Stack-Based Architecture and Stack-Based Query Language

Radosław Adamus[1,2], Piotr Habela[1], Krzysztof Kaczmarski[1,3], Michał Letner[1],
Krzysztof Stencel[1,4], Kazimierz Subieta[1,2,5]

[1] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[2] Computer Engineering Department, Technical University, Łódź, Poland
[3] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
[4] Institute of Informatics, Warsaw University, Warsaw, Poland
[5] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
radamus@kis.p.lodz.pl. {habela, m.lentner, stencel, subieta}@pjwstk.edu.pl, k.kaczmarski@mini.pw.edu.pl

Abstract. Stack-Based Query Language (SBQL) is a model query and programming language for a wide family of object-oriented database models. SBQL is the result of investigations into a uniform theoretical and conceptual basis for object-oriented query languages integrated with programming capabilities and abstractions, including database abstractions: updatable views, stored procedures and transactions. SBQL is developed according to the stack-based architecture that is a conceptual frame for developing object-oriented query and programming languages. SBQL deals with object store models that include complex objects, associations, classes, types, methods, inheritance, roles, encapsulation, polymorphism, semi-structured data and other features. The paper presents fundamental ideas and topics of SBA and SBQL.

1. Introduction

The Stack-Based Architecture (SBA) is a formal frame addressing object-oriented database query and programming languages (PLs). The approach is motivated by the belief that there is no definite border line between querying and programming; thus there should be a universal theory that uniformly covers both aspects. SBA offers a unified and universal conceptual and semantic basis for queries and programs involving queries, including programming abstractions such as procedures, functions, classes, types, methods, views, etc.

SBA assumes a semantic specification method that is referred to as *abstract implementation*. It is an operational semantics where one has to determine precisely an abstract query (and program) execution machine. It involves all the data structures that participate in query/program processing and then, specifies the semantics of all the languages' operators in terms of actions on these structures. SBA introduces three such structures that are well-known in the specification of PLs: (1) an object store, (2) an environment stack (ENVS), (3) a query result stack (QRES); thus the *stack-based* architecture. Query operators, such as selection, projection and join, can be precisely specified using the above three structures.

SBA introduces a model query and programming language SBQL (Stack-Based Query Language). SBQL plays the same role as relational algebra for the relational model, but SBQL is incomparably more powerful. The power of SBQL concerns a wide spectrum of data structures that it is able to serve and complete algorithmic power of querying and manipulation capabilities. At the same time, SBQL is precise with respect to the specification of semantics. The quality of SBQL is achieved by orthogonality of introduced data/object constructors, orthogonality of all the language constructs, object relativism, orthogonal persistence, typing safety, introducing all the classical and some new programming abstractions (procedures, functions, modules, types, classes, methods, views, etc.) and following commonly accepted programming languages' principles.

Functionality of SBQL includes all well-known query operators (selection, projection, navigation, path expressions, join, quantifiers, etc.), some less known operators (transitive closures, fixed-point equations, etc.), imperative (updating) statements integrated with queries, modules, procedures, functions and methods (with parameters being queries and recursive). SBQL deals with static strong type checking and with query optimization methods based on indices, rewriting rules and other techniques. The idea of SBA and SBQL has been thoroughly validated , since SBQL has been implemented within several research prototypes; the last and the most complete one is ODRA (Object Database for Rapid Applications) [4].

This paper is organized as follows. In Section 2 we advocate the uniformity of query and programming languages. In Section 3 we present universal store models used in SBA/SBQL. In Section 4 we present the syntax of SBQL queries. In Section 5 we focus on one of main concepts in SBA, i.e. the environment stack. In Section 6 we shortly describe some more advanced notions which can be built atop the architectural elements presented in Sections 3-5. Section 7 contains simple examples of SBQL queries. Section 8 concludes.

2. Query Languages as Programming Languages

SBA is an alternative to theoretical concepts emerging on the object-oriented wave, such as nested relational algebras, object algebras, object calculi, F-logic, comprehensions, structural recursion, monoid calculus, functional approaches, etc. The SBA solution relies on adopting a classical run-time mechanism of PLs, and then introducing some necessary improvements to it. The main syntactic decision of our approach is the unification of PL expressions and queries; queries remain the only kind of PL expressions. For instance, in SBQL there is no conceptual difference between expressions such as 2+2 and $(x+y)*z$, and queries such as *Employee* **where** *Salary* = 1000 or (*Employee* **where** *Salary* = $(x+y)*z$).*Name*. All such expressions/queries can be used as arguments of imperative statements, as parameters of procedures, functions or methods, as a return from a functional procedure, and so on.

Concerning semantics, SBQL is based on the classical *naming-scoping-binding* paradigm. Each *name* occurring in a query is *bound* to run-time programming entities (persistent data, procedures, actual parameters of procedures, etc.), according to the actual *scope* for the name. The common PLs' approach that is used in SBA is that the scopes are organized in an *environment stack* with the "search from the top" rule. The role of the stack in PLs is fundamental and well explained e.g. in [1]. For the same reasons the stack should be involved in query languages. Some extensions to the structure of stacks used in PLs are necessary to accommodate the fact that a database contains persistent and bulk data structures and the fact that the data is kept on a server machine, while the stack is kept on a client machine. Hence the stack contains references to data rather than data themselves (i.e., the stack is separated from a store of objects), and possibly multiple objects can be simultaneously bound to a name occurring in a query (for processing collections). The semantics of query operators, imperative programming constructs and procedures (functions, methods, views, etc.) is defined in terms of actions on the object store, ENVS and QRES.

3. Object Store Models

Because various object models introduce a lot of incompatible notions, SBA assumes a family of object store models which are enumerated AS0, AS1, AS2 and AS3[1]. The simplest is AS0, which covers relational, nested-relational and XML-oriented databases. AS0 assumes

---

[1] Originally AS0-AS3 were denoted M0-M3, correspondingly.

hierarchical objects with no limitations concerning the nesting of objects and collections. AS0 also covers pointer links (relationships) between objects. In the AS0 model all objects can be represented by tuples (triplets) as follows:

- **Atomic object**: $\langle i,n,v \rangle$ where i is the ID of the object, n is an external name assigned to the object, and v is the value of the atomic object (e.g. an integer, a string, etc.)
- **Pointer (reference) object**: $\langle i_1,n,i_2 \rangle$ where i_1 is the ID of the reference object, n is an external name assigned to the object, and i_2 is the ID of the object referred to.
- **Aggregation object**: $\langle i,n,T \rangle$ where i is the ID of the object, n is an external name assigned to the object, and T is a set of objects comprising the aggregate.

AS0 itself can be represented as $\langle S,R \rangle$ where S is a set of objects and R is the set of "root" object IDs.

AS1 extends AS0 by classes and static inheritance, AS2 extends AS1 by object roles and dynamic inheritance, and AS3 extends AS1 or AS2 by encapsulation.

An example of a complex objects having three sub-objects is presented below:

$$\langle i_5, Emp, \{\langle i_6, name, \text{"Poe"}\rangle, \langle i_7, sal, 2000\rangle, \langle i_8, worksIn, i_{22}\rangle\}\rangle$$

The objects named *Name* and *Sal* are atomic objects, the objects named *worksIn* is a pointer object. Each object has a unique identifier. Unique object identifiers are somehow internally represented, but representation of identifiers is not interesting for the SBA theory. In this paper we will used the letter i with a subscript to denote a particular object identifier.

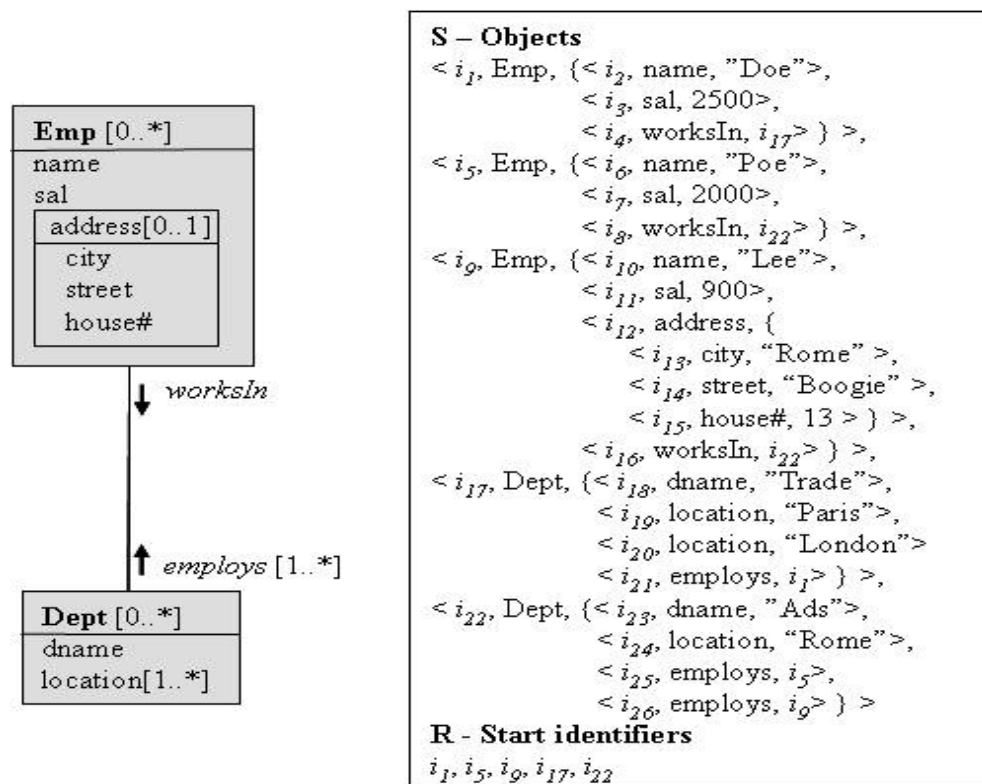Fig. 1. The AS0 model - a small database

In Figure 1 we present a database schema (which is an informal notion in AS0) and one of the possible database states according to the schema. Object specifications presented in the schema are associated with cardinalities, i.e. minimal and maximal number of occurrences in an actual database. Unlimited maximal cardinality is represented by *. Cardinalities [1..1] are

dropped. In Figure 2 we present the same database state in a graphical notation, where objects are boxes with round corners, sub-object boxes are depicted within their parent object boxes, pointer objects are represented by arrows and root identifiers are presented within circles.
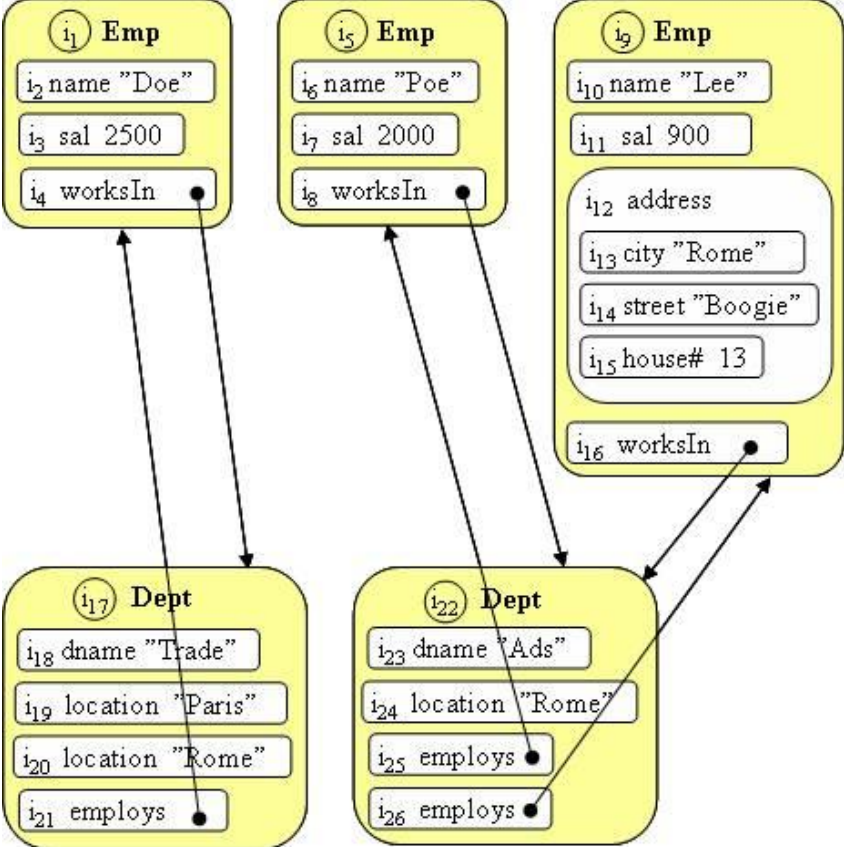

Fig. 2. The AS0 model - a graphical representation of a small database

4. SBQL queries

SBQL is based on the principle of compositionality, i.e. the semantics of a complex query is recursively built from the semantics of its components. SBQL queries are defined as follows:

1. A name or a literal is a query; e.g., 2, "Niklaus Wirth", *Book*, *author*.
2. $s\,q$, where s is an unary operator and q is a query, is a query; e.g., *count*(*Book*), *sin*(*x*).
3. $q_1\,t\,q_2$, where t is a binary operator, is a query; e.g., 2+2, *Book.title*, *Customer* **where** <condition>.

In SBA there are two special unary operators parameterized by a name. They are **as** and **qroup as**. If *q* is a query and *n* is a name, then *q* **as** *n* and *q* **group as** *n* are queries. These operators will be used in all cases which require defining an auxiliary name within a query.

A binary operator is either algebraic or non-algebraic. Algebraic operators do not use ENVS and cover typical operators known from other languages, such as arithmetic and string comparisons and functions, set-oriented operators, aggregate functions, auxiliary naming operators, Boolean operators, etc. In contrast, the semantics of non-algebraic operators assumes operations on ENVS. In general, non-algebraic operators cannot be expressed by any mathematically correct algebra defined in the style of the relational algebra; more discussion on this topic can be found in [1]. Non-algebraic SBQL operators include selection (where), projection/navigation and path expressions (dot), dependent join (join), existential and

universal quantifiers, sorting (<u>order by</u>), iterator (<u>for each</u>) and several variants of transitive closure. Auxiliary naming operators <u>as</u> and <u>group as</u> are defined through a very simple semantics based on the concept of *binder*.

5. Environment stack

We focus here on the conceptual view of ENVS rather than on its physical implementation. The construction of the stack will reflect basic concepts of the query languages' semantics that we have introduced in the AS0 store model. In contrast to typical PLs the stack is prepared to treat uniformly individual data and collections. The stack is a client-side main-memory structure. However, this is not a significant limitation, because nowadays main memories are very large and can be extended by virtual memories. Moreover, there are a lot of possible optimizations aiming reduction of the stack size. Thus the danger that the main-memory stack implies some limits in scalability does not occur. Because the stack is a client-side structure, its idea is orthogonal to geograhical distribution of database resources.

The environment stack consists of ordered sections. The *newest* section is named *top*, the oldest one is named *bottom*. As in case of PLs, each ENVS section presents information on some run-time environment. The size of a section is conceptually unlimited.

The bottom of ENVS contains global sections, in particular, a database section, a section related to the current user session, libraries that are available in the computer environment and some computer environment variables, such as date, time, user login information, etc. We assume that each name that may occur in a source query must possess its counterpart on ENVS, hence the stack will be the only name binding mechanism; no name occurring in a query can be bound otherwise.

ENVS should be prepared to store uniformly information on persistent and volatile objects. Persistent objects are usually shared among many user sessions. Volatile objects are properties of a single user session and are not available to other sessions. Local objects of procedures are considered volatile. ENVS will also be prepared to store all information that stems from definitions that are local for queries. Almost all query languages introduce such definitions; for instance, "correlation variables" in SQL, names defined by *as* operator in OQL, *for* operator in XQuery, names bound by quantifiers, cursors in *for each* statements, etc. All such names occurring in queries will be bound according to the same standard routine.

In PLs the run-time environment stack usually plays also the role of the *store of values* of all volatile entities (e.g. local variables of procedures). In contrast, we assume that the stack and the object store are separate notions. We have several reasons to separate them. The main reason is conceptual - the separation is quite convenient for the definition of the semantics of a query language. The second reason is that we consider persistent objects which live on a database server machine, while the stack lives on a client machine. Hence the stack may contain *references* to persistent objects, but not the objects themselves. The third reason is that the same object can be referred from several stack sections, thus it is impossible to include it to a single section. In implementation it is possible to make some optimizations which assume that some volatile objects live on the stack.

5.1 The concept of *binder*

A basic structure that is stored at ENVS is called *binder*. The concept is specific only for SBA, it does not occur in other approaches to query and programming languages. Binder is also a key concept for query languages defined through SBA. It is incredible how many semantic issues can be explained through such a simple notion. Binder supports universal

solutions to all semantic issues related to naming, scoping, binding, parameter passing, and definition of many query operators. Lack of this concept in former theories, (such as algebras and calculi) is caused by the fact that naming issues have the second-class citizenship in these theories: names are properties of informal meta-languages rather than the properties of defined languages. In contrast, naming issues are fundamental in all known query languages, including SQL, OQL and XQuery, hence lack of formal treatment of names inevitably results in underspecified semantics.

Binder is a pair $n(x)$, where n is an external name which can be written in a source query or program, and x is any result of a query, in particular, a reference to an object or a value. The idea of a binder is very simple. Its role is to binding names, hence it joins an external name n with a run-time entity x that is denoted by this name. Binding name n means that we are looking at ENVS for a binder (binders) $n(x)$. The result of the binding is x. If the stack contains no such a binder, the situation can be qualified as a binding error. Alternatively, for semi-structured data the situation can be qualified as correct, but the result of the binding is the empty bag. However, we reject such an alternative, because it makes impossible to distinguish absent data (normal in semi-structured data) from spelling mistakes in names.

5.2 Binding names

Fig.3 presents a sample state of the environment stack that address a volatile store and the persistent store from Fig.1 and Fig.2. In this example binders contain object identifiers (references) and values. Single references and single values are particular cases of query results, but in general, because query results can be nested and arbitrarily complex, binders can be nested and complex too. Note that the stack mixes and unifies volatile and persistent data. This is due to the orthogonal persistence principle which requires no difference between querying and operations on persistent and volatile data.
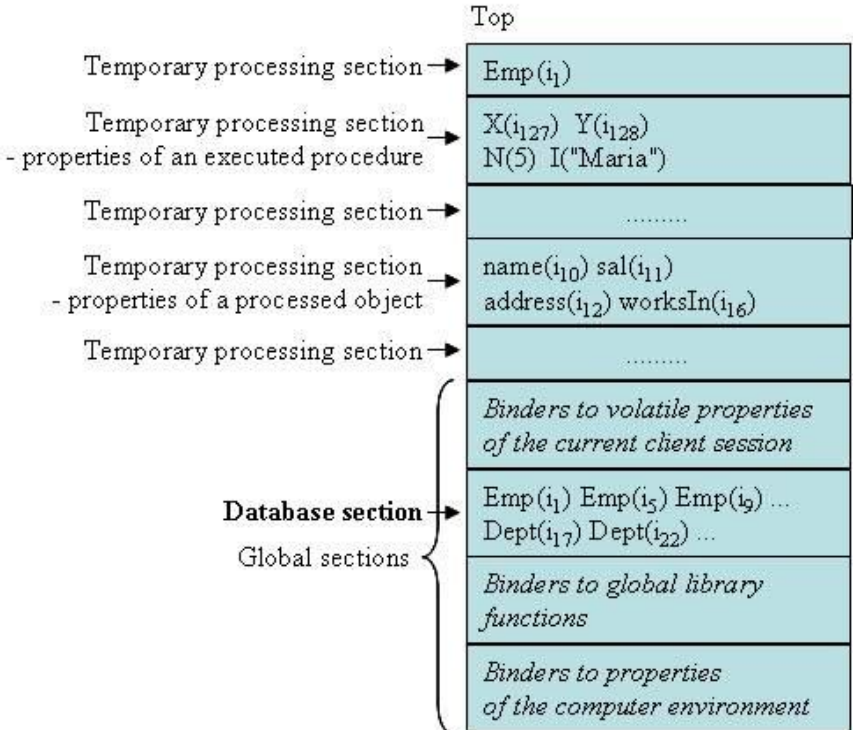

Fig.3. A sample state of an environment stack ENVS

In case of concurrent access of many clients, each of them maintains an own evironment stack. A server can also maintain an environment stack for own needs (e.g. for processing views). Moreover, if a client session has some parallel processes, threads or transaction, then

6

it is possible that there are as many stacks as processes/threads/transaction within the session. When a session (process/thread/transaction) is started a stack dedicated to it is created and fulfilled in advance by global sections, as shown in Fig.3. Then the stack is maintained by the query execution engine, as will be determined in the following parts of this SBA description.

The semantics of a query is a function which maps a state into a query result. Because in SBA each single name is a legal query (c.f. Section 4), the question is how the mapping looks for it. This mapping we call *binding*. Binding is performed on ENVS according to the very simple "search from the top" rule. Some stack sections must be skipped because of scoping rules; so far we abstract from this feature. Fig.4 presents the order of searching within ENVS. Binding name n requires searching for the binder $n(x)$ which is present on the stack, according to the rule. The first success stops the search. The result of the binding is x. To take into account collections we assume *multi-valued binding*. If the binding of a name n succeeds in some section, and it contains more binders named n: $n(x_1)$, $n(x_2)$, ..., $n(x_k)$, then all such binders are taken into account and the result of the binding is the bag$\{x_1, x_2, ..., x_k\}$. The function that does the search we call *bind*. It has a name as an argument and returns a value being the result of the binding. The result will be stored at another *query result stack* that will be explained later. The function is quite easy to explain, for instances see Fig.4.

$bind(X) = i_{127}$ $\qquad$ $bind(I) = $ "Maria" $\qquad$ $bind(sal) = i_{11}$
$bind(Dept) = $ bag$\{ i_{17}, i_{22}, ...\}$ $\qquad$ $bind(Emp) = i_1$

Fig.4. The order of searching ENVS during the binding a name

Note that in the last case there is a section that contains more binders named *Emp*; however, because the top section contains a single binder named *Emp*, the search is terminated on this binder. This feature is known as *overriding*; it has significance in query languages that we intend to cover and more generally, in object-oriented query/programming languages. The overriding accomplishes the principle of a local context (lexical scoping), which provides that a particular name is bound in the environment that is currently most close according to the programmer thinking. Note also that the binder $Emp(i_1)$ occurs two times on the stack. Such a case is normal in query languages that we want to address.

5.3 Results Returned by Queries

In SBA we assume that queries never return objects but references to objects, sometimes within more complex structures. Objects live in the object store; no entity called object occurs elsewhere. Queries can also return values stored in objects and values calculated by some functions or algorithms. As in other approaches, we introduce structures, bags and sequences as results of queries. An essential novelty of SBA in comparison to other theories and proposals of query languages is that queries can return named structures that we already know as *binders*. The results of queries are defined as follows:

- Each atomic value can be a result of query.
- Each reference to an object (object identifier) can be a result of query.
- If x can be a result of a query and n is an external name, then the binder $n(x)$ can also be a result of a query. Such a result we will also call *named value*.
- If x_1, x_2, x_3, ... can be results of a query, then **struct**$\{ x_1, x_2, x_3, ... \}$ can also be a result of a query. **struct** is a structure constructor. The order of elements in a structure can be significant. In contrast to typical structures known from e.g. C/C++, we do not assume that all elements of structures must be named.
- If x_1, x_2, x_3, ... can be results of a query, then **bag**$\{ x_1, x_2, x_3, ... \}$ and **sequence**$\{ x_1, x_2, x_3, ... \}$ can also be results of a query. **bag** and **sequence** are collection constructors,

Note that **struct, bag** and **sequence** are *constructors of values* rather than constructors of types. **struct**$\{ x_1, x_2, x_3, ... \}$ is considered an individual element (it is not a collection), while **bag**$\{ x_1, x_2, x_3, ... \}$ and **sequence**$\{ x_1, x_2, x_3, ... \}$ are collections

5.4 Opening a New Section of ENVS

In classical programming languages opening a new scope on the environment stack is caused by entering a new procedure (function, method) or entering a new block. Respectively, removing the scope is performed when the control leaves the body of the procedure or the body of the block. To these classical situations of opening a new scope on the environment stack we add a new one. It is the essence and motive of SBA. The idea is that some query operators (called non-algebraic) behave on the stack similarly to program blocks. For instance, in the SBQL query:

$$Emp \textbf{ where } (\ name = \text{"Poe" } \textbf{and } sal > 1000 \ )$$

the part ($name$ = "Poe" **and** sal > 1000) behaves as a program block executed in an environment consisting of the interior of an *Emp* object. Because we have assumed that each name occurring in a query has to be bound on the stack, this concerns also names *name* and *sal*. Hence, to make the binding possible we push on ENVS a section with the interior of the currently processed Emp object. The situation is illustrated in Fig.5. The operator **where** iterates over the result of the query *Emp*. In each cycle it pushes onto the ENVS a section with the internal environment of a next *Emp* object. After processing of the condition the section is removed from the stack. Note that the interior of an *Emp* objects is represented on the stack as a set of binders leading to properties of the object.
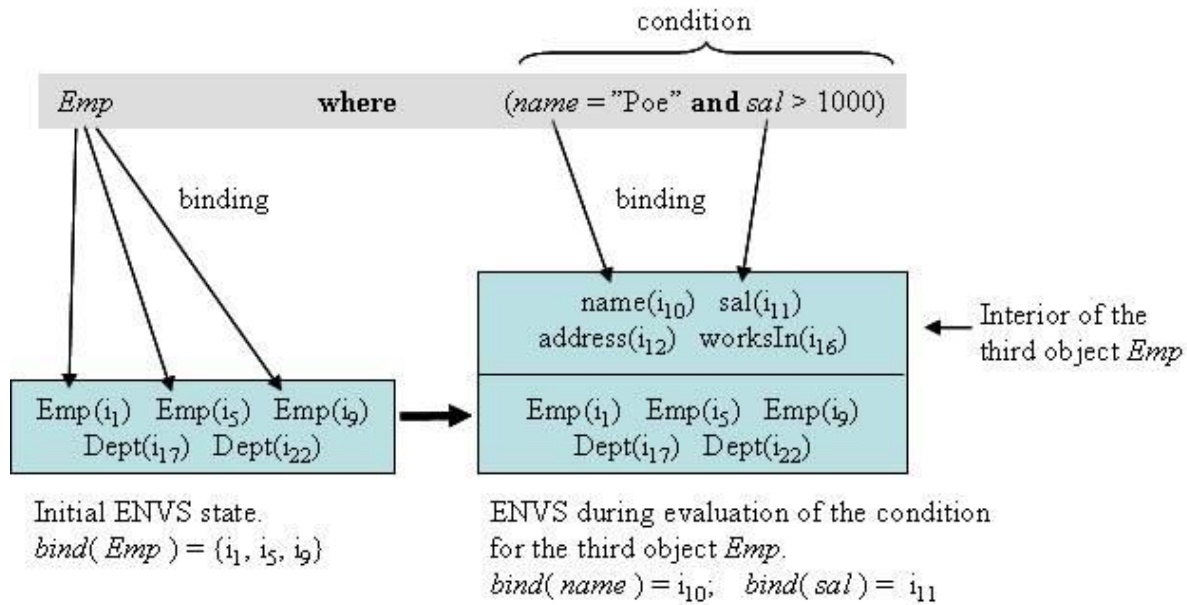
Fig.5. Pushing a new section on ENVS to evaluate a condition

5.5 Function *nested*

We have introduced the concept of *interior* of an object, which is to be pushed on the top of ENVS. Now we formalize the concept. In the formalization we strive to generalize it to cover all the cases that may require pushing on the stack some new environment. The formalization is quite simple and it is expressed through the function named *nested*. The function takes any query result (as defined previously) as an argument and is implicitly parameterized by an object store. For the argument it returns a set of binders. For a complex object *nested* returns binders to the interior of this object. For a pointer object *nested* returns the binder of the object that it points to. For a binder *nested* returns this binder. This semantics is consistent with the typical understanding of auxiliary names introduced in queries. For a structure *nested* returns the union of the results of the *nested* function applied. For other arguments the result of *nested* is the empty set.

6. Advanced Notions of Stack-Based Architecture

6.1 Classes, Roles and Inheritance

The AS1 store model introduces classes and (multiple-) inheritance in the classical variant known from languages such as C++ and Java and modeling tools such as UML. This new feature requires rather obvious and relatively small extensions to SBQL defined for the AS0 store model. The changes concern only non-algebraic operators and behavior of ENVS. Invocation of methods can also be easily expressed through actions of a stack-based machine.

AS2 introduces dynamic object roles, which assume that an object consists of sub-objects that are its *roles*; each role dynamically inherits properties of its super-role, in particular, of the entire object. Roles can be dynamically inserted and deleted from an object. This feature requires small extensions to SBQL defined for AS0 and AS1. The changes concern non-algebraic operators and behavior of ENVS.

6.2 Programming Capabilities

SBQL queries can be embedded within statements that can change the database or program state. Concerning this issue SBQL follows state-of-the-art standards known from majority of

9

programming languages. Typical imperative constructs are *creating* a new object, *deleting* an object, *assigning* new value to an object (updating) and *inserting* an object into another object. Program control and loop statements such as *if…then…else…,* switch, *while* loops, *for each* iterators and others are also introduced. Some peculiarities are implied by queries that may return collections; thus there are possibilities to generalize imperative constructs according to this new feature.

In classical programming languages procedures, functions and methods (procedural abstractions) are accomplished by the stack-based machine and there is no essential difference concerning SBQL. All procedural abstractions of SBQL can be invoked from any procedural abstractions and can be recursive.

The stack-based machine is prepared to specify precisely various parameter passing methods, including variants of *call-by-value*, *call-by-reference*, *strict-call-by-value*, etc. SBQL deals with parameters being any queries; thus corresponding parameter passing methods are generalized to take collections into account.

SBQL assumes orthogonal encapsulation, where each kind of property (a method, an attribute, a pointer, etc.) can be public or private, depending on the decision of the designer or programmer. Encapsulation implies rather minor changes to the stack-based machine and to the semantics of SBQL.

6.3 Virtual Updatable Views

SBQL involves a new method for updatable views that allows one to achieve the power that has not been even considered so far in the database domain. The method has some commonalities with i*nstead of* trigger views implemented in Oracle, SQL Server and DB2. In general, the method is based on overloading generic updating operations (create, delete, update, insert, etc.) acting on virtual objects by invocation of procedures that are written by the view definer. The procedures are the inherent part of the view definition. The procedures have full algorithmic power, thus there are no limitations concerning the mapping of view updates into updates of stored data. SBQL updatable views allow one to achieve full transparency of virtual objects: they cannot be distinguished from stored objects by any programming option. This feature is very important for distributed and heterogeneous databases. SBQL views can be used as *mediators* on top of local resources to convert them virtually to the required format, as *integrators* that fuse fragmented data from different sources, and as *customizers* that adopt the data to the needs of a particular application.

6.4 Classes, Interfaces, Types, Schemas and Metamodels

SBA/SBQL clarifies these concepts by assigning pragmatic roles to them. Classes are source code units that contain implementation; after compilation they became special kind of database objects employed by the stack-based machine. Interfaces are external specifications of access to objects; they contain no implementation. Interfaces need not 1:1 match classes - one class may have zero or many interfaces and one interface can be defined for more than one class (due to views). Interfaces usually contain more information than types. Types are determined by interfaces, but types can also exist without interfaces. Types are intended as constraints on the construction and behavior of any program entities (in particular, modules, objects, values, links, procedures, etc.) and constraints on the query/programming context in which these entities can be used. Schemas are external (application programmer oriented) specifications of a database content and are inevitable pragmatic part of a query/programming languages. A metamodel is an internal representation of a schema; it is internally used by the database management system and externally for generic programming with reflection.

6.5 Static (semi-) Strong Typing

SBQL is based on a new strong typing theory. It distinguishes *internal* and *external* type systems. The internal type system reflects the behavior of the type checking mechanism, while the external type system is used by the programmer. A static strong type checking mechanism simulates run-time computations during compile time by reflecting the run-time semantics with the precision that is available at the compile time. Roles of the SBQL typing system are the following: compile-time type checking of query operators, imperative constructs, procedures, functions, methods, views and modules; user-friendly, context dependent reporting on type errors; resolving ambiguities with automatic type coercions, ellipses, dereferences, literals and binding irregular data structures; shifting type checks to run-time, if it is impossible to do them during compile time; restoring a type checking process after a type error, to discover more than one type error in one run; preparing information for query optimization by properly decorating a query syntax tree. The internal SBQL type system includes three basic data structures that are compile-time counterparts of run time structures: a metabase, a static environment stack and a static result stack. Static stacks process type signatures − typing counterparts of corresponding run time entities. Signatures are additionally associated with attributes, such as mutability, cardinality, collection kind, type name, multimedia, etc. For each query/program operator a decision table is provided, which determines allowed combinations of signatures and attributes, the resulting signature and its attributes, and additional actions.

6.6 Query optimization

In SBQL new methods of object-oriented query optimization are implemented. One group of the methods is based on rewriting rules: factoring out independent subqueries, employing the distributiveness property, removing dead subqueries, query modification and query tail absorption. Another group concerns the methods based on indices, which include utilization of dense and sparse indices, fast linear hashing indices and index management utilities. Currently new methods are developed, including the methods based on caching, pipelining, cost models and optimization methods for distributed databases.

7. More SBQL Examples

In this section we show some examples of SBQL queries addressing the schema and the store presented in Fig.1 and Fig.2.

- Get all information on departments for employees named Doe:
  (*Emp* **where** *name* = "Doe").*worksIn.Dept*

- Get names and cities of employees working in departments named Toys:
  (*Dept* **where** *name* = "Toys").*employs.Emp*.
  (*name*, **if exists**(*address*) **then** *address.city* **else** "No address")

- Get the average number of employees in all departments:
  avg(*Dept*.**count**(*employs*))

- Get names of departments and the average age of their employees:
  Dept . (*dname*, **avg**(*employs.Emp.age*) **as** *a*)

- Get departments together with the average salary of their employees:
  Dept **join avg**(*employs.Emp.sal*)

- For each employee get the name and the percent of the annual budget of his/her department that is consumed by his/her monthly salary:

  Emp . (name **as** *n*, (((**if exists**(*sal*) **then** *sal* **else** 0) **as** *s*).

  $$((s * 12 * 100)/(worksIn.Dept.budget)) \text{ as } percentOfBudget)$$

- For each person having no salary give the minimal salary in his/her department:

  for each (*Emp* **where not exists**(*sal*)) **as** *e* **do**

      e.changeSal(**min**(*e.works_in.Dept.employs.Emp.sal*))

8. Conclusion

In this paper we have described the idea of the Stack-Based Architecture and the Stack-Based Approach. We have attempted to justify that SBA/SBQL is a holistic semantic framework which can be the foundation for the development of integrated query/programming language with no impedance mismatch. We have explained the semantics of SBA/SBQL based on the notion of the environment stack. We have also shown the abundance of advanced notions which can be created on top of basic SBA/SBQL ideas.

Recently OMG formed a working group to create a new standard on object databases. Currently the group examines SBA as a possible starting point for this standardization process. The group preliminarily proposes the definitions and semantic mechanisms of SBA for the construction of a complete and correct object model that supports a powerful object query language AOQL as well as a complete and correct set of native programming languages' bindings [11].

References

1. SBQL Web pages: http://www.sbql.pl/

2. Various information on SBQL, including recent presentations and implementations: http://www.sbql.pl/various

3. Papers and reports: http://www.sbql.pl/articles

4. PhD theses: http://www.sbql.pl/phds

5. M.Lentner, K.Subieta: ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development. Springer LNCS 4690, 130-140

6. K.Subieta. *Theory and Construction of Object-Oriented Query Languages* (in Polish), PJIIT - Publishing House, 2004, 522 pages

7. K.Stencel: *Semi-strong Type Checking in Database Programming Languages* (in Polish), PJIIT - Publishing House, 2006, 207 pages.

8. J.Płodzień, A.Kraken. *Object Query Optimization through Detecting Independent Subqueries.* Information Systems 25(8), Pergamon Press, September 2000, pp. 467-490

9. K.Subieta, Y.Kambayashi, J.Leszczyłowski. *Procedures in Object-Oriented Query Languages.* Proc. 21-st VLDB Conf., Zurich, 1995, pp.182-193

10. K.Subieta, C.Beeri, F.Matthes, J.W.Schmidt. *A Stack-Based Approach to Query Languages.* Proc. 2nd East-West Database Workshop, 1994, Springer Workshops in Computing, 1995, 159-180

11. OMG Object Database Technology Working Group: *Next-Generation Object Database Standardization, OMG White paper,* http://www.omg.org/docs/mars/07-09-13.pdf, September 27, 2007