Type Consistency of Queries in an Object-Oriented Database System*

Dave D. Straube†
M. Tamer Özsu

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

{daves,ozsu}@cs.ualberta.ca

Abstract

Queries in object-oriented databases can return non-homogeneous sets of objects when no type restrictions are placed on the inputs to the query. The tradition has been to force homogeneity on the result by restricting the types of the inputs. This restricts the range of permissible, and possibly useful, queries. We propose a type consistency theory for queries in object-oriented databases which supports the existence of multiple types in the query result. The technique is illustrated by developing type inference rules for an object algebra. The main result is that the loss of type information associated with a query operation is reduced in most cases. We also show how type information is increased when queries are qualified by conjunctive predicates.

1 Introduction

An information system can be viewed as a large database of persistent facts and a collection of application programs which are run against this database [5]. A common problem in such a system is *impedance mismatch* [14] which results from incompatibilities between the application programming language and the database query language. An application language is best suited for specifying operational semantics while the database language is designed for managing concurrency and specifying queries.

One approach to this problem is the design of database programming languages which combine data definition, flow of control and query constructs in a consistent syntax [2]. A common requirement, independent of any particular language, is that a program variable be iteratively bound to each element in the set of objects returned by a query, e.g., portals in [17] and cursors in [1]. Ideally, a compiler should insure that this binding is type consistent in order to detect improper use of data as early during query processing as possible. This paper presents a set of type inference rules for queries in object-oriented databases (OODB) which make such type checking possible.

The topic of this paper is part of our broader research which aims at developing a query processing methodology for object-oriented databases (Figure 1). Briefly, the steps of the methodology are as follows. Queries are expressed in a declarative language which requires no user knowledge of object implementations, access paths or processing strategies. The calculus expression is first reduced to a normalized form by eliminating duplicates, applying identities and rewriting [11]. The normalized expression is then converted to an equivalent object algebra expression. This form of the query is a nested expression which can be viewed as a tree whose nodes are algebra operators and whose leaves represent the extents of classes in the database.

*This research has been supported in part by the National Sciences and Engineering Research Council (NSERC) of Canada under operating grant OGP-0951.

†Author presently at Banyan Systems, Inc., 115 Flanders Road, Westboro, MA, 01581.

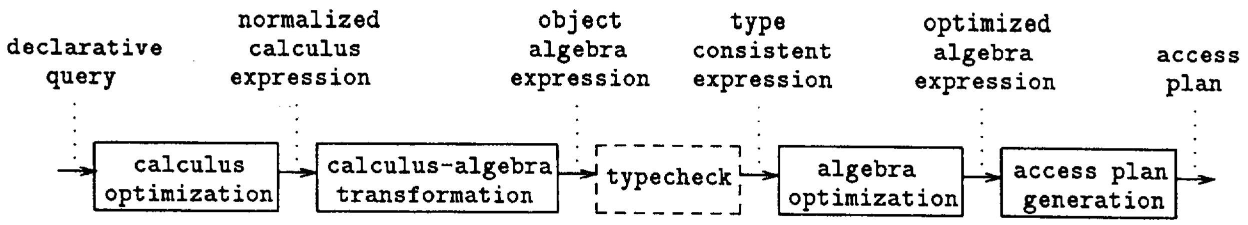
declarative query
normalized calculus expression
object algebra expression
type consistent expression
optimized algebra expression
access plan
calculus optimization
calculus-algebra transformation
typecheck
algebra optimization
access plan generation

Figure 1: Query processing methodology

The algebra expression is next checked for type consistency to insure that predicates and methods are not applied to objects which do not support the requested function. The next step in query processing is the application of equivalence preserving rewrite rules [18] to the type consistent algebra expression. Lastly, an access plan which takes into account object implementations is generated from the optimized algebra expression. The scope of this paper corresponds to the dashed box in Figure 1.

A distinguishing feature of our object model is that types (classes) inherit behavioral specifications as opposed to structural representations, e.g., record types [6, 7, 8]. This provides the following benefits [9]:

1. a type may have multiple representations and implementations,
2. subtypes may have representations and implementations which differ from their supertypes,
3. an implementation hierarchy may exist separate from the type hierarchy [12].

As a result, previous notions of type consistency based on the existence of record fields is inappropriate. Our type consistency of queries is based on the notion that each member of a query result supports a set of type specifications.

We investigate types and type consistency in the context of object algebra queries for an object-oriented database. Operators in the algebra consume and produce sets of objects which are instances of types in a type lattice supporting multiple inheritance. Previous algebras have imposed type restrictions such as *union compatibility* [16, 19] on the algebra operators to insure the type consistency of the result. Union compatibility states that members of the sets being operated on must be instances of types which are in a subtype relationship with one another. The type of the result is the most general supertype of the types involved in the operation. Such restrictions are too strong and can be avoided by a notion of type consistency which allows for multiple types.

For example, what is the objection to taking the union of a set of *Apple* objects and a set of *Orange* objects? The result of the union is a set of non-homogeneous objects in the sense that they are not necessarily related by type inclusion. The real problem lies not with the union operation, but with operators which consume the result of this union and apply methods to each object therein. We propose a type consistency theory for object algebra expressions with multiple types which resolves this issue.

The type consistency theory is developed as a series of type inference rules for object algebra expressions. We do not propose a type checking algorithm. Instead, an algorithm can be shown to be correct is it computes types which are also derivable by the inference rules. A system which is capable of performing type checking of expressions using just the inference rules as input is documented in [10]. In addition to the inference rules, this paper shows how the predicates which qualify algebra operators cause type information to be lost or gained. Type information is lost when the types associated with the result of a query are more general (higher in the type lattice) than the types associated with the query inputs. When the types associated with a query result are more specific (lower in the type lattice) than those of the inputs, type information has been gained.

The paper proceeds as follows. Section 2 outlines our OODB data model and formally develops the notion of a set of objects having multiple types. Denotations and predefined functions for use in the inference rules are presented in Section 3. Section 4 introduces predicates which can be used to qualify object algebra operators and their type inference rules. Section 5 presents the algebra operators and their corresponding inference rules. We conclude in Section 6.

2 Types in Object Oriented Databases

2.1 Data Model Overview

We briefly summarize the OODB data model in order to provide a foundation for the subsequent discussion. The model and object algebra are described in more detail in [18]. We emphasize that the focus of our research is on query processing issues in OODBs as opposed to data model issues. The key features of the model are as follows. Objects are viewed as instances of abstract data types (ADT) which can only be manipulated via functions defined by the type. Types are organized in a lattice which allows multiple inheritance. Each object has a time invariant identity which is independent of its state. Representations of objects are visible only to type implementors. Any aspect of the object representation which is required by users of a type should be revealed by the implementor via a method.

A *class* defines both an ADT interface via *methods* and stands for all the objects which are instances of the type. Methods are named functions whose arguments and result are objects. Each method has a signature of the form $t_1 \times \ldots \times t_n \rightarrow t_{result}$ where $t_1 \ldots t_n$ specify the types of the argument objects and t_{result} specifies the type of the result object. All types in the database form a lattice where the root node represents the most general type of objects and any individual type may have multiple parents. Subtypes inherit behavior from their parents and may define additional methods. Thus, the type lattice provides inclusion polymorphism [8] which allows an object of type t to be used in any context specifying a supertype of t [16]. A type s *conforms* [4] to a type t, denoted $s \preceq t$, if

1. s provides at least the operations of t, and
2. the types of the results of s's operations conform to the types of the results of the corresponding operations of t, and
3. the types of the arguments of t's operations conform to the types of the arguments of the corresponding operations of s.

Assuming a set of primitive types and their known conformity, any non-primitive type can be tested for conformity by recursively examining the types referenced by signatures of their methods until only primitive types remain. We extend the notion of conformity to apply to objects as well as types in the following manner. We say 'object o conforms to type t' to mean the same as 'object o is an instance of a type which conforms to type t'.

2.2 Types and Queries

Black et. al. [4] show how the conformity relationship is sufficient for developing a type checking algorithm for expressions denoting single objects and variable assignment. As demonstrated in the introduction, object-oriented database query languages introduce a new problem in that the result of a query is a set of objects which may not be homogeneous. In this case, what can be said about the types that each member of the query result supports?

Example 2.1 Consider the fragment of a type hierarchy in Figure 2 where types are labeled t_i. Assume we wish to take the union of the instances of types t_8 and t_9. The following can be said about the objects in $(t_8 \cup t_9)$.

1. Some objects conform to t_3 (immediate supertype of t_8),
2. Some objects conform to t_5 (immediate supertype of t_9),
3. All objects conform to t_4 (immediate supertype of both t_8 and t_9).

Intuitively then, we may say that the type of $(t_8 \cup t_9)$ is t_4 since this is the only type that all objects in the union conform to. This case is somewhat trivial as all objects in the query result conform to just one class. Referring again to Figure 2, assume we wish to take the union of the instances of types t_{10} and t_{11}. In this case the following can be said about the objects in $(t_{10} \cup t_{11})$.

1. Some objects conform to $\{t_3, t_4, t_2\}$ (immediate supertypes of t_{10}),
2. Some objects conform to $\{t_5, t_6, t_7\}$ (immediate supertypes of t_{11}),
3. All objects conform to $\{t_1, t_2\}$ (not necessarily immediate supertypes).

The last statement holds because an object conforms to the type it is an instance of, and via inheritance, any of its supertypes. ◇

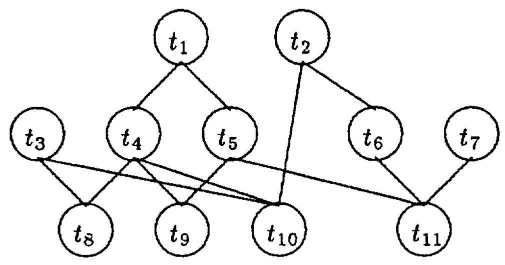

Figure 2: A type hierarchy fragment.

Definition 2.1 *Conformance:* A *conformance* is a set of types. A set of objects O has conformance $\{t_1, \ldots, t_n\}$, denoted by $O : \{t_1, \ldots, t_n\}$, when each object $o \in O$ conforms to every type $t_i \in \{t_1, \ldots, t_n\}$. □

Definition 2.2 *Conformance Inclusion Relationship:* The conformance inclusion relationship on two sets of types C_1 and C_2 is defined as $C_1 \sqsubseteq C_2$ iff $\forall t_i \in C_2, \exists t_j \in C_1 \mid t_j \preceq t_i$. In other words, $C_1 \sqsubseteq C_2$, if for every type in C_2 there is a conforming type in C_1. □

Note that C_1 may contain types which do not conform to any type in C_2 under this definition.

The notion of finding the set of types to which all members of a second set of types conforms to is central to determining the type consistency of operations on sets of objects. However, we do not always want to know all the types which are conformed to as this set would contain redundant information. In Example 2.1 the conformance of $(t_{10} \cup t_{11})$ was determined to be $\{t_1, t_2\}$. Including parents of t_1 and t_2 in the conformance would add no new type information since t_1 and t_2 define at least, if not more than, the behavior of their parents, i.e., t_1 and t_2 are specializations of their parent types. Similarly, placing more general types in the conformance, for example parents of t_1 and t_2 but not t_1 or t_2 themselves, introduces a loss of type information.

Loss of type information is undesirable when type checking a query. Consider again the type hierarchy fragment of Figure 2. Assume all objects in a query result conform to both t_{10} and t_{11} but the conformance was nonetheless specified as $\{t_1, t_2\}$. This would correspond to the case where types more general than necessary are placed into the conformance. It is possible that the query in question was just a subquery and that further operations are to be performed on its result. Some of the object algebra operators are qualified by predicates. One form of predicate involves applying a method to each member in the query set. If the method referenced in the query is defined on t_{11} but not on t_2, the query will fail during type checking when in fact each member of the query set does support that method. Thus we have the requirement that the conformance of a set of objects used in type checking include only the most specific types which satisfy the conformance definition.

Definition 2.3 *Most Specific Conformance:* The conformance of a set of objects O, $O\!:\{t_1, \ldots, t_n\}$, is defined to be the *most specific conformance* when there does not exist a subtype $s \preceq t_i$ such that all elements of O conform to s. □

The function $MSC(t_1, \ldots, t_n)$ is defined to return the most specific conformance of the types $t_1, \ldots, t_n$.

Example 2.2 Referring to Figure 2:

$$\begin{aligned} MSC(t_{10}) &\equiv \{t_{10}\} \\ MSC(t_{10}, t_{11}) &\equiv \{t_1, t_2\} \\ MSC(t_{10}, t_6) &\equiv \{t_2\} \ \diamond \end{aligned}$$

The need will arise during type checking to determine the inverse MSC relationship. Letting s and t refer to subtypes and types respectively, the function MSC^{-1} is defined as

$$MSC^{-1}(t_1, \ldots, t_n) \equiv \{\ s_1, \ldots, s_k \mid MSC(s_1, \ldots, s_k) = \{t_1, \ldots, t_n\}\ \}$$

In other words, the inverse function MSC^{-1} returns the most general set of subtypes all of whom conform to $t_1, \ldots, t_n$.

Example 2.3 Referring to Figure 2:

$$\begin{aligned} MSC^{-1}(t_1) &\equiv \{t_1\} \\ MSC^{-1}(t_1, t_2) &\equiv \{t_{10}, t_{11}\} \\ MSC^{-1}(t_5, t_7) &\equiv \{t_{11}\} \ \diamond \end{aligned}$$

Algorithms for MSC and MSC^{-1} have been developed but are omitted here due to space considerations.

3 Type Inference Rules

The notion of typing in object algebra expressions is formalized by providing a set of type inference rules. Although we call them type inference rules, they are really conformance inference rules since results of object algebra expressions are sets of possibly non-homogeneous objects. The rules determine the conformance of an expression from the conformance(s) of its subexpressions. The rules themselves do not imply a specific type checking mechanism. Instead, a type checking algorithm is considered correct if it computes types that are derivable by these rules. An expression is considered type inconsistent if the rules can not be used to derive a type (conformance) for all variables in the expression.

A syntax for inference rules similar to that of [8] will be used:

Rule Name:

$$\frac{X}{Y} \qquad (1)$$

where the horizontal line is a logic implication. If we can infer X, then we can infer Y. Variables are used in a consistent fashion to denote similar items in each of the rules. Upper case variables denote sets while lower case variables denote single entities. For example, O, P, Q, and R are object set variables while o, p, q, and r are object variables with the implication that $o \in O$, $p \in P$, etc. I is a set of conformance inclusion constraints and A is a set of conformance assumptions for free variables. C is a conformance variable denoting a set of types. $A.e:C$ is the set A extended with the assumption that expression e has conformance C. $A \vdash expr$ is an assertion meaning that from A we can infer *expr*. Table 1 summarizes the variable denotations.

The following predefined functions are used by the typing rules.

unique(m, t, C): This boolean function evaluates to *True* if $t \in C$ and method m is defined only on type t and not on any other type in C.

arg_type(m, t, i): This function returns the declared type of the i^{th} argument of method m on type t.

num_args(m, t): This function returns an integer representing the number of arguments required by method m defined on type t.

res_type(m, t): This function returns the declared result type of method m on type t.

We present some conformance inclusion rules in order to familiarize the reader with the rule format.

Top:

$$I \vdash C \sqsubseteq \{Root\} \tag{2}$$

This rule states that every conformance is related via inclusion $\sqsubseteq$ to the set containing just the *Root* type. This can be derived from the conformance inclusion definition and recognizing that all types are a subtype of the *Root* type.

Transitivity:

$$\frac{I \vdash C_1 \sqsubseteq C_2, C_2 \sqsubseteq C_3}{I \vdash C_1 \sqsubseteq C_3} \tag{3}$$

Provable by definition of the $\sqsubseteq$ relationship.

Reflexivity:

$$I \vdash C \sqsubseteq C \tag{4}$$

Provable by definition of the $\sqsubseteq$ relationship and reflexivity of $\preceq$.

We are now ready to develop a family of type inference rules for the object algebra. Section 4 first introduces rules for predicates of the object algebra. This is followed by Section 5 which develops rules for complete algebra expressions.

4 Predicates for Algebra Operators

Operands and results in the object algebra are sets of objects. Let Θ be an operator in the algebra and P, Q and Q_i denote sets of objects. The algebra contains both binary operators denoted by $P \Theta Q$ and n-ary operators denoted by $P \Theta \langle Q_1 \ldots Q_k \rangle$.

Some of the algebra operators are qualified by a predicate. Such operators will be written $P \Theta_F \langle Q_1 \ldots Q_k \rangle$ where F is a formula consisting of one or more atoms connected by $\wedge$, $\vee$ or $\neg$ using parenthesis as required. Atoms represent primitive operations of the data model which evaluate to a boolean. Atoms reference lower case, single letter variables which range over argument sets named with the corresponding upper case letter. For example, the object variables p, q_1 and q_2 in the predicate of $P \Theta_{F(p,q_1,q_2)} \langle Q_1, Q_2 \rangle$ range over the sets of objects denoted by P, Q_1 and Q_2 respectively.

Argument sets P and $Q_1 \ldots Q_k$ can be the results of previous algebra operations and thus have a conformance associated with them. This means that each $p \in P$ and $q_i \in Q_i$ conforms to each type in the parent set's conformance. In this sense then, each object variable has many types and we can associate a conformance with object variables as well as argument sets. In general, we will assume that an object variable's conformance is the same as that of the set from which it is drawn. In other words, if we use the notation $Q_i:C$ to mean that the conformance of set Q_i is C, then $Q_i:C \wedge q_i \in Q_i \Rightarrow q_i:C$.

4.1 Atoms and Query Primitives

Atoms, the building blocks of predicates, are defined as follows:

- $o_i \theta o_j$ where:
  - o_i and o_j are object variables or denote a method application on object variables.
  - θ is one of the operators $==$, $\in$ or $=_{\{\}}$.
- $a \theta o_i$ where:
  - o_i is an object variable or denotes a method application on object variables.
  - a is the textual representation of an atomic value or a set of atomic values.
  - θ is one of the operators $=$, $\in$ or $=_{\{\}}$.

The relational operators $==$, $\in$, and $=_{\{\}}$ are query primitives on object identity supported by the data model. The $==$ operator tests for object identity equality, i.e., $(o_i == o_j)$ evaluates to true when o_i and o_j denote the same object. The $\in$ operator can only be used with objects with a set value. Set valued objects have a value which is a set of object identifiers. For example, $(o_i \in o_j)$ evaluates to true when o_i's object

Table 1: Definition of variables used in the inference rules.

e	object algebra expression
f	predicate subexpression (i.e., *lhs* or *rhs* of an atom of the form (*lhs* θ *rhs*))
i, j, k	index variables and subscripts
m	method name variable
o, p, q, r	object variables
$F/[o_1 : C_1, \ldots]$	predicate F is a set of atoms connected by $\wedge$ and/or $\vee$ with all occurrences of o_i having conformance C_i, etc.
t	type variable
A	set of conformance assumptions for free variables
C	conformance variable
I	set of conformance inclusion constraints
O, P, Q, R	object set variables

identifier is contained within the set value of object o_j. The $=_{\{\}}$ operator is similar to $\in$ only it tests if two set valued objects are pairwise equal. For example, $(o_i =_{\{\}} o_j)$ evaluates to true when o_i and o_j are both set valued objects containing the same set of object identifiers. The $=$ operator applies to atomic objects only, i.e., objects whose value is drawn from a primitive domain supported by the database system such as integer, string, etc. For example, ("33" $= o_i$) evaluates to true when o_i is an atomic object whose value component is the integer "33".

The dot notation $<o_1 \ldots o_n>.m_1.m_2 \cdots m_m$ is used to denote method application and method composition. For example, assume methods m_1 and m_m take three arguments each, and method m_2 takes 2 arguments, then Figure 3 illustrates the processing denoted by $<o_1 \ldots o_n>.m_1.m_2 \cdots m_m$. Method m_1 is applied to objects $<o_1, o_2, o_3>$ resulting in object r_1, method m_2 is applied to objects $<r_1, o_4>$ returning object r_2, and so on until the final result object, r_m, is obtained by applying method m_m to objects $<r_{m-1}, o_{n-1}, o_n>$. Note that the dot notation denotes method application and composition, not the traditional record field selection (attribute selection) as in [3, 7, 13]. $<o_1 \ldots o_n>$.*mlist* will be used when the list of method names is unimportant.

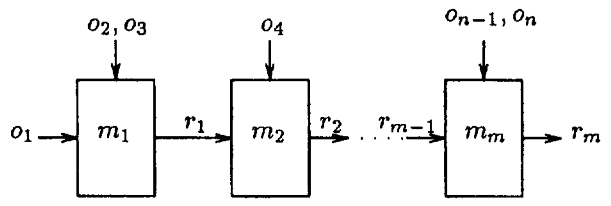

Figure 3: Composition of method applications.

Example 4.1 Let p, q and r be object variables. Then the following are examples of legal atoms and their semantics:

1. $(p == q)$ – Are the objects denoted by p and q the same object?

2. $(p \in <q, r>$.*mlist*) – Is the identifier of p contained in the set value of the object obtained by applying the methods in *mlist* to the objects $<q, r>$?

3. ($<p, q>$.*mlist* $=_{\{\}} r$) – Is the set value of the object obtained by applying the methods in *mlist* to the objects $<p, q>$ pairwise equal to the set value of the object denoted by r?

4. ("59" $= p$) – Is the atomic value of the object denoted by p "59"?

5. ("59" $\in p$) – Does the set value of the object denoted by p include an identifier for the object whose atomic value is "59"?

6. ({"59", "61"} $=_{\{\}} <p, q, r>$.*mlist*) – Does the set value of the object obtained by applying the methods in *mlist* to the objects $<p, q, r>$ contain only two identifiers for objects whose atomic values are "59" and "61"? $\diamond$

4.2 Predicate Typing Rules

As stated above, predicates qualify algebra operators and are composed of legal atoms connected by $\wedge$, $\vee$ and $\neg$. Each atom is of the form (*lhs* θ *rhs*) where *lhs* and *rhs* are either an object variable, a constant value, or a method application on object variables and $\theta \in \{=, ==, \in, =_{\{\}}\}$. The following predicate typing rules can be used to determine the type consistency of object variables when used as either the *lhs* or *rhs* of an atom.

Defining Set:

$$\frac{I, A \vdash O{:}C}{I, A \vdash o{:}C} \qquad (5)$$

This rule states that an object variable conforms to the same types as the set from which it is drawn. This is a restatement of the implication given earlier: $O{:}C \wedge o \in O \Rightarrow o{:}C$.

This rule is fundamental to the notion of type consistency. While arguments to algebra operators are sets of objects, predicates on algebra operators reference individual object variables. Since at query execution time the predicate is evaluated once for each object in the argument sets[1], the entire algebra operator is type consistent only if the predicate is type consistent for each type which exists in the argument sets, i.e., the conformance.

The next two rules determine the result type of a sequence of method applications. The first rule, for single methods only, insures that the types of the arguments to the method match those specified in its signature. The second rule recursively determines the result type of a sequence of method applications of length n by defining itself in terms of a sequence of methods of length $n-1$. When $(n-1) = 1$, the recursion terminates and the first rule is applied.

Single Method:

$$\frac{I, A \vdash \left[\begin{array}{l} \mathit{unique}(m, t, C_1), \\ \mathit{num\_args}(m, t) = k - 1, \\ C_i \sqsubseteq \{\mathit{arg\_type}(m, t, i)\}, 2 \leq i \leq k \end{array}\right.}{I, A \vdash <o_1 : C_1 \ldots o_k : C_k>.m : \{\mathit{res\_type}(m, t)\}} \qquad (6)$$

This rule determines the conformance of the result of a single method application. The method application is legal if three conditions are met:

1. $\mathit{unique}(m, t, C_1)$ insures that there exists a type t which is a member of conformance C_1 and that method m is defined only on t and not on any other members of C_1. This restriction insures that there is no ambiguity as to which type's method m is to be applied.

2. $\mathit{num\_args}(m, t) = k - 1$ insures that the signature of method m on type t requires the same number of parameters $(o_2 \ldots o_k)$ as are provided in the operand list of the method application.

3. $C_i \sqsubseteq \{\mathit{arg\_type}(m, t, i)\}, 2 \leq i \leq k$ insures that the types of the i^{th} argument, represented by the conformance C_i, are subtypes (and therefore substitutable) of the type stipulated by the signature of method m on type t.

[1]Actually, the predicate is evaluated once for each element in the cross product of the argument sets.

If all of the above conditions are met, then the conformance of the result of the method application is the singleton set containing the result type as specified by the signature of m on t.

Multiple Methods:

$$\frac{I, A \vdash \left[\begin{array}{l} <o_1 : C_1 \ldots o_{j-1} : C_{j-1}>.m_1 \cdots m_{n-1} : \{t\}, \\ \mathit{unique}(m_n, t, \{t\}), \\ \mathit{num\_args}(m_n, t) = k - j + 2, \\ C_i \sqsubseteq \{\mathit{arg\_type}(m_n, t, i)\}, j \leq i \leq k \end{array}\right.}{\begin{array}{r} I, A \vdash <o_1 : C_1 \ldots o_j : C_j \ldots o_k : C_k>.m_1 \cdots m_n : \\ \mathit{res\_type}(m_n, t)\} \end{array}} \qquad (7)$$

This recursive rule determines the conformance of a sequence of method applications based on the conformance of the sequence which applies methods m_1 through m_{n-1}. The multi-operation

$$<o_1 \ldots o_k>.m_1 \cdots m_n$$

is considered to be logically equivalent to the multi-operation

$$<o_1 \ldots o_j>.m_1 \cdots m_{n-1}$$

followed by the single operation

$$<\mathit{res}, o_{j+1} \ldots o_k>.m_n$$

where *res* is the result of the first multi-operation. The first condition of rule 7 stipulates that the conformance of the multi-operation which applies methods $m_1 \cdots m_{n-1}$ is known. This conformance is denoted as the singleton set $\{t\}$ since the signature of method m_{n-1} defines a single result type. The second condition insures that method m_n is actually defined on type t, an ancillary result of the ***unique*** function. The third condition guarantees that the proper number of arguments are present for method m_n while the last condition insures that each argument conforms to the types declared by m_n's signature.

If all of the above conditions are met, then the conformance of the result of the sequence of method applications $m_1 \cdots m_n$ is the set containing the result type of method m_n on type t.

Object Identity:

$$\frac{I, A \vdash f{:}C}{I, A \vdash o{:}C == f{:}C} \qquad (8)$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o == f$ implies that o has conformance C as well. This should make sense intuitively based upon the meaning of the $==$ relationship. If two expressions are identity equal, i.e., denote the same object, then they conform to the same types.

Set Inclusion:

$$\frac{I, A \vdash f : C}{I, A \vdash o : \{Root\} \in f : C} \tag{9}$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o \in f$ implies that o conforms to the *Root* type. This apparent loss of type information is due to the fact that the data model does not include a parametrically polymorphic set type definition operator such as Set[t] [16]. The data model supports only 'generic' set valued objects which make no restrictions on the type of the objects in the set.

Set Equivalence:

$$\frac{I, A \vdash f : C}{I, A \vdash o : \{Root\} =_{\{\}} f : C} \tag{10}$$

This rule states that if there is a predicate subexpression f whose conformance is known to be C, then the use of f in the atom $o =_{\{\}} f$ implies that o conforms to the *Root* class. The reasoning is the same as in rule 9.

Atom Disjunction:

$$\frac{I, A \vdash F_1/[o : C_1], F_2/[o : C_2]}{I, A \vdash (F_1 \vee F_2)/[o : MSC(C_1 \cup C_2)]} \tag{11}$$

This rule states that if o has conformance C_1 for all occurrences in predicate F_1 and conformance C_2 for all occurrences in predicate F_2, then the conformance of o in the disjunction $(F_1 \vee F_2)$ is $MSC(C_1 \cup C_2)$. Consider that F_1 and F_2 both independently define a set of types for o. Their disjunction then implies that o represents objects which conform to types in C_1 **or** types in C_2. This is similar to the case of Example 2.1. The only statement one can make about all instances of o in the disjunction is that they conform to the most specific common ancestors of types in C_1 and C_2 which is given by $MSC(C_1 \cup C_2)$.

Atom Conjunction:

$$\frac{I, A \vdash F_1/[o : C_1], F_2/[o : C_2]}{I, A \vdash (F_1 \wedge F_2)/[o : MSC^{-1}(C_1 \cup C_2)]} \tag{12}$$

This rule states that if o has conformance C_1 for all occurrences in predicate F_1 and conformance C_2 in all occurrences of predicate F_2, then the conformance of o in the conjunction $(F_1 \wedge F_2)$ is $MSC^{-1}(C_1 \cup C_2)$. The MSC^{-1} can be rationalized as follows. Consider that F_1 and F_2 both independently define a set of types for o. Their conjunction then implies that o represents objects which conform to types in C_1 **and** types in C_2. Clearly, only subtypes which inherit from all types in C_1 and C_2 can conform in this manner. $MSC^{-1}(C_1 \cup C_2)$ determines that set of types.

The previous two rules, atom disjunction and atom conjunction, are important results. They show that the manner in which atoms are combined in a predicate affects whether type information is lost (disjunction) or gained (conjunction). Type information can be lost in the case of disjunction since the inference rule derives a conformance for the variable in question which contains types which are more general, i.e., higher in the type hierarchy. Type information can be gained in the case of conjunction since the inference rule derives a conformance for the variable in question which contains types which are more specific, i.e., lower in the type hierarchy.

5 The Object Algebra and its Typing Rules

This section presents the algebra operators and their associated typing rules. First some general inference rules for algebra expressions are needed.

Top:

$$I, A \vdash e : \{Root\} \tag{13}$$

This rule states that all object algebra expressions minimally conform to the type $\{Root\}$. This is clear since algebra expressions denote sets of objects and all objects conform to the *Root* type.

Transitivity:

$$\frac{I, A.e : C_1 \vdash C_1 \sqsubseteq C_2}{I, A \vdash e : C_2} \tag{14}$$

This rule is the same as the conformance inclusion transitivity rule applied to algebra expressions.

Basis:

$$I, A \vdash instances\_of(t) : \{t\} \tag{15}$$

Leaves of object algebra expression trees denote all instances of some type t in the database. This rule states that since all objects in $instances\_of(t)$ conform to t, the conformance of a leaf node is $\{t\}$. This rule is called **Basis** since the only type information initially available in a query is the types it references explicitly. Just as query processing proceeds from the leaves of the query tree to the root, one can think of type inference as proceeding from the leaves to the root as well.

The union operator, denoted $P \cup Q$, returns the set union of P and Q.

Union:

$$\frac{I, A \vdash P : C_1, Q : C_2}{I, A \vdash (P \cup Q) : MSC(C_1 \cup C_2)} \tag{16}$$

This rule states that the conformance of a union operation is the MSC of types contained in the conformances of its operands. The reasoning is that $MSC(C_1 \cup C_2)$ denotes the most specific types to which **all** members of $(P \cup Q)$ conform.

The difference operator, denoted $P - Q$, returns those objects which are in P and not in Q.
Difference:

$$\frac{I, A \vdash P : C_1, Q : C_2}{I, A \vdash (P - Q) : C_1} \tag{17}$$

This rule states that the conformance of a difference operation is the conformance of the first operand. This should be clear as the result of a difference is a subset of the first operand.

The select operator, denoted $P \; \sigma_F \; \langle Q_1 \ldots Q_k \rangle$, returns the p elements of each vector $<p, q_1 \ldots q_k> \in P \times Q_1 \times \cdots \times Q_k$ which satisfies the predicate F.
Select:

$$\frac{I, A \vdash F/[p : C'_p, q_1 : C'_1 \ldots q_k : C'_k]}{I, A \vdash (P : C_p \; \sigma_F \; \langle Q_1 : C_1 \ldots Q_k : C_k \rangle) : C'_p} \tag{18}$$

Here F denotes the predicate of the select operation. The rule states that if the input sets P, $Q_1 \ldots Q_k$ have conformances $C_p, C_1, \ldots, C_k$ respectively, then the result of the select operation has conformance C'_p as derived for occurrences of p in predicate F. Since F may have multiple atoms connected by $\wedge$ and/or $\vee$, the atom conjunction and atom disjunction rules may determine conformances for variables in F (C'_p, C'_1, etc.) which are different from the conformances of the input argument sets (C_p, C_1, etc.). This allows for predicates which restrict or enhance the types of p.

The generate operator, denoted $Q_1 \; \gamma_F^r \; \langle Q_2 \ldots Q_k \rangle$, returns the objects represented by r in predicate F for each vector $<q_1 \ldots q_k> \in Q_1 \times \cdots \times Q_k$. In other words, r does not range directly over one of the argument sets. Instead it represents objects which are returned by method applications or other operations in F.
Generate:

$$\frac{I, A \vdash F/[r : C'_r, q_1 : C'_1 \ldots q_k : C'_k]}{I, A \vdash (Q_1 : C_1 \; \gamma_F^r \; \langle Q_2 : C_2 \ldots Q_k : C_k \rangle) : C'_r} \tag{19}$$

Similar to the rule for select expressions, this rule states that the result of a generate operation has the same conformance as that derived for variable r in the predicate F, C'_r.

The map operator, denoted $Q_1 \mapsto_{mlist} \langle Q_2 \ldots Q_k \rangle$, is a special case of the generate operator whose predicate is the multi-operation $r == <q_1 \ldots q_k>.mlist$.
Map:

$$\frac{I, A \vdash <q_1 : C_1 \ldots q_k : C_k>.mlist : \{t\}}{I, A \vdash (Q_1 : C_1 \mapsto_{mlist} \langle Q_2 : C_2 \ldots Q_k : C_k \rangle) : \{t\}} \tag{20}$$

This rule states that the result of a map operation has the same conformance as that derived for the multi-operation $<q_1 \ldots q_k>.mlist$. expression results based on the conformances of the argument sets.

6 Conclusion

We presented a theory of type consistency for queries in object-oriented database systems. This issue arises when the object data model allows multiple inheritance and the query language does not restrict the types of the inputs to a query. These two conditions are necessary for the development of languages with sufficient expressive power, but they cause retrieval of non-homogeneous sets of objects as a result. If methods are to be applied to the retrieved objects, it is necessary to determine their type, and thus their applicable methods.

The type checking theory presented in this paper follows the approach proposed in [8]. We demonstrate the theory on an object algebra which is object-preserving [15, 18]. We do not address the problem of determining the types of newly created objects by object-creating operators (e.g., join and project in [16]). We have chosen to restrict our study to object-preserving algebras since our fundamental aim is to investigate query processing issues, not the development of a full language.

References

[1] M. Astrahan. System R: A Relational Approach to Data. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[2] M. Atkinson and P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[3] J. Banerjee, W. Kim, and K. Kim. Queries in Object-Oriented Databases. In *Proc. 4th Int'l. Conf. on Data Engineering*, pages 31–38, February 1988.

[4] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[5] A. Borgida. Class Hierarchies in Information Systems: Sets, Types, or Prototypes. In M. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistance*, chapter 10, pages 137–154. Springer Verlag, 1988.

[6] V. Breazu-Tannen, P. Buneman, and A. Ohori. Static Type-Checking in Object-Oriented Databases. *Quart. bull. of the IEEE TC on Data Engineering*, 12(3):5–12, September 1989.

[7] L. Cardelli. A Semantics of Multiple Inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer Verlag, 1984.

[8] L. Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[9] S. Danforth and C. Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[10] T. Despeyroux. Executable Secification of Static Semantics. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Tupes*, volume 173 of *Lecture Notes in Computer Science*, pages 215–232. Springer Verlag, 1984.

[11] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2):112–152, June 1984.

[12] W. R. LaLonde, D. A. Thomas, and J. R. Pugh. An Exemplar Based Smalltalk. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 322–330, 1986.

[13] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 171–182, September 1986.

[14] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proc. of the Object-Oriented Programming Systems and Languages Conference*, pages 472–482, July 1986.

[15] M. Scholl and H. Schek. A Relational Object Model. Unpublished manuscript, 1990.

[16] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

[17] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 340–355, May 1986.

[18] D. Straube and M. T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *Submitted to ACM Transactions on Information Systems.* Also available as University of Alberta Computing Science Technical Teport TR 90-11, April 1990.

[19] S. B. Zdonik. Data Abstraction and Query Optimization. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 368–373. Springer Verlag, 1988.