As for *create*, in LOQIS we implemented an assignment of a complex object, with applied recursively copy semantics.

In combination with the construct $q.p$ these assignments make possible updates similarly to SQL.

(EMP where JOB = "clerk").
(SAL := SAL+100; JOB := "officer");

(EMP where SAL > 1800). (WORKS_IN :=
↑ (DEPT where DNAME = "Toys"));

Let each employee earn at least the actual average salary in her/his department:

DEPT.((a ← avg(EMPLOYS.EMP.SAL)).
((EMPLOYS.EMP where SAL<a).(SAL := a)));

5.3 Insertions

For insertions we use the syntax $q_1 \lhd= q_2$, where q_1 returns an identifier of a complex object, and q_2 returns a single-column table of identifiers of arbitrary objects. The statement causes that the objects identified by q_2 become a sub-objects of the first one; the move is without copying and without changing the identifiers of the objects being moved. In some cases a binder of the second object should be removed from an *ES* section, as in the following fragment:

create local LOC("Tokyo");
(DEPT where DNAME = "Toys") ⊲= LOC;

After insertion the second object may change its persistence status, depending on the status of the first object.

6 Object Orientation

We introduce a simple extension of the model which will allow us to define all features of query languages necessary to deal with object-oriented databases. In the definitions we follow Smalltalk in the way classes are used, and Modula-2 as far as encapsulation is concerned. The proposed extensions to the object store presented in Section 2 include the following features.

6.1 Encapsulation

Similarly to modern polymorphic PLs we assume that an object may contain not only passive sub-objects, but also procedures, functional procedures, rules, constraints, etc. As in Modula-2, we assume that any kind of sub-objects can be exported outside the object, as *NAME*, *WORKS_IN*, *Age*, *ChangeSal* and *SalNet* in Fig.4, or can be private, as *BIRTH_DATE*, *SAL* and *Tax* in Fig.4. The idea could be called as *orthogonality of encapsulation to a kind of objects.* Encapsulation in the above sense means that an object is associated with an export list which specifies its private and public sub-objects.

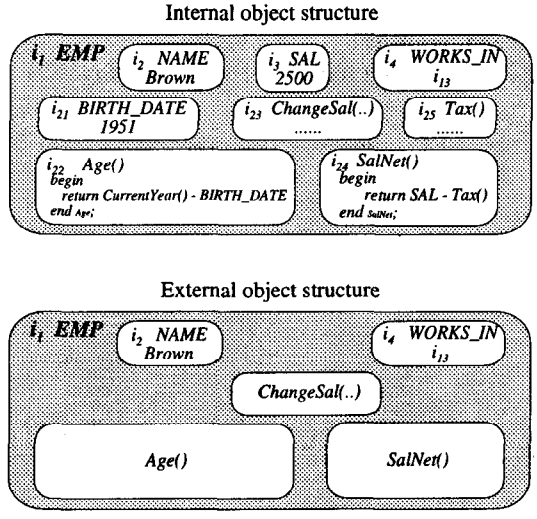

Figure 4: Internal and external object structure

Let o be an object containing a procedure (method) p as a sub-object. A name occuring inside the body of the procedure can be bound to any direct sub-object of o, including private sub-objects. For example, in Fig.4 inside the procedure *Age* the private sub-object *BIRTH_DATE* can be bound, thus the query

CurrentYear() - BIRTH_DATE

is valid; however, it is invalid outside the object, because *BIRTH_DATE* is not exported. Thus encapsulation can be handled by scoping rules. According to them the following query and statement are correct:

(EMP where Age() > 40).(NAME × SalNet()
× (WORKS_IN.DEPT.DNAME))

(EMP where NAME="Brown").ChangeSal(3500);

6.2 Classes and Inheritance

Fig.4 presents procedures (methods) such as *Age* and *ChangeSal* within an *EMP* object. Usually, the same procedures should be stored within all *EMP* objects. We may avoid the redundancy by introducing an additional "master" object, storing all common components for a collection of (similar) objects; we assume that each object in the collection treats the components as its own ("imports" them), Fig.5. In other

words, we assume that each object o of the collection has a special link to the master object. Thus if i identifies o then $nested(i)$ is the set of binders of exported sub-objects of o, and of the master object. Such a master object we call a *class*. This idea of the class concept is presented in [SMSRW93] and implemented in LOQIS.

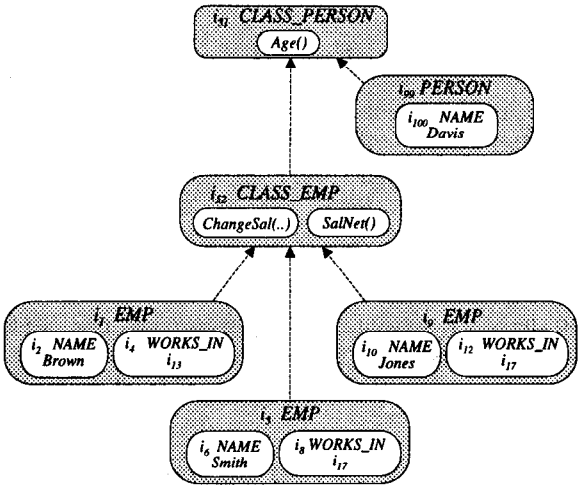

Figure 5: Objects, classes and inheritance

In Fig.5 we show inheritance of invariant properties by dashed arrows. As seen, classes form a hierarchy and inheritance of invariants is transitive: an *EMP* object inherits invariant properties from *CLASS_EMP* and from the super-class *CLASS_PERSON*. Overriding appears naturally as a side effect of the assumed binding and scoping rules. Multi-inheritance can be easily modelled if we assume that each object can have more such links to class objects (name conflicts can be resolved by scoping rules). Besides procedures (i.e. methods), the class may also store object types, export lists for objects, default values, active rules, integrity constraints, etc.[4]; that is, all properties that are *invariant* for the class members.

A class may also store some features common to the whole class, for example, a method for creating new objects. Such common features can also be exported outside the class, but they should be distinguished from features which can be inherited by class members. As shown in Fig.5, classes are objects too and can be queried and processed by standard query/programming functionalities. For example, assuming *NewEmp* is an (exported) procedure stored inside the class *CLASS_EMP*, the construct *CLASS_EMP.NewEmp*() is a correct statement. Access and encapsulation rules, and other dependencies, can be easily introduced into the function *nested* and scoping/binding rules. The method can also be used to implement *independent object roles*, [SMSRW93].

[4]Note that if class objects inherit from a class only procedures (methods, operators), and all own properties of the class members are private, then the class is *abstract data type*.

7 Procedures and Views

Procedures that we deal with can call arbitrary procedures, can be recursive, may have parameters being queries, and encapsulate their local objects so no binding to them can be performed from outside. The local objects of a procedure are associated with a procedure invocation rather than with the procedure code. These assumptions lead to an environment stack concept, and we use for this new role our stack *ES*. An invocation of a procedure means opening a new section on the environment stack for local data objects. In our case we push onto *ES* a section of binders of local objects, which will be stored elsewhere. Scoping rules should provide skipping irrelevant stack sections: if p_1 calls p_2 then after the call the local environment of p_1 should be unvisible. A functional procedure before termination pushes its output onto the *QRES* stack. An invocation of a functional procedure is considered a query and may be used in all contexts in which queries can be used.

We use a typical syntax, with a header containing the formal parameter list. The syntax of functional procedures (views) is the same; the only difference concerns *return* statements.

(C.f. Fig.1) A functional procedure *Poor* has one column table of strings as a parameter; the table denotes a list of jobs. It returns identifiers of names, salaries, and department names of employees (giving them auxiliary names *N, S, D*, respectively), who do one of the specified jobs and earn less than the average. The procedure may be considered as an updatable, parameterized, SQL-like view $Poor(N, S, D)$.

procedure *Poor*(*Jobs*)
begin
  create local AVERAGE(*avg*(*EMP.SAL*));
  create local POOR(↑ *EMP where*
    (*JOB* ∈ *Jobs* ∧ *SAL* < *AVERAGE*));
  return POOR.EMP.(
    (*N* ← *NAME*) × (*S* ← *SAL*) ×
    (*D* ← (*WORKS_IN.DEPT.DNAME*)))
end *Poor*;

First, a local value object *AVERAGE* is created, with a value of the average salary. Then, zero, one or more local pointer objects *POOR* are created, depending on the number of employees doing the specified jobs and earning less than the average. The last statement returns as the procedure output a table having rows $<N(i_1), S(i_2), D(i_3)>$ (containing binders), where i_1, i_2, i_3 are identifiers of *NAME*, *SAL* and

DNAME atributes (respectively) for each of the poor employees; *N*, *S*, *D* are virtual names for these attributes.

Increase salaries of poor clerks and tailors from the department "*Toys*" by 100:

(*Poor*("*clerk*" ∪ "*tailor*") *where* D = "*Toys*").
($S := S+100$);

(C.f. Fig.1) A procedure '*ChangeDept*' has a parameter E which is a table of identifiers of *EMP* objects, and a parameter D which is an identifier of a department (both called by reference). It causes moving the specified employees to the specified department.

procedure *ChangeDept*(**var** E; **var** D)
begin
delete (*DEPT.EMPLOYS*) *where* $EMP \in E$;
($e \leftarrow E$).(
create local EMPLOYS($\uparrow e$);
$D \lhd= EMPLOYS$; $e.WORKS\_IN := \uparrow D$;
$e.EDNO := D.DNO$;)
end *ChangeDept*;

Let Kim become the manager of all designers working so far for Lee:

ChangeDept(*EMP where JOB* = "*designer*" ∧
WORKS_IN.DEPT.DMGR.EMP.NAME = "*Lee*";
DEPT where DMGR.EMP.NAME = "*Kim*");

Define a method (a virtual attribute) *Boss* for the *EMP* class, to navigate from an employee directly to her/his manager. Note that the navigation starts from *WORKS_IN*, as this procedure is executed after opening a local scope for an *EMP* object.

procedure *Boss*()
begin
return WORKS_IN.DEPT.DMGR.EMP
end *Boss*;

Get employees earning more that their managers:

EMP where SAL > *Boss*().*SAL*

We present two classical parameter passing methods, *call-by-value* and *call-by-reference*, modified in order to make query processing possible. Besides, we developed and implemented other methods (*strict call-by-value* and *call-by-union*); they will be described in forthcoming papers.

7.1 Call-By-Value

The syntax

procedure *p*(..; *Fpar*; ..) **begin** .. **end** *p*;

means that *Fpar* is a name a formal parameter for which we designate the *call-by-value* method. An invocation of the procedure has the form *p*(..; *q*; ..) where *q* is a query which has to return a single-column table of values $\{<v_1>,<v_2>,...,<v_k>\}$; automatic dereferencing is assumed. Then we have two possibilities (semantically not equivalent):

(1) Create k new value objects
$<i_{p1}, Fpar, v_1>,<i_{p2}, Fpar, v_2>,...,<i_{pk}, Fpar, v_k>$
in the volatile pool; all identifiers $i_{p1}, ..., i_{pk}$ are distinct. All the objects have the same name *Fpar*. Then, create k binders
$Fpar(i_{p1})$, $Fpar(i_{p2})$, ..., $Fpar(i_{pk})$
and insert them into the stack section created for this procedure invocation. The method does the same as the statement *create local Fpar*(*q*) executed after the procedure invocation (providing *q* is evaluated in the environment of a caller).

(2) Insert binders
$Fpar(v_1)$, $Fpar(v_2)$, ..., $Fpar(v_k)$
into the stack section. In this case we can refer to the parameters inside the procedure body, but we cannot update them (i.e., the parameters are local constants).

The method is illustrated by the procedure *Poor*. In the first case the invocation *Poor*("*clerk*"∪"*tailor*") means the creation of local objects
$<i_{p1}, Jobs, $"*clerk*"$>$, $<i_{p2}, Jobs, $"*tailor*"$>$
and the insertion of two binders
$Jobs(i_{p1}), Jobs(i_{p2})$
into the stack section created for this procedure invocation. In this way name *Jobs* inside the procedure body can be bound, as usual, to identifiers i_{p1}, i_{p2}. In the second case no local objects are created but binders
Jobs("*clerk*"), *Jobs*("*tailor*")
thus binding name *Jobs* returns an one-column table
$\{<$"*clerk*"$>, <$"*tailor*"$>\}$.

We can also deal with the case when a value of a parameter is an identifier of a complex object, assuming *copy semantics* or *reference semantics*, or introducing some concept of a complex value.

7.2 Call-By-Reference

The syntax

procedure *p*(..; **var** *Fpar*; ..) **begin** .. **end** *p*;

means that *Fpar* is a name of a formal parameter for which we designate the *call-by-reference* method. An invocation of the procedure has the form *p*(...; *q*; ...) where *q* is a query which has to return a single-column table of identifiers $\{<i_1>, <i_2>, ... <i_k>\}$. After the invocation the binders $Fpar(i_1)$, $Fpar(i_2)$, ..., $Fpar(i_k)$ are inserted into the stack section created for this invocation. This method we used in the procedure *ChangeDept* above; in this case the stack section contains zero, one or more binders $E(i_{EMP})$ and one binder $D(i_{DEPT})$.

8 Optimization by Rewriting

If a functional procedure does not introduce local objects, its stack-based semantics is equivalent to macro-substitution semantics, what makes possible optimizations based on rewriting. We present it by example.

Let the view *DptMgrAv*() return a table containing triples $<d, m, a>$ with identifiers of departments, identifiers of their managers, and the average salary of their employees, respectively:

procedure *DptMgrAv*()
begin
return $(d \leftarrow DEPT) \bowtie (m \leftarrow (d.DMGR.EMP)) \bowtie$
$(a \leftarrow avg(d.EMPLOYS.EMP.SAL))$
end *DptMgrAv*;

Consider the query "Give name of the manager of the Toys department":
$(DptMgrAv()\ where\ d.DNAME="Toys").m.NAME$

After the textual substitution of the procedure invocation by the procedure body (note our extremely simple "algorithm" of query modification):
$((d \leftarrow DEPT) \bowtie (m \leftarrow d.DMGR.EMP) \bowtie$
$(a \leftarrow avg(d.EMPLOYS.EMP.SAL)))$
$where\ d.DNAME = "Toys").m.NAME$

Since auxilary name a is not used, the part $(a \leftarrow avg(d.EMPLOYS.EMP.SAL))$ can be dropped:
$((d \leftarrow DEPT) \bowtie (m \leftarrow (d.DMGR.EMP)))$
$where\ d.DNAME = "Toys").m.NAME$

After shifting the selection before the join:
$(((d \leftarrow DEPT)\ where\ d.DNAME = "Toys") \bowtie$
$(m \leftarrow (d.DMGR.EMP)))).m.NAME$

After changing the join into navigation (because the final projection does not refer to the first join argument):
$((d \leftarrow DEPT)\ where\ d.DNAME = "Toys").$
$(m \leftarrow (d.DMGR.EMP)).m.NAME$

After reducing name m:
$((d \leftarrow DEPT)\ where\ d.DNAME = "Toys").$
$(d.DMGR.EMP.NAME)$

After reducing name d:
$(DEPT\ where\ DNAME = "Toys").$
$DMGR.EMP.NAME$

As shown, some optimization techniques developed for relational QLs can be adopted and generalized for stack-based QLs, for instance, performing selections and projections before joins. Because of regularity and formality our model makes it possible to develop many formal axioms and theorems concerning such transformation steps. New techniques are also possible: in [SBMS93] we presented a general optimization method based on observations concerning bindings.

9 Conclusion

We have presented an approach to integration of queries and procedures based on a modification of classical PL concepts. Being a kind of theory, the approach decisively departs from the traditional and some new formal approaches to QLs, such as nested relational algebras, object algebras, relational calculi, predicate logic, F-logic, comprehensions, etc. We argue that these frameworks are too abstract and limited in power, thus *not precise enough* to deal with semantic issues which occur in real-life database query/programming languages such as OQL of ODMG-93 and SQL-3. In comparison, our approach supports consistent and universal semantics, follows modern software engineering principles, and avoids artificial problems, for instance, how to define updating. The approach makes it possible to define very powerful QLs, integrated with macroscopic imperative statements, procedures, views, and object-oriented notions, for a variety of data models. It also contributes a lot to the understanding of the true nature of semantic decisions in the languages that are actually used, implemented or designed for these models. Because of regularity, orthogonality and universality of the concepts it should be well perceived by programmers and is promising for query/program optimization.

References

[ANSI94] American National Standards Institute (ANSI) Database Committee (X3H2). Database Language SQL3. J.Melton, Editor. August 1994.

[Catt94] R.G.G. Cattel (Ed.) The Object Database Standard ODMG-93. Morgan Kaufman 1994.

[SMA90] K. Subieta, M. Missala, and K. Anacki. The LOQIS System. Institute of Computer Science Polish Acad. Sci., Report 695, Warszawa, Nov. 1990.

[SMSRW93] K. Subieta, F. Matthes, J.W. Schmidt, A. Rudloff, I. Wetzel. Viewers: A Data-World Analogue of Procedure Calls. Proc. 19th VLDB Conf., Dublin, Ireland, pp.269-277, 1993.

[SBMS93] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt. A Stack-Based Approach to Query Languages. Institute of Computer Science Polish Acad. Sci., Report 738, Warszawa, Dec. 1993. http://banjo.imel1.kuis.kyoto-u.ac.jp/~subieta

[SBMS94] K. Subieta, C. Beeri, F. Matthes, J.W. Schmidt. A Stack-Based Approach to Query Languages. Proc. of 2nd Intl. East-West Database Workshop, Klagenfurt, Austria, September 1994, Springer Workshops in Computing, 1995.