

Rafał Hryniów, Michał Lentner
Krzysztof Stencel, Kazimierz Subieta
Types and Type Checking
in Stack-Based Query Languages

Nr 984

Warszawa, March 2005

Abstract

In this report we propose a new approach to types and static type checking in object-oriented database query and programming languages. In contrast to typical approaches to types which involve very advanced mathematical concepts we present a type system from the practitioners' point of view. We argue that many features of current object-oriented query/programming languages, such as ellipses, automatic coercions and irregularities in data structures, cause that very formal type systems are irrelevant to practical situations. We treat types as some syntactic qualifiers (tokens or some structures of tokens) attached to objects, procedures, modules and other data/program entities. Such syntactic qualifiers we call *signatures*. We avoid the simpleminded notion that a type has some internal semantics e.g. as a set of values. In our assumptions a type inference system is based on predefined decision tables involving signatures and producing type checking decisions, which can be the following: (1) type error, (2) new signature, (3) dereference, coercion and/or delegation of a type check to run-time. A type inference decision table is to be developed for every query/programming operator. Type inferences are implied by the stack-based approach (SBA) to object-oriented query/programming languages. Static type checking is just a compile time simulation of the run-time computation. Thus the type checker is based on data structures that statically model run-time structures and processes, that is: (1) metabase (internal representation of a database schema, a counterpart of an object store), (2) static environment stack (a counterpart of run-time environment stack), (3) static result stack (a counterpart of run-time result stack) and (4) type inference decision tables (a counterpart of run-time computations). Then, we present the static type check procedure which is driven by the metabase, the static stacks and the type inference decision tables. To discover several type errors in one run we show how to correct some type errors during the type check. Finally we present our prototype implementation showing that our approach is feasible and efficient with moderate implementation effort.

Key words: type, static type checking, query language, database, object-oriented, data schema, Stack-Based Approach.

Streszczenie

W raporcie proponujemy nowe podejście do typów i statycznej kontroli typologicznej w obiektowych językach zapytań/programowania. W przeciwieństwie do podejść wykorzystujących zaawansowane koncepcje matematyczne prezentujemy tu pozycję praktyków. Wiele cech obecnych języków zapytań/programowania, takich jak elipsy, automatyczne koercje oraz nieregularności struktur danych, powodują, że bardzo formalne systemy typologiczne nie odpowiadają praktyce. Proponujemy typy jako syntaktyczne kwalifikatory (znaki lub struktury znaków) przypisane do obiektów, procedur oraz innych bytów programistycznych. Takie kwalifikatory nazwaliśmy *sygnaturami*. Unikamy popularnego punktu widzenia, w którym typ posiada wewnętrzną semantykę, np. w postaci zbioru wartości. System wnioskowania o typie jest oparty na tabelach decyzyjnych działających na sygnaturach i generujących decyzje w zakresie kontroli typologicznej, które mogą być następujące: (1) błąd typologiczny, (2) nowa sygnatura, (3) dereferencja, koercja i/lub oddelegowanie kontroli typu do czasu wykonania. Tablice decyzyjne powinny być sporządzone dla każdego operatora występującego w zapytaniach/programach. Wnioskowanie o typie jest implikowane przez podejście stosowe (SBA) do obiektowych języków zapytań/programowania. Styczna kontrola typologiczna symuluje podczas kompilacji tę sytuację, która zajdzie podczas czasu wykonania. Stąd kontroler typów jest oparty na strukturach danych, które statycznie modelują struktury i procesy czasu wykonania, tj.: (1) metabaza (wewnętrzna reprezentacja schematu, odpowiednik składu obiektów), (2) statyczny stos środowiskowy (odpowiednik stosu środowiskowego), (3) statyczny stos rezultatów (odpowiednik stosu rezultatów), (4) tablice decyzyjne wnioskowania o typie (odpowiednik operatorów). Następnie prezentujemy procedurę statycznej kontroli typów, której działanie jest oparte na metabazie, statycznych stosach i tabelach decyzyjnych. Aby wykryć wiele błędów typologicznych w jednym przebiegu pokazujemy, jak należy skorygować pewne błędy typologiczne podczas kontroli typologicznej. Na końcu prezentujemy prototypową implementację pokazującą, że nasze podejście jest osiągalne i efektywne przy umiarkowanym wysiłku implementacyjnym.

Słowa kluczowe: typ, statyczna kontrola typologiczna, język zapytań, baza danych, obiektowo-zorientowany, schemat danych, Podejście Stosowe.

Content

1. INTRODUCTION	6
2. STACK-BASED APPROACH (SBA)	14
2.1 OBJECT STORE MODEL	15
2.2 ENVIRONMENT STACK AND NAME BINDING	18
2.3 STACK BASED QUERY LANGUAGE (SBQL)	20
3. METABASE GRAPH	21
4. INTERNAL TYPES OF VALUES	24
5. TYPE INFERENCE	27
5.1 ARITHMETIC AND STRING OPERATORS (ALGEBRAIC)	27
5.2 UNION, INTERSECTION AND DIFFERENCE (ALGEBRAIC)	27
5.3 STRUCTURE CONSTRUCTOR (ALGEBRAIC).....	28
5.4 AUXILIARY NAMES (ALGEBRAIC)	29
5.5 NAVIGATION (NON-ALGEBRAIC)	29
5.6 DEPENDENT JOIN (NON-ALGEBRAIC)	30
5.7 SELECTION (NON-ALGEBRAIC)	31
5.8 QUANTIFIERS (NON-ALGEBRAIC)	31
5.9 TRANSITIVE CLOSURE (NON-ALGEBRAIC).....	31
6. STATIC TYPE CHECKING	32
7. EXAMPLE STATIC TYPE CHECK	37
8. AUGMENTING THE QUERY	41
8.1 ELLIPSIS (AUGMENT 1).....	41
8.2 DEREFERENCE (AUGMENT 2)	42
8.3 COERCION (AUGMENT 3).....	43
9. RESTORING THE PROCESS	44
9.1 MISSPELLED NAME (RESTORATION 1).....	45
9.2 MALFORMED SUBQUERIES (RESTORATION 2).....	46
9.3 MALFORMED LEFT SUBQUERY (RESTORATION 3).....	47
10. EXTERNAL TYPES	47
11. ODB AND ITS TYPE SYSTEM	48

11.1	TYPES AND VARIABLES	49
11.2	DATA STORE AND METABASE.....	50
11.3	PARSER AND TYPE CHECKER	51
11.4	SEMANTIC CHECKING IN ODRA.....	52
11.5	THE PROTOTYPE AT WORK.....	56
12.	CONCLUSIONS.....	57
13.	REFERENCES	57

1. Introduction

Strong type checking is an important feature of a programming language. It protects programmers from many of their own mistakes. Thus supports the reliability of software. It has been assessed that strong typing detects up to 80% of semantic and conceptual errors in a developed software code. Therefore the presence of a type system within a programming language is referred to as *type safety* and it is considered by many professionals as a fundamental principle of programming languages. A type system has also great meaning in conceptual modelling, because carefully chosen names of types are also domain qualifiers of business entities. Types, especially atomic ones, bear information on representation of data, in particular, the sizes of data items and physical coding methods. This feature is particularly important in distributed and heterogeneous applications. Physical representation determined by types is a basic component of interoperability and data exchange standards.

In this report we discuss strong type checking of object-oriented or XML-oriented query languages integrated with programming languages. We also propose a new approach to the problem. Apparently, the topic is exhausted by the current state-of-the-art. The history of strong typing has more than 30 years and evolved from the Pascal type system based on name equivalence up to modern concepts based on structural equivalence and inclusion/parametric polymorphism. There are thousands of papers devoted to types and a lot of strong mathematical theories. Many of them deal with bulk types (collections) typical for databases and database programming languages, including query languages. There are also many practical proposals of typing systems implemented in popular object-oriented programming languages (e.g. C++ or Java) as well as in research prototypes, e.g. PS-Algol, DBPL, Galileo, Napier-89, Fibonacci, and many others. An overview of these proposals from the strong typing perspective can be found in [AtBu87, Atki95].

Experts in strong typing usually adopt the simpleminded notion that a type is a (possibly infinite) set of values and that a variable of that type is constrained to assume values of that type [AtBu87]. Mathematically, the notion is clear and (what is important for formal research) makes it possible to develop very advanced type theories with a lot of formal properties and theorems. However,

there are features of query/programming languages and environments that make it difficult to adopt the above notion. We subdivide these features into four groups:

- Irregularities in data structures, such as null values (optional data), repeating data (collections with various cardinality constraints), exclusive variants/unions, pointer links between data, unconstrained data names (see e.g. CORBA Property Service [CORBA] or attributes in XML), and perhaps others. Such irregularities make the strong typing much more difficult (or even impossible) and may cause the necessity for shifting type checking to run time.
- Ellipses, automatic coercions, automatic dereferences and other options of query/programming languages. Such features are introduced by designers of database query/programming languages to increase user friendliness of programming interfaces and to relieve the programmers from annoying, too formalistic and too verbose style of writing queries/programs.
- Features of types not covered by above simpleminded type notion, such as mutability, collection cardinality constraints, collection types (set, bag, sequence, etc.), type names (for name type equivalence), methods/procedures/functions, and perhaps others.
- Other properties of the programming environment, such as interfaces, classes, abstract data types, inheritance and multiple inheritance, dynamic object roles and dynamic inheritance, modules, export and import lists, scoping rules for names occurring in queries, etc.

Although the literature contains explanations of some notions (e.g. multiple inheritance [Card84], or abstract data types [Mitc88]) in general, such explanations suffer as a rule from two kinds of drawbacks:

- They assume a very strict formal model which hardly meets practical situations;
- They isolate some particular aspect from other aspects of the strong typing problem. In practice, however, all aspects must be considered together and no such isolation of a particular aspect is possible.

Such papers are like islands in the sea of decisions which the developers must take during development and implementation of a type system. As our experience has shown, typing environments for object-oriented or XML-oriented databases and their query languages present so many peculiarities that

the ideas presented in these papers are not much helpful and in most cases inapplicable.

An example of such problems concerns object names. Usually database schemata determine the names of database objects because the names bear external (business) semantics. Hence, an object name should be a property of the object type. Unfortunately, type systems known from programming languages do not assume that object name is a type invariant. E.g. for the type *StudentType* the programmer can create objects named *Student*, but as well s/he can create objects named *x*, *y* or *z*. And v/v: typical programming languages assume that each component of a structure (a record in Pascal terms) must possess a unique name. But for query languages this is not obligatory: in SBQL (due to our attempts to achieve the conceptual closure) components of structures can be named or unnamed (e.g. when a structure consists of references to database objects). Moreover, if a reference component is named, the name attached to it can be different than the name of an object that the reference points to.

The ODMG standard for object-oriented databases [ODMG00] has shown basic difficulties with strong typing of database query/programming languages. The developers of the standard seriously approached the development of the type system, materializing it in the form of the Object Definition Language (ODL), in the CORBA IDL [OMG02] style. Their approach, although based on reasonable engineering trade-offs, have roots in notions that the typing community have developed for years. Unfortunately, the approach did not result in an acceptable proposal. To our knowledge, till now there is no successful implementation of the standard. Moreover, the approach has met severe criticism as inconsistent if one considers all its features [Alag97]. Obviously, some part of it can be implemented, but this undermines the proposal as a thorough database standard.

Developers of query/programming languages take various attitude to types. For some languages, especially for script-oriented ones, types are neglected at all (e.g. Smalltalk, PHP, etc.). The reason is that such languages are initially designed for small scale or niche applications, thus the developers do not consider types as important. When popularity of a particular language grows up to large mission-critical applications, types are very difficult or impossible to introduce without affecting many already written applications and without violating the skills and habits of thousands of programmers. In contrast to the dot-com boom period, the commercial world is currently oriented towards

short-term return of investment rather than long term predefined strategy. Thus such bottom-up development of lower quality programming technologies has deeper economical justification.

For typical object-oriented languages (e.g. C++ or Java) types are represented in a relatively straightforward manner and suitable type equivalence algorithms are not hard to specify and implement. In contrast, a trend in modern programming languages, particularly in database programming languages, is to provide more and more sophisticated type systems which allow more program errors to be detected statically. This is pushing knowledge of static type checking to its limits determined by acceptance of programmers and implementation challenge. Sophisticated type systems cause type specifications to be extremely large, complex and obscure, thus requiring more effort from programmers, especially concerning the learning. In effect, sophisticated type systems may undermine the general goals of type systems, since they may not increase the productivity of programmers by relaxing the testing phase. Moreover, implementation of sophisticated type systems can be very challenging, requiring deeper research into implementation methods. An example of these difficulties is the SML polymorphic type system [SML04] developed by the Microsoft Research, which on one hand is a challenging implementation problem, and on another hand its impact on the programmers' productivity is unknown and can be questioned (it is not object-oriented and does not deal with collections, associations and persistency, what is not according the current trends in software development).

In popular programming languages types are properties of programs and all the type control is in hands of a particular programmer, who declares types and uses them to specify local program entities. The *data independency* concept pushes types towards data, which are designed, administered and maintained independently of programs. As the result, the notion of the *data schema* or the *database schema* has appeared. The data schema plays two roles:

- Gives the information on the content, organization, ontology and constraints related to data stored in a database. Programmers use this information while writing queries and programs. A schema is an inevitable part of the programmers' interface to the database and participates in the conceptual modeling of the database content, allowing the programmer to understand this content and to write proper manipulation programs addressing this content.

- It is used internally by the database management system to maintain and check the data stored in the database. These checks usually go beyond the role traditionally expected from the type checking mechanism, e.g. a schema may be used to dynamic object checking rather than to static program checking (c.f. the role of XML Schema).

In terms of programming languages database schemata are declarations of objects (or other data structures) rather than pure types. Moreover, a data schema can contain features not relevant to types, such as integrity constraints (e.g. functional dependencies, primary keys and referential integrities), specification of exceptions and reactions to exceptions, assertions (e.g. pre- and post-conditions), side effects of functions/methods, etc. There are many proposals of schema languages for various environments, e.g. CORBA IDL, ODMG ODL, XML DTD, XML Schema, RDF Schema, and so on. In the Model Driven Architecture (MDA) [MDA05] a schema is simply an UML class diagram. Such a proposal can be advocated as the most natural, “seamless” transition from an UML conceptual model to the corresponding databases model. From another side UML class diagrams are not precise enough considering specification of data structures and their types. For instance (compare ODMG), it is unclear how UML handles (nested) collections, (multiple-) inheritance, associations/aggregations, null values and variants, etc. Nevertheless, minimizing the conceptual distance between UML class diagrams and some conceivable database description language (allowing for algorithmically precise specification of data structures and their types) we consider an important goal currently pursued by our research group.

Data schemata partly release programming languages from keeping type information. Run-time libraries or programs read this information from schemata stored by databases. In this way types become neutral to programming languages. This assumption is the basis of the type systems of CORBA [OMG02] and ODMG [ODMG00].

However, the source code of a program needs types as well. Therefore, hybrid solutions are proposed. Such solutions consist in textual mapping of types (defined e.g. in CORBA IDL or ODMG ODL) to the types specific to the programming language. This mapping is performed by a precompiler. The .NET environment contains a unified type system for all programming languages of this platform. This type system allows keeping types of objects independently of any programming language.

Another difficulty with database typing systems is that static typing information is usually tangled with other notions such as classes, interfaces, abstract data types, inheritance, dynamic object roles, name spaces, etc. A huge variety of concepts, definitions, approaches and judgments have caused a lot of chaos, which especially is experienced by persons who are trying to develop and implement an own static typechecker for a query/programming language based on a bit advanced object-oriented database model.

An example of difficulties concerns so-called recursive types. A typical argument for advanced typing theories concerns situations such as presented in [Cann89]:

$$\text{Point} = \{ x: \text{void} \rightarrow \text{Real}, y: \text{void} \rightarrow \text{Real}, \\ \text{move}: \text{Real} \times \text{Real} \rightarrow \text{Point}, \text{equal}: \text{Point} \rightarrow \text{Boolean} \}$$

In this example the body of the type Point definition contains four methods, where last two refer to the Point type; hence the definition is recursive. Moreover, this recursion has no ending condition: it resembles an infinite loop or an *ignotum per ignotum* definitional error. To resolve such cases authors have proposed so-called *F-bounded polymorphism*, which is an advanced formal notion. In [Alag94] the notion is adopted for object databases and proposed as a cure for flaws in the ODMG ODL specification [Alag97].

On the other hand, type specification, as presented above, are normal for database schema and imply no special implementation challenge. Thus we have asked fundamental queries: if implementation is easy, why theories are so complex? Do we need such theories and what is the reason to follow them? Similar questions concern such concepts as inheritance, subtyping and the substitutability principle. Again we have discovered that straightforward implementation of these concepts is much easier than formal theories supporting them.

Deeper investigations concerning such fundamental queries and confronting them with our implementation experience has led us to the following conclusions:

- The notion that a type is a (possibly infinite) set of values and that a variable of that type is constrained to assume values of that type is misconception and *should be rejected*. The notion is completely unnecessary during implementation of a strong static type checking system.

- The recursion presented in the above example is apparent. There is no recursion at all. References to type Point within the body of the above definitions present in fact the notion that in UML is known as *association*.
- Types should possess no deep semantics. Internally, types are represented by some words or symbols, referred later to as *signatures*. Signatures have no semantics, they are simply some tokens or structures of tokens. Static type inference rules are a kind of decision tables acting on signatures and producing signatures. Such decision tables are to be developed and implemented for all query operators. The mechanism can be easily extended to imperative constructs, methods, procedures, views, etc.

Note that these conclusions imply rejection of majority of current type theories, especially having roots in functional languages, as irrelevant to practical cases. Complexity and irregularity of the type inference rules (decision tables) convinced us that developing any elegant and powerful mathematical type theory is unfeasible.

Our idea of type checker follows the Stack-Based Approach [Subi95a, Subi95b, Subi04], which explains the mechanism of query evaluation for object-oriented databases in terms of mechanisms well known in programming languages. The mechanism involves three basic architectural elements:

- Object store, which is a formal structure representing objects stored in a database.
- Environmental stack, the structure responsible for scoping, binding, non-algebraic query operators, method or function invocations, parameter passing, implementation of object-oriented concepts (such as inheritance and polymorphism), and other aspects.
- Result stack, the structure accumulating temporary and final query results.

The idea of static typechecker consists in simulating the above three run-time structures during the compile time. Thus the static typechecking mechanism consists of the following architectural elements:

- Metabase (counterpart of an object store), which is a formal data structure representing database schema.
- Static environmental stack (counterpart of a run-time environmental stack) that stores so-called *static binders*, i.e. named signatures. The structure

during compile time simulates all the operations that are made on the environmental stack by query/program run-time mechanism.

- Static result stack (counterpart of a run-time result stack), the structure that accumulates signatures being the type description of corresponding run-time temporary and final query results.
- Type inference decision tables assigned to all operators introduced in a given query/programming language. A particular rows of a decision table determine signatures of the given operator arguments and the decision, which can be one of three cases: (1) can determine the resulting signature for the given operator and signatures of its arguments; (2) can qualify situation as a type error; (3) can qualify the situation as impossible to check statically thus contains the decision of making run-time coercions and/or pushing the type check to run-time.

As will be shown, the development of the type inference decision tables is an art with no obvious rules. It depends on engineering tradeoffs between a very rigorous attitude to strong typing (typical e.g. for Pascal-like languages) and less rigorous attitude, which is necessary in loosely typed, semistructured data environments (c.f. XML). Some decision tables have very arbitrary, irregular construction, thus we do not believe that the design of a type system can be supported by some elegant mathematical theory. For instance, if we take into account the operator `+` and argument signatures *integer* and *string*, the decision can be the following: (a) a type error; (b) coercing string to integer and a typecheck during run time (`+` is arithmetic); (c) coercing integer to string and a typecheck during run time (`+` is concatenation). Although the decision (a) would be probably the most close to what is commonly understood as “strong typing”, in many cases decisions such as (b) or (c) are also reasonable.

Another peculiarity of our approach to typing concerns collections. The concept of collection in case of object-oriented database models is not so obvious and easy as one can expect. Collections, especially heterogeneous ones, combined with inheritance (subtyping) and substitutability lead to conceptual difficulties. For this reason in this report we substitute the notion of collection by cardinalities, in the spirit of UML association cardinality constraints. Such an approach allows us for much more precise determining the nature of collections. Cardinalities are additional attributes added to signatures. Type inference decision tables take the cardinalities into account.

In this report we make the distinction between external and internal type systems. An *external type system* is a user-friendly language for description types and schemata. The external type system is a part of user interface to the database. Usually the external type system does not stand alone but is entangled in the definition of other artifacts like interfaces, classes, modules, functions and methods. An *internal type system* is usually richer than the external one and consists of definitions of signatures, metabase, static environment stack, static result stack and type inference decision tables. In the report we mostly focus on the internal system, as understanding of it is crucial for implementing a static typechecker for any conceivable database environment.

All the features discussed above cause that our approach to strong typing is very different from everything what till now has happened in the domain. This originality is supported by the experimental implementation in the ODRA system, where we have shown that the approach is implementable, is efficient and has many advantages over current proposals for database typing systems.

The rest of the report is organized as follows. In section 2 we briefly present the stack-based approach. In section 3 we present the syntax and semantics of data schemata. Section 4 and 5 introduce the type inference decision tables for the internal type system. Section 6 describes the static query evaluation performed to check the type safety and to output possible type errors. Section 7 contains an example of the static evaluation. In section 8 we elaborated possible repairs of the query which can be applied during the static evaluation. In section 9 we discuss how to recover after a type error is detected, so that we can find more than one type error during one type check pass. Section 10 shortly discusses basic issues of an external type system. Section 11 describes the type system in our prototype implementation ODRA. Section 12 concludes.

2. Stack-Based Approach (SBA)

In SBA [Subi95a, Subi95b, Subi04] a query language is considered a kind of a programming language. Thus, the semantics of queries is based on mechanisms well known from programming languages like the environment stack. SBA extends this concept for the case of query operators, such as selection, projection/navigation, join, quantifiers and others. SBA is able to determine precisely the operational semantics (abstract implementation) of query languages, including relationships with object-oriented concepts, embedding

queries into imperative constructs, and embedding queries into programming abstractions: procedures, functional procedures, views, methods, modules, etc.

SBA respects the naming-scoping-binding principle, which means that each name occurring in a query is bound to the appropriate run-time entity (an object, an attribute, a method a parameter, etc.) according to the name scope. The principle is supported by means of the environment stack, extended (in comparison to programming languages) to cover database collections and all typical query operators occurring e.g. in SQL and OQL. Due to stack-based semantics we achieve full orthogonality and compositionality of query operators. The stack also supports recursion and parameters: all functions, procedures, methods and views defined by SBA can be recursive by definition. Rigorous formal semantics implied by SBA creates a very high potential for query optimization. SBA and its query language SBQL has several implementations: for the LOQIS system, for XML DOM model, for the European project ICONS, for Objectivity/DB and for the currently developed object-oriented platform ODRA.

2.1 Object Store Model

SBA is defined for the general data store. In this report we explain our ideas for two models called M0 and M1 in [Sub04]. M0 is a simple data store model where objects can contain other objects with no limitations on the level of the nesting of objects. There are also relationships between objects. M1 is slightly more complex. It describes object-oriented data stores with classes and inheritance. SBA is extended in [Sub04] to comply with other notions like dynamic object roles and encapsulation.

M0 is based on the principles of object relativism and internal identification. In the store each object has the following properties:

- Internal identifier (OID) that neither can be directly written in queries nor printed,
- External name (introduced by a programmer or the designer of the database) which is used to access the object from an application,
- Content, which can be a value, a link, or a set of objects.

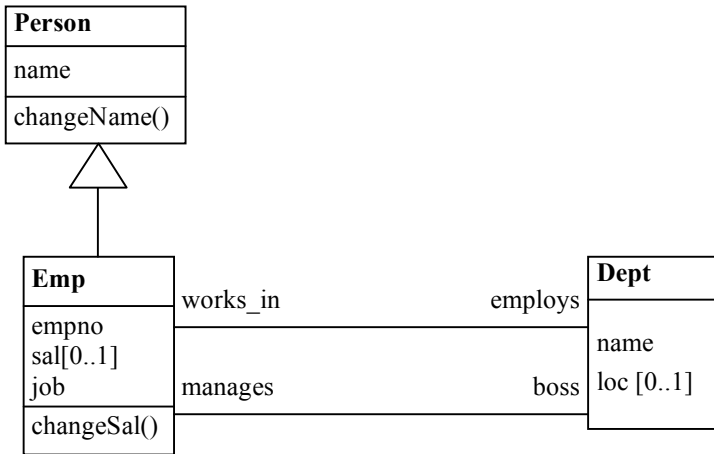


Fig. 1. Example database schema in UML

Let I be the set of internal identifiers, N be the set of external names, and V be the set of atomic values, e.g. strings, integers, etc. Formally, objects in M_0 are triples defined below ($i_1, i_2 \in I$, $n \in N$, and $v \in V$).

- An atomic objects has the form $\langle i_1, n, v \rangle$: an object identified by i_1 has external name n and stores an atomic value v .
- A link object has the form $\langle i_1, n, i_2 \rangle$, where an object identified by i_1 points to the object identified by i_2 .
- A complex objects has the form $\langle i_1, n, S \rangle$, where S is a set of objects.

Note that this definition is recursive and models nested objects with an arbitrary number of hierarchy levels. An object store consists of:

- The structure of objects defined above.
- Internal identifiers of *root objects* (they are accessible from outside, i.e. they are starting points for querying).
- Constraints (e.g. the uniqueness of the internal identifiers, referential integrities, etc.).

M_1 is an extension of M_0 which introduces classes and inheritance. An M_1 data store contains following additional entities:

- The structure of classes.
- The relation *CC* (Class-Class) which defines inheritance relationships between classes.
- The relation *OC* (Object-Class) which defines the membership of objects in classes.

Example database schema and the corresponding data store are depicted respectively in

Fig. 1 and **Fig. 2**. The store contains the information on persons, departments and their employees. Each person has a name. Each employee is a person and has personal identification number, salary and job. Each department has the attribute *name*. The employees work in departments. Some employees may manage departments.

<p>Objects:</p> <p>$\langle i_1, Person, \{ \langle i_2, name, "Black" \rangle \} \rangle$ $\langle i_3, Person, \{ \langle i_4, name, "Johnson" \rangle \} \rangle$ $\langle i_5, Person, \{ \langle i_6, name, "Anderson" \rangle \} \rangle$ $\langle i_7, Emp, \{ \langle i_8, empno, 123 \rangle, \langle i_9, name, "Smith" \rangle,$ $\quad \langle i_{10}, sal, 3500 \rangle, \langle i_{11}, job, "consultant" \rangle, \langle i_{12}, works_in, i_{26} \rangle \} \rangle$ $\langle i_{13}, Emp, \{ \langle i_{14}, empno, 234 \rangle, \langle i_{15}, name, "White" \rangle, \langle i_{16}, sal, 2900 \rangle,$ $\quad \langle i_{17}, job, "salesman" \rangle, \langle i_{18}, works_in, i_{32} \rangle \} \rangle$ $\langle i_{19}, Emp, \{ \langle i_{20}, empno, 456 \rangle, \langle i_{21}, name, "Blake" \rangle, \langle i_{22}, sal, 3200 \rangle,$ $\quad \langle i_{23}, job, "manager" \rangle, \langle i_{24}, works_in, i_{26} \rangle, \langle i_{25}, manages, i_{27} \rangle \} \rangle$ $\langle i_{26}, Dept, \{ \langle i_{27}, name, "IT" \rangle, \langle i_{28}, employs, i_1 \rangle, \langle i_{29}, employs, i_{13} \rangle,$ $\quad \langle i_{31}, boss, i_{19} \rangle \} \rangle$ $\langle i_{32}, Dept, \{ \langle i_{33}, name, "Sales" \rangle, \langle i_{34}, loc, "Paris" \rangle, \langle i_{35}, employs, i_{13} \rangle \} \rangle$ $\langle i_{36}, ClassPerson, \{ \langle i_{37}, changeName, (code\ of\ the\ method\ changeName) \rangle \} \rangle$ $\langle i_{38}, ClassEmp, \{ \langle i_{39}, changeSal, (code\ of\ the\ method\ changeSal) \rangle \} \rangle$</p> <p>ROOT = $\{ i_1, i_3, i_5, i_7, i_{13}, i_{19}, i_{26}, i_{32} \}$</p> <p>CC = $\{ \langle i_{38}, i_{36} \rangle \}$</p> <p>OC = $\{ \langle i_1, i_{36} \rangle, \langle i_3, i_{36} \rangle, \langle i_5, i_{36} \rangle, \langle i_7, i_{38} \rangle, \langle i_{13}, i_{38} \rangle, \langle i_{19}, i_{38} \rangle \}$</p>

Fig. 2. Example of SBA data store

2.2 Environment Stack and Name Binding

The basis of SBA is the environment stack (ES). It is one of the most basic auxiliary data structures in programming languages. It supports the abstraction principle, which allows the programmer to consider the currently written piece of code to be independent of the context of its possible uses. In SBA the environment stack consists of sections which contain sets of entities called binders. A *binder* is a construct which binds a name with a run-time object. Formally, it is a pair (n, i) (further written as $n(i)$) where n is an external name ($n \in \mathbb{N}$) and i is the reference to an object ($i \in I$). In the following the binder concept will be extended to $n(x)$, where x is any result returned by a query.

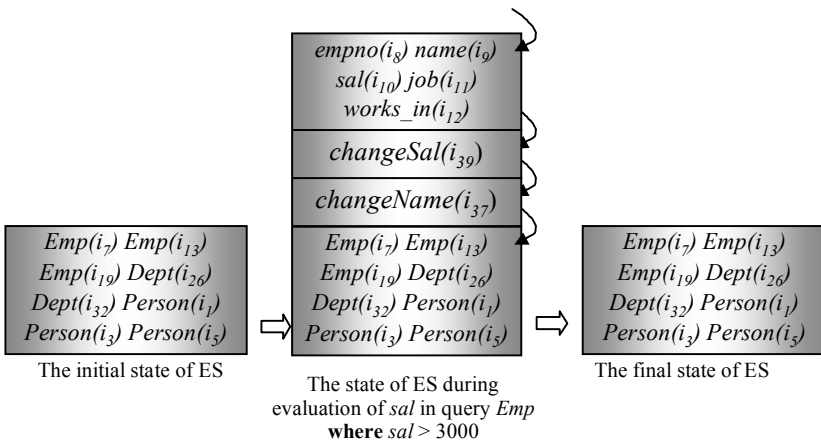


Fig. 3. Example of ES states during evaluation of a query

SBA respects the naming-scoping-binding principle, which means that each name occurring in a query is bound to a run-time entity (an object, an attribute, a method a parameter, etc.) according to the scope of its name. The principle is supported by means of the environment stack. The process of name

binding follows the “search for the top rule” and it returns the second component of a binder with the given name which is the closest to the top of ES and is visible within the scope of the name. Because names associated with binders are not unique, the binding can return multiple identifiers. In this way we deal with collections.

Some states of ES during evaluation of the query *Emp where sal > 3000* are depicted in Fig. 3. In this figure solid arrows indicate the order of name binding. At the beginning of the query evaluation ES consists only of the base section which contains binders to all root objects of the data store. There can be other base sections containing binders to the variables of the environment of the system, properties of the current session etc. If a query has no side effects, the state of the stack after the evaluation of the query is the same as before it. During query evaluation the stack grows and shrinks according to the nesting of the query.

New sections pushed onto ES are constructed by means of function *nested*. The function works in the following way:

- For the identifier of a link the function returns the set with the binder of the object the link points to.
- For a binder the function returns the set with this binder.
- For a complex object the function returns the set of binders to the sub-objects (attributes) of the object.
- For structures, $nested(\mathbf{struct}\{x_1, x_2, \dots\}) = nested(x_1) \cup nested(x_2) \cup \dots$
- For other arguments the result of *nested* is empty.

If an object O is a member of a class C_1 which has superclasses C_2, C_3 , then the environment stack is pushed with four sections. They respectively contain (from the bottom to the top): $nested(C_3)$, $nested(C_2)$, $nested(C_1)$ and $nested(O)$.

Fig. 3 contains the result of *nested* for i_7 . It is the set: $empno(i_8), name(i_9), sal(i_{10}), job(i_{11}), works_in(i_{12})$). This figure also contains the sections for the classes of i_7 . These sections are: $nested(i_{36}) = \{ changeName(i_{37}) \}$ and $nested(i_{38}) = \{ changeSal(i_{39}) \}$.

2.3 Stack Based Query Language (SBQL)

In this section we present the basis of the Stack Based Query Language (SBQL) [Subi85, Subi87, Subi93, Subi95a, Subi95b, Subi04]. The language has been first implemented in the LOQIS system [Subi90, Subi91] then in several other systems. SBQL is based on the principle of compositionality, which means that more complex queries can be built of simpler ones. The description of the syntax of queries in SBQL follows.

- A name or a literal is a query; e.g. *name*, *Emp*, 2, “Black”.
- σq is a query, where σ is a unary operator and q is a query; e.g. *sum(sal)*, *sin(x)*.
- $q_1 \tau q_2$ is a query, where q_1 and q_2 are queries and τ is a binary operator; e.g. *2+2*, *Emp.sal*, *Emp where (sal > 2000)*.

Therefore, in SBQL we can construct complex queries by composing them from simpler ones using unary and binary operators. With the exception of typing constrains the operators are orthogonal.

In SBA we divide operators into two main groups: algebraic and non-algebraic. In the following we describe the difference.

Algebraic Operators. The operator is algebraic, if its evaluation does not require the use of the environment stack. The evaluation of the query $q_1 \Delta q_2$, where Δ is algebraic operator, looks as follows. Queries q_1 and q_2 are evaluated independently and the final result is a combination of these partial results depending on the semantics of operator Δ . Note that the key property of algebraic operators is that the order of evaluation of q_1 and q_2 does not matter.

Algebraic operators include string comparisons, boolean and numerical operators, aggregate functions, operators on sets, bags and sequences (e.g. the union), comparisons, structure constructors, etc.

Non-algebraic Operators. If the query $q_1 \theta q_2$ involves a non-algebraic operator θ , then q_2 is evaluated in the context determined by q_1 . Thus, the order of evaluation of subqueries q_1 and q_2 is significant. The query $q_1 \theta q_2$ is evaluated as follows. The subquery q_2 is evaluated for each element r of the collection returned by q_1 . Before each such evaluation ES is augmented with a new scope determined by *nested(r)*. After the evaluation the stack is popped to the previous state. A partial result of the evaluation is a combination of r and

the result returned by q_2 for this value; the method of the combination depends on θ . Finally, these partial results are merged into the final result depending on the semantics of operator θ .

Non-algebraic operators include selection (where), projection/navigation (the dot), dependent join, quantifiers (for all, for any), transitive closures, etc.

Examples of Queries in SBQL. Here we present examples of queries in SBQL addressing the example database shown in Fig.1 and Fig. 2.

- Get employees earning more than 30000:
Emp where sal > 30000
- Get the name of the boss of IT department:
(Dept where name = "IT").boss.Emp.name
- Get the departments that have employees that have salary higher than the boss of the department:
(Dept as d) where ((d.employs.Emp as e) \exists (e.salary > d.boss.Emp.sal))

3. Metabase Graph

A type system for a database is necessary to express the information on the conceptual and logical structure of the data store. The information is commonly referred to as *database schema*. Although basically the query engine is independent of the database schema, we need it to reason statically about type correctness of queries and programs. We also need it for other reasons, in particular, to enforce type constraints inside the object store, to resolve some binding ambiguities, to reason about ellipses, dereferences and coercions occurring in queries, and to optimize queries. A schema statically models and reflects the data store and itself it looks like a data store. This idea is similar to DataGuides of Lore [Gold97].

A schema has two forms: *external* (as presented for programmers) and *internal* (as used by the type inference engine). In this report we are mostly interested in an internal schema, which will be called *metabase*. A metabase represents schema information in a form of well-structured data (a graph), allowing for querying and for modification. In general, a metabase for complex object-oriented models can be quite sophisticated (see [Habe02, Habe03] presenting

major issues related to the topic). To simplify the presentation, we skip some metabase features, in particular *typedef* statements (we consider them macro-definitions, hence their presence is not obligatory), the possibility to represent exclusive type variants (unions in C/C++ terminology), associations between function/method signatures (types of parameters and results) and nodes of the metabase graph, subdivision of class/object properties into *public* and *private*, and several other options. We hope that including such features into the presented metabase will be straightforward and clear for the reader.

A node of a metabase graph represents some data entity (entities) of the object store: object(s), attribute(s), link(s), procedure, class, method, view, and perhaps others. Each node itself is a complex object having its non-printable identifier (we call it *static identifier*) which uniquely identifies the node. In this report we assume a meta-language convention that the node describing entities named n has the identifier denoted i_n . The convention has no meaning for semantics and implementation of the static typing mechanism.

For simplification in this paper we “stick” representation of a complex object with representantation of its direct class. This does not lead to ambiguity, because on the run-time environment stack the section with binders to a complex object has always a twin, neighbouring section with binders to properties of its direct class. Such sticking metabase nodes implies similar sticking of these two sections on the static environment stack, thus simplifies the metabase and implementation.

As a complex object, a metabase graph node contains several attributes. In this report we distinguish the following ones:

- **name.** It stores a name $n \in N$ of objects described by the node.
- **kind.** It stores information on kind of objects described by the node (atomic, complex, link, procedure, method, class, etc.). For simplification in this paper we make no distinction between representation of a complex object and a class
- **type.** An atomic type of objects described by the node. The attribute is not obligatory for nodes describing complex objects, because their types are composed from types of other entities in the metabase graph. The same concerns nodes representing link objects.
- **card.** A cardinality of objects described by the node. Cardinalities will be written in the form $i..j$, where i is the minimal number of described objects

and j is the maximal number of described objects. If the number of described objects is unlimited, j has the value $*$. Cardinalities $0..0$ and $1..1$ will be written 0 and 1 , and $0..*$ will be written as $*$. Cardinality 0 will be used as default when it is irrelevant, e.g. for abstract class nodes or for procedures returning *void*.

There could be other attributes of the node, in particular, mutability (update-ability, insert-ability, delete-ability), a collection kind (bag, sequence, array, etc.), type name (if one would like to assume type equivalence based on type names, as e.g. in Pascal), and perhaps others. For simplicity of presentation in this report we omit them, but they can be easily introduced in implementation.

Let IS be the set of static identifiers. Besides the nodes, the metabase graph consists of four elements:

- The set $static_roots \subseteq IS$ containing static identifiers of nodes representing data store objects which are starting points for querying (roots). On diagrams root nodes will be denoted by a thicker line around a box.
- The binary relation $static_CC \subseteq IS \times IS$ which defines the inheritance between classes. It connects nodes representing a subclass with a node representing its direct superclass. On diagrams elements of the relation will be denoted as arrows with white triangle ends pointing a superclass (as in UML).
- The binary relation $static_OC \subseteq IS \times IS$ which defines the ownership of objects in other objects, in particular, membership of objects in classes. On diagrams elements of the relation will be denoted as arrows with ends pointing to member nodes. Because function *nested* in fact unifies complex objects and links, $static_OC$ will also contain information on the relationship between nodes representing links (associations) with nodes representing objects that these links point to.

Further properties of a metabase may require other elements for describing it, e.g. relationships between parameters of procedures/methods and metabase graph nodes. An example schema for the data store from Fig. 2 is presented in fig 4. Note that assuming some predefined (the same) name for all nodes and predefined names for $static_CC$ and $static_OC$ relationships the graph metabase can be easily represented as an object store built according to the M0 store model. In this way the metabase can be queried through SBQL.

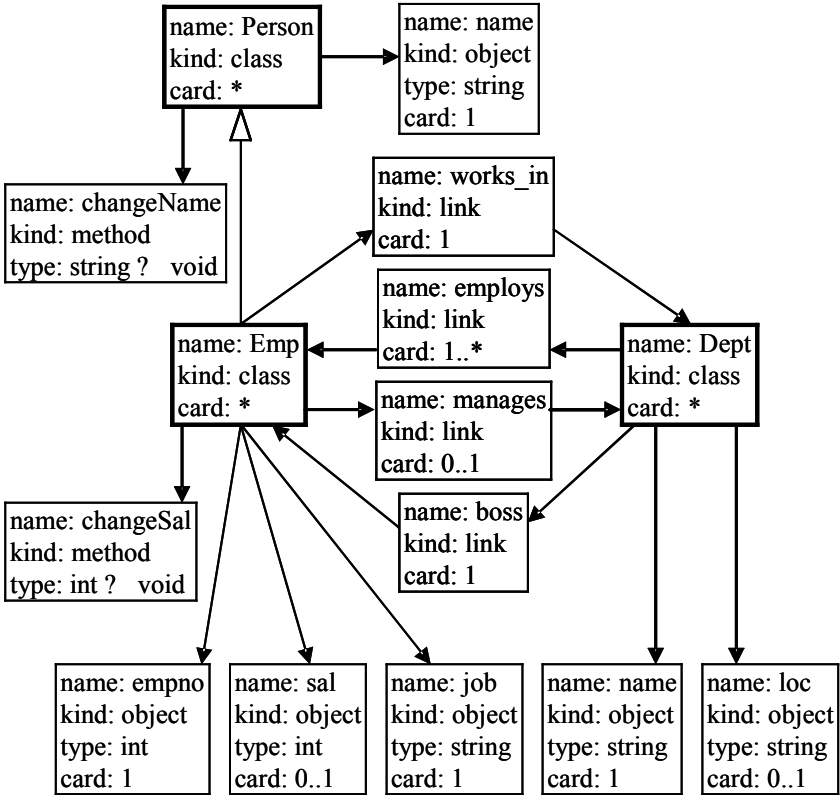


Fig. 4. A metabase graph

4. Internal Types of Values

In SBA objects always reside in the store and no query can return objects, but references to objects. The query engine processes *values* which can also be the references. The purpose of introducing static stacks is precise simulation of the actual computation. The static stacks contain so-called *signatures* which are description of values used during the actual computation. The set of signatures S is the smallest set that satisfies the following conditions:

- Names of atomic types (e.g. int, float, string, date, etc.) belong to S .

- All static identifiers of the metabase graph nodes (e.g. i_{Emp} and i_{Dept}) belong to S . Identifiers of graph nodes represent types defined in the database schema. Static identifiers are signatures of references to store objects.
- If x belongs to S and n belongs to \mathbb{N} , then the pair $n(x)$ belongs to S . Such a pair will be called *static binder*. It is a signature of a binder known from run-time stacks.
- If x_1, x_2, \dots, x_n belong to S , then **struct** $\{x_1, x_2, \dots, x_n\}$, **bag** $\{x_1, x_2, \dots, x_n\}$, **sequence** $\{x_1, x_2, \dots, x_n\}$ and **variant** $\{x_1, x_2, \dots, x_n\}$ belong to S . Signatures **struct**, **bag** and **sequence** reflect the structures, bags and sequences processed by the query engine. A **variant** is used when the type of an expression cannot be determined at compile time. It can happen in the case of unions of expressions of different types. For example, the type of the query $Emp \cup Dept$ is **variant** $\{i_{Emp}, i_{Dept}\}$.

Note that the definition is recursive. A function *static_nested* (a compile-time counterpart of the run-time function *nested*) is used during the static type check of queries when a non-algebraic operator is processed. The function works in the following way:

1. For a static binder $n(x)$ *static_nested* returns the set $\{n(x)\}$.
2. For a static identifier i_n of link objects, described in the schema by the static link object pointing to the object named m , *static_nested* returns the set $\{m(i_m)\}$. For instance, $static_nested(i_{boss}) = \{Emp(i_{Emp})\}$.
3. For a static identifier i_n of complex objects (described in the metabase graph by kind:class) *static_nested* returns the set with static binders to all static subobjects (for subobjects which can occur at most once) or bags of such binders (for subobjects which can occur more than once). More formally:

$$static_nested(i_n) = \bigcup_{\{i : (i_n, i) \in static_OC\}} (\{m(i) : i.card = 1\} \cup \{\mathbf{bag}\{m(i)\} : i.card \neq 1\})$$

4. For a structure *static_nested* returns a specific struct-union (\cup_s) of values of *static_nested* for all components of the structure. It works like ordinary set union, but if an item is a member of both arguments, then bag of it belongs to this union. The struct-union converts sequences to bags, because the result of a union is not ordered. More

formally, $S_1 \cup_s S_2$ is the smallest set which fulfils the following conditions:

- a. If $E \in S_1$ and E is neither **bag** $\{\cdot\}$ nor **sequence** $\{\cdot\}$ and none of E , **bag** $\{E\}$ and **sequence** $\{E\}$ belongs to S_2 , then $E \in S_1 \cup_s S_2$.
- b. If $E \in S_2$ and E is neither **bag** $\{\cdot\}$ nor **sequence** $\{\cdot\}$ and none of E , **bag** $\{E\}$ and **sequence** $\{E\}$ belongs to S_1 , then $E \in S_1 \cup_s S_2$.
- c. If $E \in S_1$, $E \in S_2$ and E is neither **bag** $\{\cdot\}$ nor **sequence** $\{\cdot\}$, then **bag** $\{E\} \in S_1 \cup_s S_2$.
- d. If **bag** $\{E\} \in S_1$ or **bag** $\{E\} \in S_2$ or **sequence** $\{E\} \in S_1$ or **sequence** $\{E\} \in S_2$, then **bag** $\{E\} \in S_1 \cup_s S_2$.

Now we can define *static_nested* for **struct**:

$$\text{static_nested}(\mathbf{struct}\{x_1, x_2, \dots, x_n\}) = \text{static_nested}(x_1) \cup_s \text{static_nested}(x_2) \cup_s \dots \cup_s \text{static_nested}(x_n)$$

5. In this paper we assume that for **variant** *static_nested* returns the same result as for **struct**. In further research we will consider a case when a variant requires splitting the checking processes to particular cases of the variant. The splitting will allow alternative type checking, which will be positive if at least one case of a variant returns the positive typecheck. Our assumption on equivalence of **variant** and **struct** for the function *static_nested* is approximation, which may cause lack of precision in discovering some type errors.
6. For all other arguments *static_nested* returns the empty set.

Note that optional subobjects of complex objects are treated as bags of subobjects (3). The reason is that we want to avoid false bindings. Let us consider the following query which finds all departments at the same location as Sales:

Dept where loc = (Dept where name = "Sales").loc

The subobject *loc* is optional. Thus, if it is not specified for Sales, the name *loc* will be wrongly bound in the section of the stack opened by the first **where**. This definitely does not conform to the intent of the author of the query. Therefore, during the static type check we treat the subobject *loc* as a bag. This allows enforcing the name *loc* to be bound in the proper section of the stack.

We simply enhance the query with the run-time check whether the subobject *loc* is present (see section 8.3). If it is not, an exception will be raised.

5. Type Inference

5.1 Arithmetic and String Operators (algebraic)

During the static type check we use rules to infer the type of complex expressions from types of their subexpressions. The following type inference decision table contains the rules for arithmetic and string operators.

LHS	RHS	Operator	Result type
integer	integer	+, -, *	integer
integer	integer	/	float
integer	integer	=, <, >, !=	boolean
float	float	+, -, *, /	float
float	float	=, <, >, !=	boolean
boolean	boolean	=, <, >, !=	boolean
string	string	+	string
string	string	=, <, >, !=	boolean
boolean	boolean	AND, OR, NOT	boolean

All other cases result in the type error. For example one cannot divide a number by a string. The above table will be further extended by introducing implicit type coercions.

5.2 Union, Intersection and Difference (algebraic)

We consider three bag operators known from the set theory. These are the union (\cup), the intersection (\cap) and the difference (**except**). Before the application of such an operator both its arguments are coerced to bags. Let signature T be the type of an argument. First we coerce it to a bag.

- If T is **bag** $\{T_E\}$ for some T_E , then no coercion is necessary.
- If T is **sequence** $\{T_E\}$ for some T_E , then it is coerced to **bag** $\{T_E\}$.
- If T is neither **bag** $\{T_E\}$ nor **sequence** $\{T_E\}$ for any T_E , then it is coerced to **bag** $\{T\}$.

Let us consider the following queries:

$$q_L \cup q_R$$

$$q_L \cap q_R$$

$$q_L \text{ except } q_R$$

Let \mathcal{L} be the set of possible types of items of q_L i.e.:

$$\mathcal{L} = \{L_1\} \quad \text{if the type of } q_L \text{ is } \mathbf{bag}\{L_1\}$$

$$\mathcal{L} = \{L_1, L_2, \dots, L_m\} \quad \text{if the type of } q_L \text{ is } \mathbf{bag}\{\mathbf{variant}\{L_1, L_2, \dots, L_m\}\}$$

Let \mathcal{R} be the set of possible types of items of q_L i.e.:

$$\mathcal{R} = \{R_1\} \quad \text{if the type of } q_L \text{ is } \mathbf{bag}\{R_1\}$$

$$\mathcal{R} = \{R_1, R_2, \dots, R_n\} \quad \text{if the type of } q_L \text{ is } \mathbf{bag}\{\mathbf{variant}\{R_1, R_2, \dots, R_n\}\}$$

Union $q_L \cup q_R$

If the union of sets \mathcal{L} and \mathcal{R} has exactly one element U , then the type of this union query is $\mathbf{bag}\{U\}$.

If the union of sets \mathcal{L} and \mathcal{R} has more than one element, then the type of this union query is $\mathbf{bag}\{\mathbf{variant}\{U_1, U_2, \dots, U_k\}\}$ where $\{U_1, U_2, \dots, U_k\}$ is the union of the sets \mathcal{L} and \mathcal{R} .

Intersection $q_L \cap q_R$

If the intersection of sets \mathcal{L} and \mathcal{R} has exactly one element U , then the type of this intersection query is $\mathbf{bag}\{U\}$.

If the intersection of sets \mathcal{L} and \mathcal{R} has more than one element, then the type of this intersection query is $\mathbf{bag}\{\mathbf{variant}\{I_1, I_2, \dots, I_k\}\}$ where $\{I_1, I_2, \dots, I_k\}$ is the intersection of the sets \mathcal{L} and \mathcal{R} .

Difference $q_L \text{ except } q_R$

The type of this query is the same as the type of q_L .

5.3 Structure Constructor (algebraic)

The structure constructor has the syntax $\mathbf{struct}(q_1, q_2, \dots, q_k)$ (the keyword \mathbf{struct} can be omitted). First the results of the queries q_1, q_2, \dots, q_k are coerced to bags in the same way as arguments of union, intersection and difference (cf. the coercion to bag in section 5.2).

Let us assume that after the coercion the types of the queries q_1, q_2, \dots, q_k are the signatures $\mathbf{bag}\{T_1\}, \mathbf{bag}\{T_2\}, \dots, \mathbf{bag}\{T_k\}$ respectively. Then the type of the above structure is the following signature:

$$\mathbf{bag}\{\mathbf{struct}\{T_1, T_2, \dots, T_k\}\}$$

Since a **struct** nested directly inside another **struct** is equivalent to the flattened **struct** (see the definition of the function *static_nested* in section 4), this signature is also flattened. If for any i the signature T_i is $\mathbf{struct}\{R_1, R_2, \dots, R_h\}$ then we flatten the signature $\mathbf{bag}\{\mathbf{struct}\{T_1, T_2, \dots, T_k\}\}$ to obtain:

$$\mathbf{bag}\{\mathbf{struct}\{T_1, T_2, \dots, T_{i-1}, R_1, R_2, \dots, R_h, T_{i+1}, \dots, T_k\}\}$$

We repeat this operation until we obtain no **struct** nested directly in another **struct**.

5.4 Auxiliary Names (algebraic)

Auxiliary names can be introduced in two following ways:

$$q \text{ group as } n$$

$$q \text{ as } n$$

The first way consists in assigning the name n to the whole result of the query q . If the result of the query q is a bag or a sequence, the operator **as** adds the name n to each element of the collection. The following table summarizes the type inference for auxiliary naming:

The type of q	The type of q group as n	The type of q as n
$\mathbf{bag}\{T\}$	$n(\mathbf{bag}\{T\})$	$\mathbf{bag}\{n(T)\}$
$\mathbf{sequence}\{T\}$	$n(\mathbf{sequence}\{T\})$	$\mathbf{sequence}\{n(T)\}$
T	$n(T)$	$n(T)$

The last row of this table applies to any signature T which is neither bag nor sequence.

5.5 Navigation (non-algebraic)

The navigation (dot) is the first non-algebraic operator considered in this section. The decision tables for all the non-algebraic operators are generalized, i.e. each row of the table represents some collection (usually infinite) of real cases. Such generalized decision tables we will call *meta-rules*.

Consider the query:

$q_L \cdot q_R$

The type of this query depends on the types of the queries q_L and q_R . We will consider three cases for each of the two subqueries:

- the type of a subquery is the signature **bag** $\{T\}$ for some T ,
- the type of a subquery is the signature **sequence** $\{T\}$ for some T ,
- the type of a subquery is neither **bag** $\{T\}$ nor **sequence** $\{T\}$ for any signature T . In all inference tables we will use the symbol E to denote the type of such a subquery (E stands for *element*).

The type of q_L	The type of q_R	The type of $q_L \cdot q_R$
E_1	E_2	E_2
E_1	bag $\{T_2\}$	bag $\{T_2\}$
E_1	sequence $\{T_2\}$	sequence $\{T_2\}$
sequence $\{T_1\}$	E_2	sequence $\{E_2\}$
sequence $\{T_1\}$	bag $\{T_2\}$	<i>type error</i>
sequence $\{T_1\}$	sequence $\{T_2\}$	sequence $\{T_2\}$
bag $\{T_1\}$	E_2	bag $\{E_2\}$
bag $\{T_1\}$	bag $\{T_2\}$	bag $\{T_2\}$
bag $\{T_1\}$	sequence $\{T_2\}$	<i>type error</i>

5.6 Dependent join (non-algebraic)

The dependent join is similar to the navigation. However, while the navigation discards the result of the left subquery, the dependent join constructs the result out of the results of both subqueries. We will consider the following query:

$q_L \text{ join } q_R$

The type of this query depends on the types of the queries q_L and q_R . We will consider three cases for each of the two subqueries, just like in section 5.5. Again E denotes a signature which is neither **bag** $\{T\}$ nor **sequence** $\{T\}$ for any signature T .

The type of q_L	The type of q_R	The type of $q_L \text{ join } q_R$
E_1	E_2	struct $\{E_1, E_2\}$
E_1	bag $\{T_2\}$	bag struct $\{E_1, T_2\}$
E_1	sequence $\{T_2\}$	sequence struct $\{E_1, T_2\}$
sequence $\{T_1\}$	E_2	sequence struct $\{T_1, E_2\}$

<code>sequence{T1}</code>	<code>bag{T2}</code>	<i>type error</i>
<code>sequence{T1}</code>	<code>sequence{T2}</code>	<code>sequence{struct{T1, T2}}</code>
<code>bag{T1}</code>	E_2	<code>bag{struct{T1, E2}}</code>
<code>bag{T1}</code>	<code>bag{T2}</code>	<code>bag{struct{T1, T2}}</code>
<code>bag{T1}</code>	<code>sequence{T2}</code>	<i>type error</i>

5.7 Selection (non-algebraic)

A selection is much simpler than previous operators, because the right-hand query must be of the type boolean. We will consider the following query:

q_L **where** q_R

If the type of q_R is not boolean, the type error will occur. Let us assume that the type of q_R is boolean. The type of the selection depends on the type of q_L . Again E denotes a signature which is neither `bag{T}` nor `sequence{T}` for any signature T .

The type of q_L	The type of q_L where q_R
E	<code>bag{E}</code>
<code>sequence{T}</code>	<code>sequence{T}</code>
<code>bag{T}</code>	<code>bag{T}</code>

Note that if the type of q_L is E , then the type of the selection is the signature `bag{E}`, because the only element needs not satisfy the query q_R and thus the result may be the empty bag.

5.8 Quantifiers (non-algebraic)

A quantifier is even simpler than the selection, because its type is always boolean. We will consider the following queries:

$\exists q_1(q_2)$

$\forall q_1(q_2)$

If the type of q_R is not boolean, the type error will occur. Otherwise the type of both queries is boolean.

5.9 Transitive closure (non-algebraic)

SBQL defines four operators of the transitive closure:

q_L **close by** q_R

q_L **close distinct by** q_R

q_L leaves by q_R

q_L leaves distinct by q_R

First the results of the queries q_L and q_R are coerced to bags in the same way as arguments of union, intersection and difference (cf. the coercion to bag in section 5.2). If after this conversion the types of q_L and q_R are the same signature, this signature is the type of the transitive closure. Otherwise (after the conversion the types of q_L and q_R are different signatures), the type error occurs.

6. Static Type Checking

The general architecture of the type checker is presented on Fig.5. Shaded shapes are program modules, while dashed lines surround data structures which are used and created by the modules. The *query parser* takes the *query* as a text supplied by a *client* and compiles it to produce an *abstract syntax tree* of the query. This syntax tree is analysed, decorated and sometimes modified by the *type checker*. If the type checking is successful (the query is correct), the query is executed by the *query engine*. The query engine operates on two stacks (ENVS and QRES, see Section 2) and on the data store.

Analogously, the type checker which is to simulate the execution of the query operates on corresponding static structures (the static environment stack S_ENVS, the static result stack S_QRES and the metabase).

The type checker uses the information known during the parsing and does not retrieve any information from the data store. The static stacks contain, in particular, signatures of objects from the data store. The query parser processes the signatures exactly in the same way as the query engine will later process the concrete object from the data store.

We present the procedure *static_type_check* which is the heart of the type checker and operates on the syntactic tree of a query, both static stacks and the metabase. This function accomplishes the following actions:

- Checks the type correctness of the syntactic tree of a query by simulating the execution of this query on the static stacks S_ENVS and S_QRES.
- Generates messages on type errors.
- Augments the syntactic tree in order to resolve ellipses.

- Augments the syntactic tree with automatic dereferences and coercions.
- Augments the syntactic tree with dynamic type checks, if the type correctness cannot be asserted statically.
- Possibly modifies the syntactic tree and the static stacks in order to restore the process after a type error has been encountered. These modifications are driven by rules which define most probable result types in certain cases. These modifications allow detecting more than one error during one type check pass.

Changes of the syntactic tree will be marked by the label **AUGMENT** in the text of the procedure below. We will discuss these augments in more detail in section 7.

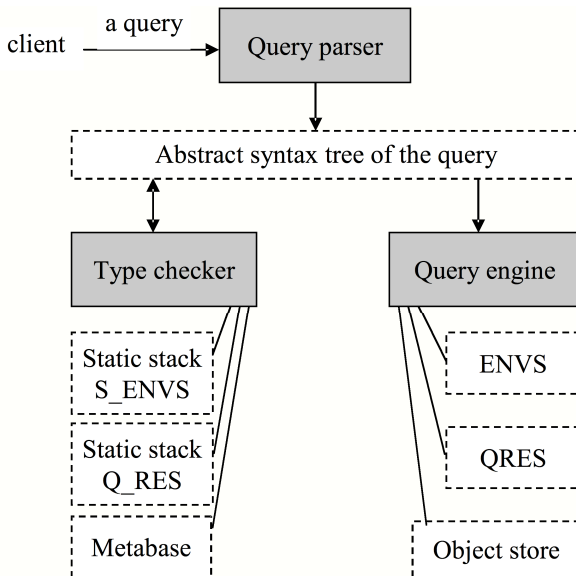


Fig. 5 The general architecture of the type checker

Exceptional changes of the static stacks will be marked by the label **RESTORATION** in the text of the procedure below. We will discuss these changes in more detail in section 9.

Before the function *static_type_check* is run, the stack S_ENVS must contain the static base section which consists of the static binders to all static root identifiers (cf. section 3). For each static root binder i_n defined in the schema, the base section of S_ENVS will contain the signature **bag** $\{n(i_n)\}$. For the schema presented in fig. 4 the base section of S_ENVS will consist of **bag** $\{Person(i_{Person})\}$, **bag** $\{Emp(i_{Emp})\}$ and **bag** $\{Dept(i_{Dept})\}$.

```

procedure static_type_check(query) {
  case query is  $l$  {                                     //  $l$  is a literal value
    Find the most appropriate signature  $t$  for the literal value  $l$ , e.g.
    a. If  $l$  has the form of "...", then  $t = \text{string}$ .
    b. If  $l$  is an integer, then  $t = \text{integer}$ .
    c. If  $l$  is a floating point number, then  $t = \text{float}$ .
    d. If  $l$  is either true or false, then  $t = \text{boolean}$ .
    push(S_QRES,  $t$ );
  } // case query is  $l$ 

  case query is  $n$  {                                     //  $n$  is a name
    Starting from the top section of S_ENVS search for the first signature of
    the form  $n(s)$ , bag $\{n(s)\}$  or sequence $\{n(s)\}$ .
    if found then {
      Let  $t$  be  $s$ , bag $\{s\}$  or sequence $\{s\}$  respectively.
    } else {
      Try augmenting the query to resolve an ellipsis (AUGMENT 1).
      Let the signature  $t$  be the type of the query  $n$  after this augmentation.
      if the augmentation fails then {
        Issue a message on the type error.
        Restore the process by finding the most probable signature  $t$  in
        this position (RESTORATION 1).
      }
    }
    push(S_QRES,  $t$ )
  } // case query is  $n$ 

  case query is  $\Delta q$  {                                 //  $\Delta$  is unary
    static_type_check ( $q$ );
  }

```

```

u := pop(S_QRES);
Find the type inference rule for  $\Delta$  (cf. section 5).
if u fits this inference rule then {
    Let the signature t be the type of  $\Delta q$  indicated by this inference rule.
} else {
    Try augmenting the query by adding a dereference (AUGMENT 2) or
    a coercion (AUGMENT 3) between  $\Delta$  and q.
    Let the signature t be the type of the query  $\Delta q$  after this augmentation.
    if the augmentations fail then {
        Issue a message on the type error.
        Restore the process by finding the most probable signature t in
        this position (RESTORATION 2).
    }
}
push(S_QRES, t);
} // case query is  $\Delta q$ 

case query is  $q_1 \Delta q_2$  { //  $\Delta$  is algebraic
    static_type_check (q1);
    static_type_check (q2);
    u2 := pop(S_QRES);
    u1 := pop(S_QRES);
    Find the type inference rule for  $\Delta$  (cf. section 5).
    if u1 and u2 fit this inference rule then {
        Let the signature t be the type of  $q_1 \Delta q_2$  indicated by this inference rule.
    } else {
        Try augmenting the query by adding a dereference (AUGMENT 2) or
        a coercion (AUGMENT 3) between  $\Delta$  and q1 and/or between  $\Delta$ 
        and q2.
        Let the signature t be the type of  $q_1 \Delta q_2$  after this augmentation.
        if the augmentations fail then {
            Issue a message on the type error.
            Try to restore the process by finding the most probable signature t
            in this position (RESTORATION 2).
        }
    }
}
push(S_QRES, t);
} // case query is  $q_1 \Delta q_2$ 

```

```

case query is  $q_1\theta q_2$  {
    //  $\theta$  is non-algebraic
    static_type_check( $q_1$ );
     $u_1 := \text{top}(\text{S\_QRES})$ ;
    Find the type inference rule for  $\theta$  (cf. section 5).
    if  $u_1$  does not fit this inference rule then {
        Try augmenting the query by adding a coercion (AUGMENT 3)
        between  $\theta$  and  $q_1$ .
        Let the signature  $u_1$  be the type of the query  $q_1$  after this augmentation.
        if the augmentation fails then {
            Issue a message on the type error.
            Try to restore the process by finding the most probable signature
             $u_1$  in this position (RESTORATION 3).
        }
    }
}

// strip the collection operator from  $u_1$  before application of static_nested
if  $u_1$  is of the form bag{ $T$ } or sequence{ $T$ } then {
     $u_{1\text{stripped}} := T$ 
} else {
     $u_{1\text{stripped}} := u_1$ 
}

// if the opened object belongs to classes, push sections of these classes
if  $u_{1\text{stripped}}$  is the static identifier connected by static_OC with a class then {
    Let  $C$  be the class connected by static_OC with  $u_{1\text{stripped}}$ .
    for each class  $C_S$  which is a superclass of  $C$  do {
        push(S_ENVS, static_nested( $C_S$ ));
    }
    push(S_ENVS, static_nested( $C$ ));
}

push(S_ENVS, static_nested( $u_{1\text{stripped}}$ ));
static_type_check( $q_2$ );
 $u_2 := \text{pop}(\text{S\_QRES})$ ;
pop(S_QRES); // pop  $u_1$  from S_QRES
pop(S_ENVS); // pop static_nested( $u_{1\text{stripped}}$ ) from S_ENVS
if  $u_1$  and  $u_2$  fit the inference rule for  $\theta$  then {
    Let the signature  $t$  be the type of  $q_1\theta q_2$  indicated by this inference rule.
} else {

```

```

    Try augmenting the query by adding a dereference (AUGMENT 2) or
    a coercion (AUGMENT 3) between  $\theta$  and  $q_2$ .
    Let signature  $t$  be the type of the query  $q_1\theta q_2$  after this augmentation.
    if the augmentations fail then {
        Issue a message on the type error.
        Try to restore the process by finding the most probable signature  $t$ 
        in this position (RESTORATION 2).
    }
}
push(S_QRES, t);
} // case query is  $q_1\theta q_2$ 
} // procedure static_type_check

```

7. Example static type check

We use the following query as an example:

((Dept **where** name = „IT”) **join** (boss . Emp))

Fig.6 and Fig.7 show the states of both static stacks (S_ENVS and S_QRES) during the static type check of this query. If the state of the static environment stack does not change, its state is not repeated on these pictures. Here is the description of subsequent steps of this type check:

1. This is the initial state. The static result stack is empty, while the static environment stack contains the static binders to all static root identifiers (cf. section 3).
2. The name *Dept* is bound on the static environment stack. As the result, the signature **bag** $\{i_{Dept}\}$ is pushed onto the static result stack.
3. The first non-algebraic operator (**where**) opens a new section on the static environment stack. It contains *static_nested*(i_{Dept}). The **bag** part of the signature **bag** $\{i_{Dept}\}$ has been stripped before the application of the function *static_nested*.
4. The name *name* is bound. As the result, the signature i_{name} is pushed onto the static result stack. The static environment stack stays intact.

5. The equality operator is evaluated, i.e. first the type of the right subquery (`string`) is pushed onto the static result stack, then the equality pops two sections and pushes the type of its result (`boolean`). In fact here the dereference augment (see section 8.2) is applied, but we skip it for the sake of clarity.
6. The evaluation of the **where** operator concludes. The top sections of the static environment stack and the static result stack are removed and the type of the result of **where** is pushed onto the static result stack (it

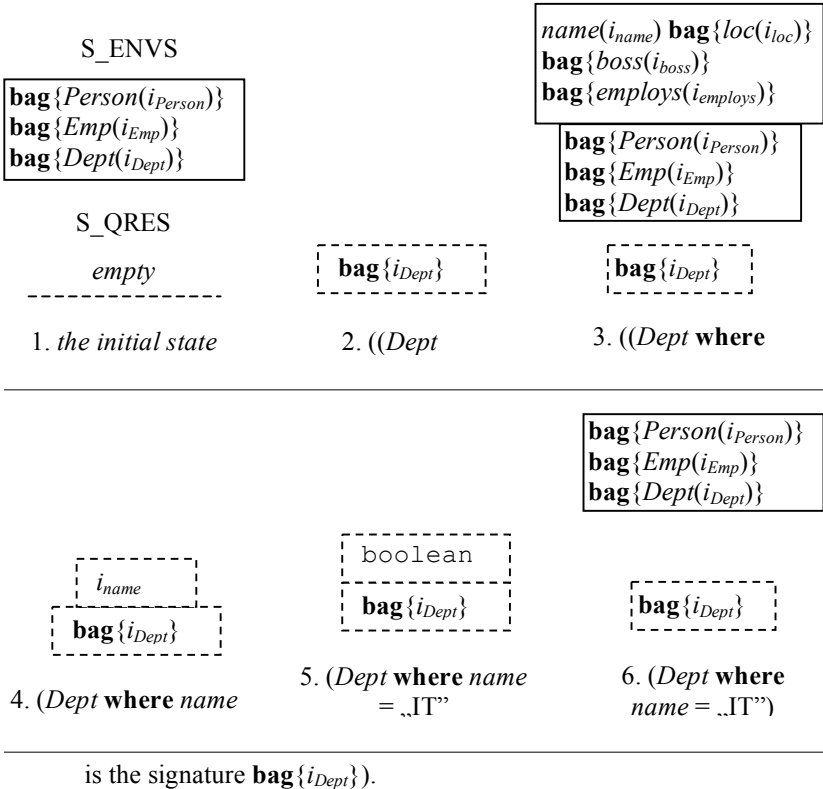


Fig.6. States of the static stacks during example type check (part 1)

7. The second non-algebraic operator (**join**) opens a new section on the static environment stack. It contains the application of the function *static_nested* to the signature i_{Dept} .
8. The name *boss* is bound. As the result, the signature **bag** $\{i_{boss}\}$ is pushed onto the static result stack (the subobject *boss* is optional inside the complex object *Dept*).
9. The third non-algebraic operator (the dot) opens a new section on the static environment stack. It contains the application of the function *static_nested* to the signature i_{boss} . This function “traverses” the static link object, therefore the new section contains the signature $Emp(i_{Emp})$.
10. The name *Emp* is bound. As the result, the signature i_{Emp} is pushed onto the static result stack.
11. The evaluation of the dot concludes. The top sections of the static environment stack and the static result stack are removed. The type inference rule for the dot is applied. The inferred result signature **bag** $\{i_{Emp}\}$ is pushed onto the result stack.
12. The evaluation of the join concludes. The top sections of the static environment stack and the static result stack are removed. The type inference rule for the join is applied. The inferred result signature **bag** $\{\mathbf{struct}\{i_{Dept}, i_{Emp}\}\}$ is pushed onto the result stack.

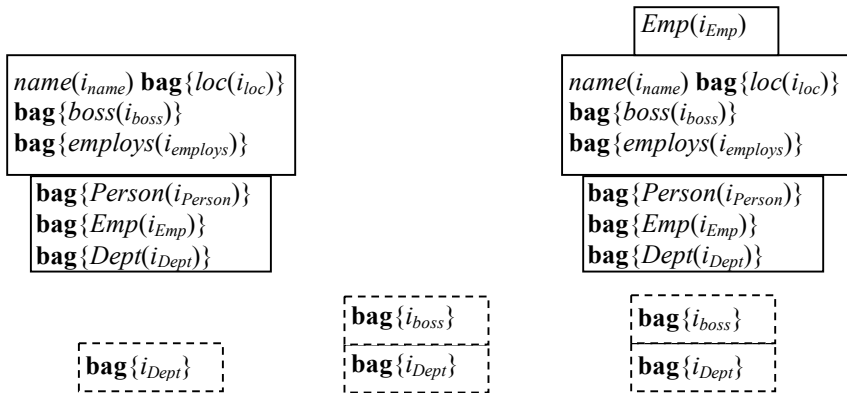
As the result of the type check, we have found out that the query:

$((Dept \text{ where } name = „IT”) \mathbf{join} (boss.Emp))$

is type correct and its type is the following signature:

bag $\{\mathbf{struct}\{i_{Dept}, i_{Emp}\}\}$

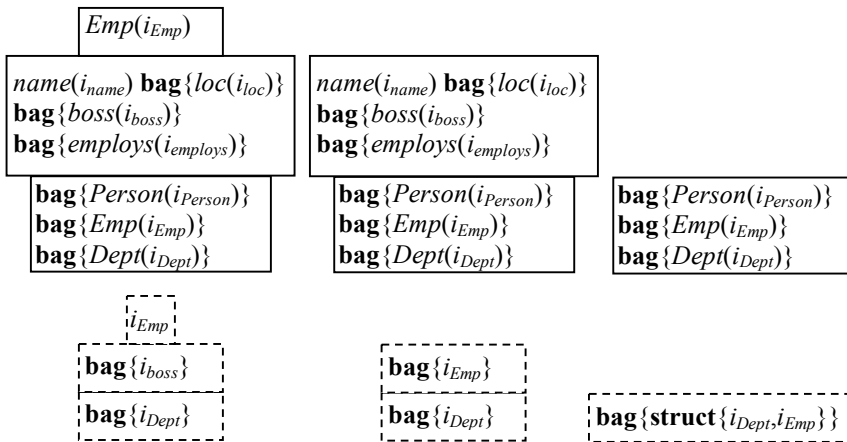
Fig.6 and Fig.7 are slightly simplified. Objects *Emp* belong to the classes *EmpClass* and *PersonClass*, so in the third step the sections of these classes are pushed onto S_ENVS below the section of the object *Emp*. These sections contain static references to methods which are not used in the analyzed query. Therefore we decided not to present them in both figures. Fig. 8 shows non-simplified state of S_ENVS for the third step. These two additional sections with methods are pushed exactly in the same place also in steps 7, 9, 10 and 11.



7. ((Dept where name = „IT”) join

8. ((Dept where name = „IT”) join (boss

9. ((Dept where name = „IT”) join (boss.



10. ((Dept where name = „IT”) join (boss.Emp

11. ((Dept where name = „IT”) join (boss.Emp

12. ((Dept where name = „IT”) join (boss.Emp)

Fig.7. States of the static stacks during example type check (part 2)

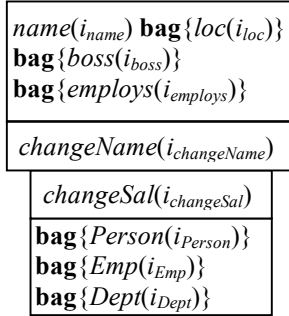


Fig. 8. Non-simplified state of S_ENVS in the third step

8. Augmenting the query

During the static type check described in section 6 the syntactic tree of a query can be augmented with some operators. In this paper we consider three such situations: coercion, derefence and ellipsis. We marked them in the text of the procedure *static_type_check* with the label **AUGMENT**. Below we describe each of these augments.

8.1 Ellipsis (**AUGMENT 1**)

People tend to shorten queries. They omit phrases which can be deduced by the query engine. For example they often “forget” to put the name of the object referenced by a link object. Let us recall the following correct query:

$(Dept \textbf{ where } name = "IT").boss.\underline{Emp.name}$

It might seem obvious that the underlined name can be omitted. In fact, it should be present, because after the dot operator any subquery is allowable (not only the name of the referenced object). Such an omission is called an *ellipsis* and the whole query is called an *elliptic query*.

If one types a query like this:

$(Dept \textbf{ where } name = "IT").boss.name$

the procedure *static_type_check* will reach the label **AUGMENT 1**, because the static environment stack does not contain any binder labelled *name*. In this case, the procedure can detect that the object most recently traversed by the

function *static_nested* was a static link object and the procedure can augment the processed query with the missing phrase.

This leads to the general rule which can be applied in such cases. When a name cannot be bound on the static stack, augment the query by preceding this name with the phrase “*m.*”, where *m* is the name of the static link object most recently traversed by the function *static_nested*.

Although it could be dangerous, we can further generalize this augmenting method, by keeping an additional stack of static link objects which were arguments to recent calls to *static_nested*. When a name cannot be bound, we try to precede it with names of static objects referenced by subsequent static link objects from the top to the top of this stack.

8.2 Dereference (AUGMENT 2)

Many algebraic operators (cf. section 5.1) require arguments of primitive types (integer, float, boolean and string). The type of the second argument of selections (cf. section 5.7) and quantifiers (cf. section 5.8) must be boolean. However in many cases, the subquery being the argument of such operator yields the reference to an atomic object which holds the value of appropriate primitive type. In such cases we add a dereference between this operator and this subquery.

Let us recall the example from section 8.1:

(Dept where name = "IT").boss.Emp.name

During the analysis of the subquery

name = "IT"

the procedure *static_type_check* will reach the label **AUGMENT 2**, because the static result stack contains the section with the signature i_{name} (the result of the static evaluation of the subquery *name*) which does not match any row of the table from section 5.1. In this case, the procedure can augment the processed query with a dereference applied to the first subquery *name*:

(Dept where deref(name) = "IT").boss.Emp.name

This leads to the general rule which can be applied in such cases. When the type of an argument of an arithmetic/string operator or the second argument of

a selection or a quantifier is the signature i_N for some name N , augment the query by applying the operator **deref** to this argument.

8.3 Coercion (AUGMENT 3)

There are three situations when a coercion is necessary:

- When an operator expects an argument which is a single item and not a collection.
- When an operator expects an argument which is a collection and not a single item
- When an operator expects a scalar value of a different type.

In all these cases we augment the query with the coercion operator which performs the run-time dynamic type check and possibly coerces its argument to the value of the proper type.

From a collection to an element. The operator **element** returns the only element of the collection passed as its actual argument. If the argument collection is empty or contains more than one argument, an exception is raised. The type of arguments of the operator **element** is a signature **bag** $\{T\}$ or **sequence** $\{T\}$. The result type of this operator is the signature T . Let us consider the following query:

(Emp where sal = 1000).name

The type of the subquery *sal* is the signature **bag** $\{i_{sal}\}$ because *sal* is an optional subobject of *Emp*. However, the equality expects arguments which are single valued. Therefore this query is augmented with the operator **element**:

(Emp where element(sal) = 1000).name

If there is at least one object *Emp* without any subobject *sal*, the evaluation of this query will raise an exception. Of course the careful programmer will code this query in the following way:

((Emp where count(sal) = 1) where sal = 1000).name

This way he/she can avoid this exception.

The general rule is that whenever an operator expects a single item and not a collection (**bag** $\{T\}$ or **sequence** $\{T\}$), the query gets augmented with the operator **element** applied to this argument.

From an element to a collection. The situation is just reverse to the previous one, but much easier. When required, e.g. in a structure constructor, a type signature `T` is changed to `bag{T}`. Providing careful implementation, no change to syntactic tree is necessary. The coercion does not imply run-time typecheck.

Mapping scalar values. The operators `to_float`, `to_boolean` and `to_string` accept arguments of any primitive type and return a result of one of the types respectively: `float`, `boolean` and `string`. If their actual argument can be coerced to a value of the appropriate type, this value is returned.

All the rules below apply only if the types of arguments of an operator does not match any row of the table from section 5.1.

- If the type of one of arguments of an arithmetic operator (`+`, `-`, `*`, `/`) is `integer` while the type of the other is `float`, the query gets augmented with the operator `to_float` applied to the argument of the type `integer`.
- If the type of at least one argument of the addition (`+`) is neither `integer` nor `float`, the query gets augmented with the operator `to_string` applied to these arguments which are not of the type `string`. The operator in the syntactic tree is flagged as concatenation.
- If the type of one of arguments of a comparison (`=`, `<`, `>`, `!=`) is `integer` while the type of the other is `float`, the query gets augmented with the operator `to_float` applied to the argument of the type `integer`.
- If the type of one of arguments of a comparison (`=`, `<`, `>`, `!=`) is neither `integer` nor `float`, the query gets augmented with the operator `to_string` applied to these arguments which are not of the type `string`.

9. Restoring the Process

A type checking of a query/program should be continued despite previous type errors. Otherwise the checking will consume too much time and patience of the programmers. This implies that after signaling a type error the process of type checking should be restored to the most probable correct state. Unfortunately,

the restoring rules can be very complex depending on investigations concerning the most probable type errors for the given environment. In this report we approximate these rules by some simpler ones. More complex rules can be introduced to the type checker after observations of annoying false type error messages that result from improper restoring of the type checking process.

We have marked restoration rules in the text of the procedure *static_type_check* with the label **RESTORATION**. Below we describe them.

9.1 Misspelled name (RESTORATION 1)

If a name cannot be bound on the static environment stack, it is possible that the programmer has misspelled this name. In such a case the procedure *static_type_check* will issue a message on the type error, but it will also try to continue the process in order to detect more than one error during one type check pass.

When a name n cannot be bound on the static environment stack, the procedure will scan this stack from its top and search for a signature $m(T)$ or **bag** $\{m(T)\}$ such that the edit distance between n and m is 1. If such a signature is found, the procedure pushes the signature T or **bag** $\{T\}$ respectively onto the static result stack and continues. If such a signature is not found, the procedure can search for names with the edit distance to n equal 2, and so on.

For example the following query contains a misspelled name:

(Dept where deref(name) = "IT"),bos.Emp.name

The name *bos* cannot be bound on the static environment stack. However during the static type check of the subquery *bos* the top section of this stack contains the signature **bag** $\{boss(i_{boss})\}$. The edit distance between *bos* and *boss* is 1. Therefore the procedure pushes the signature **bag** $\{i_{boss}\}$ onto the static result stack and continues. Of course from this point the type check is marked as failed in order to prevent its future success.

This description of this restoration is very general. One can limit it by restricting the number of visited sections of the static environment stack or setting the maximum allowed edit distance.

9.2 Malformed subqueries (RESTORATION 2)

If arguments of an operator do not fit the type inference rule for this operator, the procedure *static_type_check* will issue a message on the type error, but it will also try to continue the process in order to detect more than one error during one type check pass. It simply searches for the most probable type of the result of this operator and pushes this type onto the result stack.

Arithmetic operators. In case of +, -, * and / we use `float` as the most probable type of the result.

Comparisons and boolean expression. In case of =, !=, <, >, AND, OR, NOT we use `boolean` as the most probable type of the result.

Other algebraic operators (\cup , \cap , **except**, \times , **as**, **group as**) described in this paper (see sections 5.2, 5.3 and 5.4) cannot cause type errors. Thus, there are no rules how to restore the type check for these operators.

Navigation and dependent join (see sections 5.5 and 5.6) have the same property, thus no restoration rules exist for them.

Selection (see section 5.7). If a type error occurs (it is always caused by the right subquery whose type is not `boolean`), we use the type of the left subquery coerced to the appropriate collection type as the most probable type of the result. More formally, if the type of the left subquery is a signature T which is neither **bag** nor **sequence**, then the most probable type of the selection is the signature **bag** $\{T\}$. Otherwise, the most probable type is simply the signature T .

Quantifier (see section 5.8). If a type error occurs (it is always caused by the right subquery whose type is not `boolean`), we use the signature `boolean` as the most probable type of the result.

Transitive closure (see section 5.9). If a type error occurs (because the types of both subqueries differ), we use the type of the right subquery coerced to the appropriate collection signature as the most probable type of the result.

If one adds new operators to those described in this paper, he/she can define specific restoration rules for these operators.

9.3 Malformed left subquery (RESTORATION 3)

Non-algebraic operators described in this paper (see sections 5.5, 5.6, 5.7, 5.8, 5.9) place no limits of the type of their left subqueries. Therefore there are no restoration rules to apply when the type of the left subquery is malformed.

However, the stack based approach (see section 2) is a general framework which allows adding new non-algebraic operators. The definer of such an operator may set some limits for types of its left subquery. If so, it is worth to define rules which allow restoring the type check after inappropriate type of the left subquery is encountered. These rules are to be applied when the procedure *static_type_check* hits the label **RESTORATION 3**.

10. External Types

In this report we will not focus on the external type system. This is currently our research and will be the subject of a next report. However, here we would like to present major issues. The external type system presented below is based on following assumptions.

- The type system should have small and non-redundant number of primitive types.
- The type system has to be consistent with query languages.
- The type system should be natural for database programmers.

We do not fix the number of primitive types. In this report we introduce 5 of them. These are: **integer** (typical signed integer known from Java or C#), **float** (floating point number), **boolean**, **string**, **date**.

The following example demonstrates the usage of primitive types in the external type system.

```
typedef TypePerson = struct {  
    name : string; surname : string; age : integer; height : integer;  
    employed : boolean; dateOfBirth : date;  
}
```

For complex types we assume structural type conformance. Let us to consider a type definition:

```
typedef TypeEmployee = struct { name : string; age : integer; };
```

and two declarations of objects:

```
employee1 : TypeEmployee;  
employee2 : struct { name : string; age : integer};
```

The comparison `employee1 = employee2` is correct from the point of view of the type system.

Assuming cardinalities introduced previously we can define optional database entities as well as collections (repeated database entities) in the UML style. For instance, the following statement declares a collection of objects `Person`:

```
Person : struct {  
    surname : string; age : integer; height : integer; employed : boolean;  
    dateOfBirth : date; maidenName : string[0..1]; names : string[1..*];  
    children : ref to Person[0..*];  
}[0..*]
```

The mapping from an external type system into an internal type system is the subject of a special processor (a database schema compiler). As usually, the compilation is driven by the syntax of type declarations and schema specification. The assumption is that *typedef* declarations are considered by this compiler as macros. Eventually, the compiler takes an external schema declaration as input and produces the corresponding metabase graph. Because an external type system is usually tangled with other options (e.g. declarations of classes, interfaces, views, modules, exports/imports) in this report it is too early to consider such a compiler in detail.

11. ODBA and its Type System

ODBA (**O**bject **D**atabase for **R**apid **A**pplication development) is a prototype application development environment currently being developed (in C#) at the Polish-Japanese Institute of Information Technology. The system is based on a new programming language (a syntactic variant of an extended SBQL) and its execution environment. The execution environment consists of the SBQL interpreter, a main-memory database management system and an infrastructure supporting grid computing.

In this section we briefly describe some aspects of the type system designed for Odra in its early stage of development. The goal is to provide a sketch of what a programmer could do to implement a type system using mechanisms described in this report.

11.1 Types and Variables

Currently, Odra supports the following primitive types: boolean, date, integer, real, string; the following complex types:

- sequence of *type*,
- bag of *type*,
- array of *type*,
- *class*,
- *struct_defined_inline* (like in databases),
- *struct_defined_out_of_line* (like in programming languages);

and the following reference type:

- *ref type*.

The main difference between arrays and sequences is that an array may be multidimensional, its size is fixed and has to be explicitly declared.

The types enumerated above can be used to declare variables (local or global objects). Every declaration allows a programmer to specify a name of an object, its type and its cardinality. As we have decided to introduce only [0..1], [0..*], [1..*] and [1..1] cardinalities, it is possible to have a program containing the following declarations:

```
first_name, last_name : string; // cardinality [1..1]
optional maiden_name : string; // [0..1]
works_in : bag of ref Department; // [1..*]
optional email : sequence of string; // [0..*]
```

The cardinality specifies the minimum and maximum allowed number of instances. When the system processes a declaration, which says that the minimum cardinality of an object is 1, the system automatically creates an object with its default value in order to satisfy this requirement. For example, in the case of the following declarations:

```
name : string;
optional phone : sequence of string;
```

during runtime, after processing these declarations, there is one object name (automatically created by the system) and zero objects `phone`. To create a new `phone` object, we use the operator `create`:

```
create phone := "022-3656344";
create phone;
create phone := "036-4434444";
```

It is impossible to explicitly create (using the operator `create`), nor delete (using the operator `delete`) the object name, because one and only one object of such name must always exist in the database (its cardinality is `[1..1]`). Similarly, it is impossible to delete all `phone` objects, but the upper limit is unrestricted (the cardinality is `[1..*]`).

11.2 Data Store and Metabase

The data store organizes data and ensures fast consistent and secure access to the database. In accordance with the M0 data store model, it is possible to create simple objects (implemented as the classes `IntegerObject`, `StringObject`, `ProcedureObject` etc), reference objects (the class `ReferenceObject`), as well as complex objects (the classes `SpiderObject`, `ViewObject`, etc.). Every object has a name, an object identifier (OID) and some content (a value, a procedure body, etc.). The objects are stored in an ordinary C# table, and the OID is an index in this table.

A metabase is a kind of a data store, comprising mostly objects of the following classes: `ReferenceObject` (used to represent references between meta-objects), `SpiderObject` (used to represent complex meta-objects) and `IntegerObject` (used to represent simple meta-objects). Every `IntegerObject` contains a value identifying a primitive type (for example 1 for integer, 2 for string, etc.). Every object in the meta-base carries a special attribute that describes cardinality, mutability, etc.

Apart from the meta-objects described above, the metabase may contains special purpose reference objects. The reference object named `$super` is used to define parent-child relationship between objects that are instances of classes connected with the inheritance relationship. The reference object named `$class` is used to represent class membership.

Both data store and metabase can be global and local. The global data store is an equivalent of the traditional database. It is stored on the server and contains data that is shared among users. The local data store is an equivalent of the

heap in traditional programming languages. It is stored on the client side and contains client's private data.

The global metabase is an equivalent of the database schema. If a global variable is declared, a suitable meta-object must be created in the global metabase. The client must download the metabase from the server in order to compile/execute SBQL programs. The local metabase is an equivalent of the symbol table in traditional programming languages. It is used both during compilation and runtime. Every stack frame contains a separate local metabase. Every local variable declaration must be reflected in the local metabase according to the scope of the variable. When the stack collapses, the local declarations (in the form of local metabases) cease to exist.

11.3 Parser and Type Checker

SBQL in ODRA is a purely interpreted language. The compiled program takes a form of an abstract syntax tree (AST), which is analysed by the interpreter. The parser constructs ASTs using a set of predefined classes. For example, in the following query:

```
Employee where Salary > 100;
```

the parser executes the operation below:

```
Program prog =
  new Program(
    new ExpressionStatement(
      new WhereExpression(
        new IdentifierExpression(new Identifier("Employee")),
        new BinaryOperatorExpression(
          new IdentifierExpression(new Identifier("Salary")),
          new IntegerExpression(new IntegerLiteral(100)),
          new Operator(Operator.GREATER))))));
```

Syntax trees constructed in this manner are then analysed during subsequent compilation phases (semantic checking, optimisation, etc.), and are either stored in the database (for example as procedures), or immediately executed (ad hoc queries).

Having an AST representing an SBQL source program, it is necessary to design an effective and easy to implement mechanism to traverse syntactic trees. A widely known solution to this problem is based on a popular design pattern called *visitor*. This pattern allows the programmer to isolate the structure of all

AST nodes from the code that operates on them. Thus, every module of the interpreter can be implemented as a separate class, comprising methods specific only to one particular mechanism (for example type checking).

During semantic analysis, the result of static query evaluation is called a *signature* (cf. Section 4). Signatures in ODRA are implemented as individual classes:

- `BinderSignature` - An object of this class denotes a signature of a query which will return a binder.
- `ReferenceSignature` - An object of this class denotes a signature of a query that will return a reference. It contains a reference to a meta-object representing a declaration of the variable referenced during runtime.
- `StructSignature` - An object of this class denotes a signature of a query that will return a structure. It contains a list of signatures describing elements of the structure.
- `VariantSignature` - An object of this class denotes a signature of a query that will return a **variant**. It contains a list of signatures describing possible results of the query.
- `ValueSignature` - An object of this class denotes a signature of a query that will return a value. The object contains information describing the type of the value. Types are internally represented as independent AST nodes/trees. Every type (for example `IntegerTypeDenoter`) inherits from the class `TypeDenoter`.

Every signature is described by an additional attribute, which defines whether the signature represents a single result, or a collection of results (bag or sequence). During semantic analysis instances of these classes are pushed onto the static query result stack.

11.4 Semantic Checking in ODRA

The semantic analysis in ODRA is supported by three components: the meta-base, the static environment stack (`S_ENVS`), and the static query result stack (`S_QRES`). The first goal of this process is to statically (at compile time) evaluate an SBQL program using a meta-base, in order to detect signatures of every expression. The second goal is to find all undeclared names that occur in an SBQL program.

The static environment stack in SBA is an equivalent of the symbol table pertaining to traditional compilers. It is therefore implemented as a class with methods `Enter`, `Bind`, `OpenScope`, `CloseScope`, etc. The class representing the static query result stack is much simpler. It contains mainly two methods: `Push` and `Pop`.

Before the process of semantic checking can be started, the stack must be initialised. First, the standard environment is established and binders to pseudo-declarations of primitive types and arithmetic operators are pushed onto `S_ENVS` as part of a separate scope. Every pseudo-declaration of an arithmetic operator consists of information on the type of the result, desired left and right operand types, and, optionally, necessary coercions between types of operands. After the standard environment is established, the stack is augmented with a new section containing binders to all global declarations (taken from the metabase).

When the initialisation process is completed, the semantic analysis can be performed. The type checker traverses the syntax tree and infers signatures in the following way:

- For a literal node, the signature can be inferred using information from the parser. For example, in the case of “some string” literal, the signature `ValueSignature(StringTypeDenoter)` is build and pushed onto `S_QRES`.
- For an identifier node, the name of the identifier is bound on `S_ENVS`, and the signature `ReferenceSignature(ref_to_declaration)` is created and pushed on `S_QRES`.
- For an arithmetic operator, the signature can be inferred from the operator’s pseudo-declaration and from operands’ signatures, which are popped from `S_QRES`.
- For a non-algebraic operator (and some algebraic) the signature is returned by a special purposed routine that is specific to the particular operator.

As a result of the type checking process, the semantic analyser may also modify the AST. The aim of such modifications is usually to implement coercions (cf. Section 8). Hence, to coerce from a collection to a single element, the type checker inserts a `ToSingleExpression` node. To implement coercion from an integer value to a string value, the type checker inserts a `ToStringExpression` node, etc.

Taking into account the above assumptions, we can provide a detailed example of the work that the contextual analyser performs. In the case of the expression $q1 + q2$, the semantic analysis consists of the following steps:

1. The signature of the left operand is inferred.
2. The signature of the right operand is inferred.
3. If one of these signatures denotes a binder, its value is used for subsequent processing. If the value is also a binder, it is recursively unnested.
4. Between `BinaryExpression` and `IntegerExpression` nodes, the node `ToSingleExpression` is inserted. The signature of the node `ToSingleExpression` is the same as the signature of its subnode, except for the collection flag (bag or sequence), which is unset.
5. The type checker inserts a `Dereference` node between the `BinaryExpression` node and the `ToSingleExpression` node. If necessary (`ToSingleExpression` results in a `ReferenceSignature`) a new `ValueSignature` is constructed using the type retrieved from the metabase.
6. If at this moment one of the operand signatures is not a `ValueSignature` (but for example a `StructSignature`), the type error exception is raised.
7. The name of the operator “+” is bound on the stack. The set of bound operator declarations is searched, and the appropriate version (suitable for the types retrieved from the signatures of the operands) is chosen. If such an operator has been found, it may be necessary to insert a coercion node between the `BinaryExpression` node and the operand node (for example `ToString` for the left operand in the case of $1 + \text{“test”}$). If the appropriate operator has not been found, the type error exception is raised. The final signature is built using a type taken from the operator declaration.

The analysis of non-algebraic operators is not so straightforward, because every operator has a separate routine specific to this operator. For example for $q1$ where $q2$, the type checker executes the following operations:

1. Evaluates the left operand and infers its signature.
2. Opens a new `S_ENVS` section.
3. Calls `Nested` on the signature inferred from the left operand.
4. Evaluates the right operand and infers its signature.

5. Removes the S_ENVS section.
6. Checks if the signature of the right operand denotes a simple value, and if its type is `boolean`. If the type is not `boolean`, the type error exception is raised.
7. The signature of the operator `where` is the signature of its left operand.

```

C:\Documents and Settings\raist\Moje dokumenty\Visual Studio Projects\Odra\bin\Deb...
Welcome to Odra!
$ print schema
0000 entry = <
0001   Department = <
0002     Name = String [1..1]
0003     City = String [1..1]
0004     Established = Integer [1..1]
0012     Employs = -> Employee [0..1]
0005   > [0..*]
0006   Person = <
0007     Name = String [1..1]
0008     Age = Integer [1..1]
0009   > [0..*]
0010   Employee = <
0011     $super = -> Person [1..1]
0012     Salary = Integer [0..1]
0013     WorksIn = -> Department [0..1]
0014   > [0..*]
0015 > [1..1]
$ print database
0000 entry = <
0001   Department = <
0002     Name = Sales
0003     City = Warsaw
0004     Established = 2004
0005   >
0006   Person = <
0007     Name = Kowalski
0008     Age = 23
0009   >
0010   Person = <
0011     Name = Nowak
0012     Age = 53
0013   >
0014   Employee = <
0015     $super = -> 5
0016     Salary = 100
0017     WorksIn = -> 1
0018   >
0019   Employee = <
0020     $super = -> 8
0021     Salary = 200
0022     WorksIn = -> 1
0023   >
0024 >
$ -

```

Fig. 9. A sample database and its schema

For $q_1 \cdot q_2$, the steps are different. The type checker executes the following operations:

1. Evaluates the left operand and infers its signature.
2. Opens a new S_ENVS section.
3. Calls `Nested` on the signature inferred from the left operand.
4. Evaluates the right operand and infers its signature.
5. Closes the S_ENVS section.
6. The signature of the operator `.` is the signature of the right operand.

11.5 The prototype at work

An example of the expression $q_1 + q_2$ is the following query:

```
Department.Established + 5;
```

The syntax tree of this query can be seen in Fig. 10. The original syntax tree produced by the parser is modified by the type checker, which assumes the query is semantically correct and modifies it introducing some new nodes. The whole processing is done on the database schema presented in Fig. 9.

```

Welcome to Odra!
$ break after parsing
$ Department.Established + 5;
 9  0 Program
 8  9 ExpressionStatement
 7  8 BinaryExpression (<+> <NS> <source: 1, 24>
 5  7 DotExpression <NS> <source: 1, 11> <scopes: 0-0>
 3  5 IdentifierExpression <Department> <NS> <source: 1, 1> <scope
 4  5 IdentifierExpression <Established> <NS> <source: 1, 12> <sc
 6  7 IntegerExpression <5> <NS> <source: 1, 26>

$ break after typechecking
$ Department.Established + 5;
16  0 Program
15 16 ExpressionStatement
14 15 BinaryExpression (<+> <ValueSignature,IntegerTypeDenoter> <sour
72 14 DereferenceExpression <ValueSignature,IntegerTypeDenoter> <so
71 72 ToSingleExpression <ReferenceSignature> <source: 1, 11>
12 71 DotExpression <bag of ReferenceSignature> <source: 1, 11> <
10 12 IdentifierExpression <Department> <bag of ReferenceSignatu
11 12 IdentifierExpression <Established> <bag of ReferenceSignatu
13 14 IntegerExpression <5> <ValueSignature,IntegerTypeDenoter> <so

$ Department.Established + true;
Type error at line 1, column 24:
Operator '+' cannot be applied to operands of types 'Integer' and 'Boolean'
Department.Established + true;
$ _

```

Fig. 10. Type checking of a query in Odra

12. Conclusions

In this report we have proposed a new approach to static type checking. We have taken the practitioners' point of view thus avoiding impractical type theories. Many interrelated aspects of a strong type checking mechanism, irregular, arbitrary choices that must be taken during the development, dependencies on programmers' psychology (usually hard to recognize without experiments), and other factors have caused our loss of belief that any academic type theory could bring an essential support in the development of strong static type systems for object-oriented query/programming languages.

We have presented the simple and intuitive type checking mechanism for an object-oriented database model. Our type system consists of two sides: internal and external. The internal type system consists of a metabase, a static environment stack, a static result stack and type inference rules. The rules are represented as decision tables and are defined for all operators occurring in the given query/programming language. We have presented a pseudo-code of the appropriated static type checking procedure. We have shown how this procedure can be used to correct certain type errors in queries and to recover a type checking process from wrong state that may occur after a type error. Such restorations allow detecting more than one error during one type check pass. The procedure makes it also possible to resolve some ellipses, to accomplish some type coercions and to insert dynamic type checking actions into a run-time query/program code.

Finally, we have presented the type system of our experimental platform ODBA, which is already operational. The implementation has fully proven feasibility and universality of our type checking mechanism.

In this report we have only discussed rough assumptions of the external type system. More detailed presentation will be the subject of our next papers and reports.

13. References

- [Abad96] M. Abadi, L. Cardelli. A Theory of Objects. Springer Verlag 1996, ISBN:0387947752.
- [Alag94] S. Alagic: F-Bounded Polymorphism for Database Programming

- Languages. East/West Database Workshop, Springer 1994, 125-137
- [Alag97] S.Alagic. The ODMG Object Model: Does it Make Sense? Proc. OOPSLA Conf., 253-270, 1997
- [Alag99a] S.Alagic, Type Checking OQL Queries in the ODMG Type Systems, ACM Transactions on Database Systems, 24 (3), pp. 319-360, 1999.
- [Alag99b] S. Alagic, A Family of the ODMG Object Models. Proceedings of ADBIS '99, Springer LNCS 1691, 1999, pp.14-30
- [Atki87] M.Atkinson, P.Buneman. Types and Persistence in Database Programming Languages. ACM Computing Surveys 19(2), 105-190, 1987
- [Atki95] M.Atkinson, R.Morrison. Orthogonally Persistent Object Systems. The VLDB Journal 4(3), 319-401, 1995
- [Cann89] P.S.Canning, W.R.Cook, W.L.Hill, W.G.Olthoff, J.C.Mitchell: F-Bounded Polymorphism for Object-Oriented Programming. FPCA '89 Conference on Functional Programming Languages and Computer Architecture. ACM Press, 1989, pp.273-280
- [Card84] L.Cardelli: A Semantics of Multiple Inheritance. Information and Computation, Volume 76, Number 1, January 1988, pp.138-164
- [Card85] L.Cardelli, P.Wegner. On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys 17(4), 471-522, 1985
- [Card89] L.Cardelli. Typeful Programming. Technical Report, Digital Systems Research Center Report 45, Palo Alto, USA, 1989
- [Gold97] R.Goldman, J.Widom, DataGuides: Enabling Query Formulation and Optimization Semistructured Databases, Twenty-Third International Conference on Very Large Data Bases, , pp. 436-445, <ftp://db.stanford.edu/pub/papers/dataguide.ps>, 1997.
- [Habe02] P.Habela, M.Roantree, K.Subieta. Flattening the Metamodel for Object Databases. Proc. of ADBIS'2002 Conf. Bratislava, Slovakia, Springer LNCS 2435, pp. 263-275, 2002
- [Habe03] P.Habela, K.Subieta: Overcoming the Complexity of Object-Oriented DBMS Metadata Management. Proc. OOIS'2003, Springer LNCS 2817, pp.214-225, 2003
- [Mitic88] J.C.Mitchell, G.D.Plotkin: Abstract Types Have Existential Type. ACM Trans. Program. Lang. Syst. 10(3), pp. 470-502 1988

- [MDA05] OMG Model Driven Architecture. Defined by the MDA Guide Version 1.0.1, <http://www.omg.org/mda/>
- [ODMG00] R.G.G.Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, F. Velez, The object data standard: ODMG 3.0, Morgan Kaufman, 2000.
- [OMG02] Object Management Group: OMG CORBA/IIOP™ Specifications. http://www.omg.org/technology/documents/corba_spec_catalog.htm 2002
- [Schm94] J.W.Schmidt, F.Matthes. The DBPL Project: Advances in Modular Database Programming. Information Systems 19(2): 121-140 (1994)
- [SML04] SML.NET: Functional programming on the .NET CLR, <http://www.cl.cam.ac.uk/Research/TSG/SMLNET/>
- [Subi04] K.Subieta. Theory and Construction of Object Query Languages, Publishers of the Polish-Japanese Institute of Information Technology, 2004 (in Polish)
- [Subi85] K.Subieta. Semantics of Query Languages for Network Databases, ACM Transactions on Database Systems, Vol. 10, No. 3, pp. 347-394, 1985
- [Subi87] K.Subieta “Denotational Semantics of Query Languages”, Information Systems, Vol. 12, No. 1, 1987
- [Subi90a] K.Subieta, M.Missala, K.Anacki “The LOQIS System. Description and Programmer Manual”, Institute of Computer Science, Polish Academy of Sciences, Report 695, Warsaw, November 1990
- [Subi90b] K.Subieta “LOQIS: The Programming System Having Database Capabilities”, Institute of Computer Science, Polish Academy of Sciences, Report 694, Warsaw, October 1990 (in Polish)
- [Subi91] K.Subieta “LOQIS: The Object-Oriented Database Programming System”, Proceedings of the 1st Intl. East/West Database Workshop on Next Generation Information System Technology, Springer LNCS 504, pp. 403-421, 1991
- [Subi95a] K.Subieta, C.Beer, F.Matthes, J.W.Schmidt. A Stack-Based Approach to Query Languages. Proc. 2nd East-West Database Workshop, 1994, Springer Workshops in Computing, 1995
- [Subi95b] K.Subieta, Y.Kambayashi, J.Leszczylowski “Procedures in Object-Oriented Query Languages”, Proc. of VLDB, pp. 182-193, 1995

Pracę zgłosiła: Elżbieta Pleszczyńska

Adresy autorów:

Rafał Hryniów# rhryniow@pjawst.edu.pl

Michał Lentner# m.lentner@pjawst.edu.pl

Krzysztof Stencel+ stencel@mimuw.edu.pl

Kazimierz Subieta*# subieta@ipipan.waw.pl

*) Institute of Computer Science Polish Academy of Sciences
ul. Ordona 21, 01-237 Warsaw, Poland

#) Polish-Japanese Institute of Information Technology
ul. Koszykowa 86, 02-008 Warsaw, Poland

+) Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland

Symbol klasyfikacji rzeczowej: H.2.3, H.2.4, H 2.5

Na prawach rękopisu

Printed as manuscript

Nakład 100 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku w marcu 2005.

Wydawnictwo IPI PAN.

ISSN 0138-0648