A Type System for an Object-Oriented Database System

Xuequn Wu
Software-Technology, Department of Computer Science
University of Dortmund
P.O. Box 50 05 00, D-4600 Dortmund 50

Abstract

As object-oriented data models have increased their expressiveness, object-oriented database systems have got large schema and complex structures, which leads to difficulties in providing closure property on the one hand, and is prone to lead into type errors in database application programs on the other hand. In this paper we present a type system with strong typing and static type checking that is not yet well supported in most existing object-oriented database systems. We define a subtyping rule for correctly structuring the inheritance hierarchy of the types. Based on the subtyping, a number of type inference rules are defined. They can be used by the type system to statically determine the types of the query results and whether a given application program is type correct.

1 Introduction

InORM is a prototype database system being developed at the university of Dortmund. The data model for InORM is an integration of the relational and the object-oriented data model [11, 12].

One important motivation of designing InORM is strong typing and static type checking, which is not yet well supported in most existing object-oriented database systems. Strong typing and static type checking enables that all expressions can be verified type correct at compile time, without dynamic type checking at run-time. As object-oriented data models have increased their expressiveness, object-oriented database systems have got large schema and complex structures, which is prone to lead into type errors in database application programs. Thus a type system in which such errors can be discovered by a static analysis of the program should be a prerequisite for an object-oriented database system.

In order to explore inheritance by strong typing a subtyping rule is defined for the type system of InORM. While the subtype ordering on tuples is defined similarly to the approach given in [3], we define subtype ordering on types considering not only structures but also behaviors of types. According to the subtyping rule types in the type system are correctly structured in a type hierarchy. Static type checking is carried out with the aid of a type inference mechanism that consists of a number of type inference rules (they are also called typing rules). With these rules the type system can infer types for expressions that appear in queries and find types for results of queries. Accordingly, it can be determined which methods could be correctly applied on the expressions or the results of queries. Thus the type system can statically determine whether a given expression or query is type correct.

In comparison with InORM neither systems based on a Smalltalk[6] like approach such as GemStone [8] nor systems based on Lisp as ORION [1] provide static type checking. POSTGRES[10] and EXODUS [4] are typical ones that extend the relational approach. While the result of a query in InORM is a new collection with safe type determined statically by the type system, the results of queries in POSTGRES and EXODUS could be type-loss. That is, such systems do not maintain the closure property. Moreover, in most existing object-oriented database systems there are some features that prevent strong typing and static type checking, e.g. retyping of objects in Iris [5] and exceptional attributes in O₂ [7].

The paper is organized as follows: section 2 gives an informal description of types and type hierarchy of the type system of InORM, where some basic concepts are introduced. In section 3 we define a subtyping rule for inheritance, which is defined using a partial ordering on type structures and on methods associated to types. Based on the concepts defined in section 3, the type inference rules are defined in section 4, where the late binding is also mentioned shortly. Conclusion is given in section 5.
2 Types and type hierarchy

All conceptual entities are modeled as objects. Every object has a system-created surrogate that is unique and independent from the value of the object. The type concept is considered only in coherence with data abstraction. That is, the common properties and behaviors of the objects are described with abstract data types, and an object is then an instance of a type. There are three sorts of types: The predefined types, the user-defined abstract data types (UADTs) and the complex types.

The predefined types include primitive types and parameterized types which are Tuple, Set and List [11]. The primitive types are NUMBER, CHAR and LONGFIELD, which have predefined methods. The type LONGFIELD is offered to handle large data objects as well as unformatted data such as images and software documents. The parameterized types have fixed structures and methods. The parameters to be chosen for a tuple type \( [A_1:T_1, \ldots, A_n:T_n] \) are the number \( n \) of the attributes, their names \( A_i \) and their types \( T_i \). There are predefined methods project and join for the tuple type. The parameter to be chosen for a set type \( \{T\} \) is the type \( T \) for the elements of a instance of the set type. The interface of a set type contains the usual set operations such as union, intersection and difference. Specially, if the type \( T \) for a set type is a tuple type then that set type is called a relation type. A list type \( <T> \) has the parameter \( T \) that is the type for the elements in an instance of that list type and provides the methods list concatenation, list element access etc. If it is necessary, other useful parameterized types such as array and graph could be easily added into the system.

The UADT is similar to the abstract data types defined in POSTGRES, which can be introduced to the database system by the user. In comparison with predefined ones there must be mechanisms to program new methods outside the database system and to register them into the database system, but the description about that is beyond the area of this paper.

With parameterized types users can define new and stronger types, and in this case the interface of the parameterized type may be augmented through several user-defined methods. A so defined type is called a complex type that can be used to define types for complex objects. With respect to complex types, primitive types and UADTs are called basic types. In example 2.1 we define some complex types.

Example 2.1

```
DEFINE TYPE Person = TUPLE (name CHAR, birthday Date, address CHAR);
```

```
DEFINE METHOD Address FOR Person:
Person -> CHAR;
```

```
DEFINE TYPE Expertise = LIST OF (DESIGNATE Topic);
```

```
DEFINE TYPE Department = TUPLE (dname CHAR, members REFERENCE Empl_Set);
```

```
DEFINE METHOD Address FOR Department:
Department -> CHAR;
```

The type Empl_Set used in the definition of Department will be defined with set type later on. It means that the type could be defined with those types which have not yet been defined. This allows recursive type definitions. We assume that the type Topic used here has already been defined. Both Person and Department have a user-defined method Address. That means different methods for different types could have the same name, which will be further discussed in the following sections. The reference specifications such as DESIGNATE and REFERENCE seen here will not be treated in this paper and are described elsewhere [11].

A type may be declared to be a subtype of another type. In that case, all instances of the subtype are also instances of the supertype, i.e. there is a containment relation between the supertype and the subtype. It follows that all properties and methods defined on the supertype are inherited by the subtype, that facilitates strong typing. Though the methods can be refined as we will see below. Some examples for the definition of subtypes are given in example 2.2.

Example 2.2

```
DEFINE TYPE Employee = TUPLE (dept REFERENCE Department, exper PART Expertise)
SUBTYPE OF Person;
```

```
DEFINE METHOD Address FOR Employee:
Employee -> CHAR;
```

```
DEFINE TYPE Empl_Set = SET OF (PART Employee);
```

```
DEFINE TYPE Student = TUPLE (enrollment NUMBER)
SUBTYPE OF Person;
```

```
DEFINE METHOD Address FOR Student:
Student -> CHAR;
```

Similar to GemStone and EXODUS, we distinguish between type and collections of instances of that type and queries are then carried out over these collections. After a type has been defined, a collection of instances of that type can be declared and placed in a relation. In example 2.3 we declare collections for types Person, Student and Employee.
3 Type Inheritance

In order to define inheritance of types we first give a definition for types. Then we define a partial ordering on type structures and on methods in section 3.1 and section 3.2 respectively. After that, these partial orderings are used to define the subtype ordering on types.

We assume that \( B \) is the set of names for all basic types, \( C \) is the set of names for all complex types, and \( T \) is the union of \( B \) and \( C \). \( M \) is a set of all methods defined for all types. \( A \) is a set of names for all attributes. We define now the types as follows:

Definition 3.1

1. A basic type is denoted as a pair \((t, m)\) where \( t \) is an element of \( B \) and \( m \) a subset of \( M \).
2. A tuple type is denoted as a triple \((t, [a_1 : t_1, \ldots, a_n : t_n], m)\) where \( t \) is an element of \( C \), \( a_1, \ldots, a_n \) are elements of \( A \), \( t_1, \ldots, t_n \) are elements of \( T \), and \( m \) a subset of \( M \).
3. A set type is denoted as a triple \((t, \{t'\}, m)\) where \( t \) is an element of \( C \), \( t' \) is an element of \( T \), and \( m \) a subset of \( M \).
4. A list type is denoted as a triple \((t, < t' >, m)\) where \( t \) is an element of \( C \), \( t' \) is an element of \( T \), and \( m \) a subset of \( M \).

3.1 Structures of types

A type consists of a structure part and a methods part. Given a type \( t \), its structure part and methods part will be denoted by \( \text{Struct}(t) \) and \( \text{Methods}(t) \) respectively. We will define methods more formally in section 3.2. We define now the structure of a type:

Definition 3.2

1. For a basic type \((t, m)\) we have: \( \text{Struct}(t) = t \).
2. For a tuple type \((t, [a_1 : t_1, \ldots, a_n : t_n], m)\) we have: \( \text{Struct}(t) = [a_1 : t_1, \ldots, a_n : t_n] \).
3. For a set type \((t, \{t'\}, m)\) we have: \( \text{Struct}(t) = \{t'\} \).
4. For a list type \((t, < t' >, m)\) we have: \( \text{Struct}(t) = < t' > \).

As mentioned in section 2 there is a containment relation between the instances of a type and its subtypes. It implies that if a database object has a certain structure then any more defined (or “better defined”) objects also have the structure. So there is a ordering on type structures. We define a syntactic relation \( \leq_{\text{str}} \) on type structures to represent this ordering.

Definition 3.3

Let \( t \) and \( t' \) be two types, and \( s = \text{Struct}(t) \), \( s' = \text{Struct}(t') \), then:

1. if \( s = t \), i.e. \( t \) is a basic type, then \( s \leq_{\text{str}} s \);
2. if \( s = [a_1 : t_1, \ldots, a_n : t_n], s' = [a'_1 : t'_1, \ldots, a'_m : t'_m] \), \( m \leq n \), and for all \( 1 \leq i \leq m \)
   \( \text{Struct}(t_i) \leq_{\text{str}} \text{Struct}(t'_i) \) then \( s \leq_{\text{str}} s' \);
3. if \( s = \{t_1\}, s' = \{t'_1\} \) and \( \text{Struct}(t_1) \leq_{\text{str}} \text{Struct}(t'_1) \) then \( s \leq_{\text{str}} s' \);
4. if \( s = < t_1 >, s' = < t'_1 > \) and \( \text{Struct}(t_1) \leq_{\text{str}} \text{Struct}(t'_1) \) then \( s \leq_{\text{str}} s' \).

Note that the assumed order of the attributes in 2. is not essential and only for convenience of talking. It is easy to check that \( \leq_{\text{str}} \) is a partial ordering. In the items 2, 3, 4 of the definition the \( \leq_{\text{str}} \) is used recursively and this process will be terminated when item 1 is met.

For example we have:

\[
\begin{align*}
\text{Struct(Employee)} & \leq_{\text{str}} \text{Struct(Person)} \\
\text{Struct(Student)} & \leq_{\text{str}} \text{Struct(Person)} \\
\text{Struct(Empl.Set)} & \leq_{\text{str}} \text{Struct(Person.Set)}
\end{align*}
\]

3.2 Methods of types

In this section we define methods of types and a partial ordering on methods. We have used \( M \) for all methods, we assume now \( MN \) is a set of all names for methods. It is necessary to introduce \( MN \), because two different methods could have the same name. We define now methods as follows:

Definition 3.4

A method \( m \) is denoted as a triple \((n, tb, sig)\) where \( n \) is an element of \( MN \), \( sig \) is a signature, and \( tb \) is an element of \( T \) and the base type of \( m \), which indicates that \( m \) is defined on the type \( tb \).

The name and the signature of a method \( m \) is denoted by \( \text{Name}(m) \) and \( \text{Sig}(m) \) respectively. A signature possesses the following form: \( t_1, \ldots, t_n \rightarrow t \).
where \( t_1, \ldots, t_n, t \) are elements of \( T \). The base type \( t \) is usually the same as \( t_1 \) in the signature, or is the same as \( t \) when by the method that creates a new object. Some signatures of methods were seen in the preceding examples.

Now we assume that the following methods are defined:

\[
\begin{align*}
m_1 &= (\text{Address}, \text{Person}, \text{sig}_1) \\
m_2 &= (\text{Address}, \text{Employee}, \text{sig}_2) \\
m_3 &= (\text{Address}, \text{Student}, \text{sig}_3) \\
m_4 &= (\text{Address}, \text{Department}, \text{sig}_4)
\end{align*}
\]

We see that though the methods are defined on different types, they have the same name. This is called overloading in the object-oriented paradigm \([2]\). We show now the signatures of these methods:

\[
\begin{align*}
\text{sig}_1 &= \text{Person} \rightarrow \text{CHAR} \\
\text{sig}_2 &= \text{Employee} \rightarrow \text{CHAR} \\
\text{sig}_3 &= \text{Student} \rightarrow \text{CHAR} \\
\text{sig}_4 &= \text{Department} \rightarrow \text{CHAR}
\end{align*}
\]

We see that the types in \( \text{sig}_2 \) are a refinement of those in \( \text{sig}_1 \) (structural refinement as defined above). So there is a refinement relation between such signatures. We define a syntactic ordering \( \leq_{\text{str}} \) on method signatures to represent this relation.

**Definition 3.5**

Let \( s \) and \( s' \) be two signatures and

\[
\begin{align*}
s &= t_1, \ldots, t_n \rightarrow t \\
s' &= t'_1, \ldots, t'_n \rightarrow t'
\end{align*}
\]

\( s \leq_{\text{str}} s' \) holds, if \( \text{Struct}(t_i) \leq_{\text{str}} \text{Struct}(t'_i) \) and \( \text{Struct}(t) \leq_{\text{str}} \text{Struct}(t') \) for \( 1 \leq i \leq n \).

Note that, \( \text{Struct}(t') \leq_{\text{str}} \text{Struct}(t) \) must be fulfilled, not the other way round. The reason will be given below. Looking at the signatures in the above example, the following relations hold: \( \text{sig}_2 \leq_{\text{str}} \text{sig}_1 \) and \( \text{sig}_3 \leq_{\text{str}} \text{sig}_1 \), but \( \text{sig}_4 \) has no such a relation to the others.

The relation \( s \leq_{\text{str}} s' \) could be exploited twofold. First it can be used to recognize if a method with signature \( s \) is a redefinition (or reimplementation) of a method with signature \( s' \). For example, the method \( \text{Address} \) on \( \text{Employee} \) reimplements the method \( \text{Address} \) on \( \text{Person} \) through overriding. Secondly, it means that a method having signature \( s' \) can be applied with the parameters that are specified by the signature \( s \). That is why \( \text{Struct}(t') \leq_{\text{str}} \text{Struct}(t) \) must be fulfilled. It gives safe types for methods, that allows method inheritance. That is, the objects of a subtype can be used in the place where a supertype of that type is expected.

### 3.3 Subtyping

We have defined structures and methods of types. Now we use the ordering \( \leq_{\text{str}} \) on structures and the ordering \( \leq_{\text{sig}} \) on signatures of methods to define the subtype ordering \( \leq \) on types.

**Definition 3.6**

Let \( t \) and \( t' \) be two types. \( t \) is a subtype of \( t' \) (denoted by \( t \leq t' \)) if

1. \( \text{Struct}(t) \leq_{\text{str}} \text{Struct}(t') \).
2. For all \( m' \) in \( \text{Methods}(t') \) there exists \( m \) in \( \text{Methods}(t) \) such that \( \text{Name}(m') = \text{Name}(m) \) and \( \text{Sig}(m) \leq_{\text{sig}} \text{Sig}(m') \).

We see that the inheritance of types is defined with the subtype ordering on types. The first item in the definition defines inheritance of structures. The second item defines inheritance of methods, where we see two things. First, \( \text{Names}(\text{Methods}(t')) \subseteq \text{Names}(\text{Methods}(t)) \), i.e. the subtype \( t \) inherits all methods of the supertype \( t' \) while \( t \) could have more defined methods than \( t' \). Secondly, for each method \( m' \) of \( t' \) there exists a method \( m \) of \( t \), such that \( m \) is either the same as \( m' \) or \( m \) is a reimplementation of \( m' \) through overriding.

In the preceding examples we have shown the following relations hold:

\[
\begin{align*}
\text{Struct}(\text{Employee}) &\leq_{\text{str}} \text{Struct}(\text{Person}) \\
\text{Struct}(\text{Student}) &\leq_{\text{str}} \text{Struct}(\text{Person})
\end{align*}
\]

and

\[
\begin{align*}
\text{sig}_2 &\leq_{\text{sig}} \text{sig}_1, \text{sig}_3 &\leq_{\text{sig}} \text{sig}_1, \text{sig}_4 &\leq_{\text{sig}} \text{sig}_2.
\end{align*}
\]

So the following subtype ordering holds:

\[
\begin{align*}
\text{Employee} &\leq \text{Person} \quad \text{and} \quad \text{Student} \leq \text{Person}.
\end{align*}
\]

### 4 Type inference

The collections of objects in the database are typed. The queries over the collections use the type hierarchy in the type system as the database schema. New objects are created as the result of queries, or as temporary collections in the middle of query evaluation. However, not all expressions in queries are meaningful. One goal of the type system is to identify all syntactically meaningful expressions and to determine right types to these expressions. In this section we introduce the type inference mechanism that consists of a number of typing rules for type inference.

We focus our attention to the expressions concerning tuples and relations, because, as shown in section 2, collections of objects in the database have relation
type. Nevertheless, the type of an attribute of a tuple could be any type in $T$. The typical expressions that appear in queries will be introduced. For every expression we give the corresponding typing rule.

We use $o$ to denote an object in the database. In order to represent incomplete information, an object $o$ could be (represented with) NULL. Expressions for tuples are defined as

$$
\text{Exp1 } o \text{ is an expression, if } o \text{ has type } t \text{ and } t \in T.
$$

$$
\text{Exp2 } [A_1 : \text{exp}_1, \ldots, A_n : \text{exp}_n] \text{ is an expression if }\text{exp}_1, \ldots, \text{exp}_n \text{ are expressions and } A_1, \ldots, A_n \in A, \text{ where } A_1, \ldots, A_n \text{ are all distinct.}
$$

A expression defined in Exp2 will also be called a tuple expression. The typing rule for tuple expressions is defined as follows:

**Rule1** $[A_1 : \text{exp}_1, \ldots, A_n : \text{exp}_n] : [A_1 : t_1, \ldots, A_n : t_n]$ if $\text{exp}_i : t_i$ for $1 \leq i \leq n$.

This typing rule determines essentially only a type structure. Using the subtype ordering on types, we define the following typing rule:

**Rule2** If $\text{exp} : t$ and $t \leq t'$ then $\text{exp} : t'$.

This rule is valid not only for tuple expressions but also for other expressions.

To support polymorphism, every method should apply to as many types as possible. The methods natural join ($\Join$) and project ($\pi$) are applied both on tuple and on relation. We define now the following rule for natural join on tuples:

**Exp3** $\text{exp}_1 \Join \text{exp}_2$ is an expression if $\text{exp}_1$ and $\text{exp}_2$ are tuple expressions.

We have then the following typing rule:

**Rule3** $\pi A_1, \ldots, A_m (\text{exp}) : [t_{i_1}, \ldots, t_{i_m}]$ if $\text{exp} : [\ldots, A_{i_1} : t_{i_1}, \ldots, A_{i_m} : t_{i_m}, \ldots]$.

We now discuss typing rules for sets. First we give the following rules.

**Exp5** $\{\text{exp}_1, \ldots, \text{exp}_n\}$ is an expression if $\text{exp}_1, \ldots, \text{exp}_n$ are expressions.

**Rule5a** $\{\text{exp}_1, \ldots, \text{exp}_n\} : \{t\}$, if $\text{exp}_1 : t, \ldots, \text{exp}_n : t$.

The Rule5a defines the typing rule for such sets whose element expressions all have the same type. If $\text{exp}_1, \ldots, \text{exp}_n$ are tuple expressions then the expression is called a relation expression. The case that a set has different typed element expressions will be discussed later on.

The rules for natural join and project on relations can be defined similarly to those on tuples and will not be shown here.

We define now rules for expressions concerning difference ($-$), intersection ($\cap$) and union ($\cup$) respectively.

**Exp6** $\text{exp}_1 \cup \text{exp}_2$ is an expression if $\text{exp}_1$ and $\text{exp}_2$ are relation expressions.

**Rule6** $\text{exp}_1 \cup \text{exp}_2 : \{1\}$, if $\text{exp}_1 : \{1\}$.

According to the definition for difference (as defined in set theory), the objects in $\text{exp}_1 \cup \text{exp}_2$ are those that belong to $\text{exp}_1$ and not to $\text{exp}_2$. That leads to Rule6.

We note that the difference could operate on relations of different types. So the difference is carried out not according to the values of the objects in $\text{exp}_1$ and $\text{exp}_2$, but according to the surrogates of them.

The following is the rule for expressions with respect to intersection.

**Exp7** $\text{exp}_1 \cap \text{exp}_2$ is an expression if $\text{exp}_1$ and $\text{exp}_2$ are relation expressions.

We expect that the objects in $\text{exp}_1 \cap \text{exp}_2$ inherit the methods of both types of $\text{exp}_1$ and $\text{exp}_2$. According to the definition, $\text{exp}_1 \cap \text{exp}_2$ could have different types. So the type that we want for $\text{exp}_1 \cap \text{exp}_2$ should be a subtype of both types of $\text{exp}_1$ and $\text{exp}_2$.

We define typing rule for such expressions in a very like way as that for the expressions concerning natural join. However, while natural join is carried out according to attribute values of objects, the intersection is carried out according to surrogates of objects.

This observation leads to define the following typing rule, that is also similar to what proposed by [9].

---

Rule4 $\pi A_{i_1}, \ldots, A_{i_m} (\text{exp}) : [t_{i_1}, \ldots, t_{i_m}]$ if $\text{exp} : [\ldots, A_{i_1} : t_{i_1}, \ldots, A_{i_m} : t_{i_m}, \ldots]$.

We have then the following typing rule:

**Rule3** $\text{exp}_1 \Join \text{exp}_2 : t_1 \cup t_2$, if $\text{exp}_1 : t_1$ and $\text{exp}_2 : t_2$.

$t_1 \cup t_2$ is the greatest lower bound (GLB) of $t_1$ and $t_2$. That is $t_1 \cup t_2 \leq t_1$ and $t_1 \cup t_2 \leq t_2$. Moreover, if there exists a type $t'$ such that $t' \leq t_1$ and $t' \leq t_2$ then $t' \leq t_1 \cup t_2$.

With noting that the order of the attributes in a tuple is unimportant, the following rule is defined for project on tuples.

**Exp4** $\pi A_{i_1}, \ldots, A_{i_m} (\text{exp})$ is an expression if $\text{exp}$ is a tuple expression.

Accordingly, the typing rule is defined as follows.
Rule 7. \(\exp_1 \cap \exp_2 : \{t_1 \vee t_2\} \) if \(\exp_1 : \{t_1\}\) and \(\exp_2 : \{t_2\}\).

With following rule we define expressions concerning union.

Exp 8 \(\exp_1 \cup \exp_2\) is an expression if \(\exp_1\) and \(\exp_2\) are relation expressions.

Again, \(\exp_1\) and \(\exp_2\) could have different types. That means the objects in \(\exp_1 \cup \exp_2\) could be of different types. However, according to the definition for set type, shown in section 3, a set is associated only to one type. In order to strongly type \(\exp_1 \cup \exp_2\), we expect that it is treated as though it is homogeneous. The consideration leads to the following typing rule.

Rule 8 \(\exp_1 \cup \exp_2 : \{t_1 \wedge t_2\}\) if \(\exp_1 : \{t_1\}\) and \(\exp_2 : \{t_2\}\).

\(t_1 \wedge t_2\) is the least upper bound (LUB) of \(t_1\) and \(t_2\). That is \(t_1 \leq t_1 \wedge t_2\) and \(t_2 \leq t_1 \wedge t_2\). Moreover, if there exists a type \(t'\) such that \(t_1 \leq t'\) and \(t_2 \leq t'\) then \(t_1 \wedge t_2 \leq t'\).

Similar to Rule 8 we add now a typing rule for the expressions defined in Exp 5.

Rule 5b \(\{\exp_1, \ldots, \exp_n\} : \{t_1 \wedge \ldots \wedge t_n\}\) if \(\exp_1 : \{t_1\}\), \(\ldots\), \(\exp_n : \{t_n\}\).

That is, a heterogeneous set will be treated as if it is homogeneous.

5 Conclusion

In this paper we described a type system that supports strong typing and static type checking for an object-oriented database system. Types defined in the type system are structured in an inheritance hierarchy, which is exploited as database schema by queries. We define the subtyping rule through using a partial ordering defined on type structures and on methods associated to types. The subtyping rule can be used for correctly structuring the type hierarchy and discovering the type definition error with respect to inheritance. Based on the definition of subtyping, a number of type inference rules are defined. With these rules the type system can infer types for expressions that appear in queries and find types for results of queries. It can therefore provide the closure property as the relational systems do and statically determine whether a given expression or query is type correct. Thus, when a application program is accepted by the type system (the type checker of the type system), the absence of type errors at run time is guaranteed. With respect to strong typing and static type checking, a comparison between ZnORM and several other object-oriented database systems was made in section 1.

References