Towards a formal type system for ODMG OQL

G.M. Bierman A. Trigoni
University of Cambridge Computer Laboratory*

September, 2000

Abstract

In this paper we consider in detail the type system of the object-oriented database query language, OQL, as defined by the ODMG [6]. Our main technical contribution is a formal definition of the typing relation for OQL—surprisingly we could not find a complete definition in the literature. We have also uncovered a number of inaccuracies in the original ODMG proposal, and other work.

*Authors' address: University of Cambridge Computer Laboratory, New Museums Site, Cambridge, CB2 3QG. UK. Email: {gmb, at263}@cl.cam.ac.uk

1 Introduction

"OQL is a functional language where operators can freely be composed, as long as the operands respect the type system."

ODMG [6, Page 89].

Object database management systems (ODBMS) provide an integration of (object-oriented) programming languages and database systems. By the early 1990s a number of proposals and systems were available, all with different underlying object models and system-specific features. Rather than continuing to work in different directions, the major companies and organisations in the ODBMS industry joined forces to form the Object Database Management Group (ODMG). The aim of the ODMG is to provide a de facto standard for ODBMS. To this aim they produced a book: the Object Data Standard (hereafter referred to simply as the Standard). The Standard consists of four major components: an object model; an object specification language (ODL); an object query language (OQL); and several programming language bindings (currently for Java, C++ and Smalltalk).

In this paper we will focus mainly on OQL. In particular we aim to make good the claim in the Standard of a well-defined type system for OQL. In contrast to the Standard's rather informal discussion, we present the type system *formally* using techniques familiar in modern programming language design [7, 12, 4]. Our paper presents a precise, mathematical description of the OQL type system which should be useful to both current implementors (we found it straightforward to implement an OQL type inference engine given our formal description), and for future extensions to the Standard.

Our paper is organised as follows. In Section 2 we identify a core OQL—a fragment of the language defined in the Standard, but which has the same expressive power. In Section 3 we define the notion of a type for OQL. In Section 4 we formalise the notion of subtyping inherent in the object model and show how schema definitions yield a subtyping relation. In Section 5 we give formal type rules for forming judgements about the type of OQL programs, definitions and queries. In Section 6 we compare our work to other related work. We conclude, in Section 7, with details of work in progress.

2 Core OQL

OQL is the object query language proposed by the ODMG to support their object data model. In this paper we address primarily the type system of OQL. Despite their aims of simplicity, the designers of OQL have, in many cases, provided several ways of writing the same query. This has been motivated by the desire of compatibility with SQL 92 and other query languages. For the purposes of this paper, however, we shall study a *core* OQL—a fragment of the language in the Standard, but which has the same expressive power. (It is easy to see how the full OQL can be translated into our core OQL.)

In the rest of this section we shall define the core OQL. First we shall assume a set of binary operators and a set of unary operators as follows.

$$\begin{aligned} \textit{binop} &= \{\texttt{and}, \texttt{or}, \texttt{intersect}, \texttt{union}, \texttt{except}, =, != , <, >, <=, >=, +, -, *, /, ||, \texttt{mod}\} \\ \textit{unop} &= \{\texttt{first}, \texttt{last}, \texttt{max}, \texttt{min}, \texttt{avg}, \texttt{sum}, \texttt{count}, \texttt{distinct}, \texttt{listtoset}, \\ &\quad \texttt{element}, \texttt{flatten}, \texttt{abs}, \texttt{not}, -\} \end{aligned}$$

The grammar for (untyped) OQL **queries** is then as follows.

```
q  ::=  b | f | i | c | s
     |  x
     |  bag(q, . . . , q) | set(q, . . . , q) | list(q, . . . , q) | array(q, . . . , q)
     |  dictionary((q, q), . . . , (q, q)) | struct(l: q, . . . , l: q)
     |  C(l: q, . . . , l: q) | q.l | (C)q
     |  q[q] | q in q | q() | q(q, . . . , q)
     |  forall x in q: q | exists x in q: q
     |  q binop q | unop(q)
     |  select [distinct] q
        from (q as x, · · · , q as x)
        where q
        [group by (l: q, · · · , l: q)]
        [having q]
        [order by (q asc|desc, · · · , q asc|desc)]
```

where b, f, i, c, s range over booleans, floats, integers, characters and strings respectively, `x` is taken from a countable set of identifiers, `l` is taken from a countable set of **labels**, and `C` ranges over a countable set of **class names**.

In OQL we are able to make named definitions. A **definition** is given by the following grammar.

```
d  ::=  define x as q
     |  define x(x: σ, . . . , x: σ) as q
```

An OQL **program** then consists of a number (maybe zero) of named definitions followed by a query.

3 Types

OQL, or more generally the underlying object model, has a quite complex notion of a type. Obviously our core OQL has a slightly simpler language of types—for example, we do not consider the types defined in the Standard [§2.3.7] for dates and time-zones. However, even with our core system we have a number of primitive types, n-ary function types, multiple (arbitrarily nested) collection types, structures and classes (the underlying object model is *class-based*).

More formally, the types for OQL are given by the following grammar.

$$
\begin{array}{rcl}
\sigma & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{string} \mid \texttt{void} \\
& \mid & \sigma \times \cdots \times \sigma \to \sigma \\
& \mid & \texttt{bag}(\sigma) \mid \texttt{set}(\sigma) \mid \texttt{list}(\sigma) \mid \texttt{array}(\sigma) \\
& \mid & \texttt{dictionary}(\sigma, \sigma) \mid \texttt{struct}(\texttt{l}\colon \sigma, \cdots, \texttt{l}\colon \sigma) \\
& \mid & \texttt{C}
\end{array}
$$

We use the '$\to$' symbol to denote a function type. For example, $\texttt{int} \to \texttt{float}$ is the type of a function which expects an argument value of type `int` and returns a value of type `float`. Function types arise when defining methods which take parameters.

The other types are self-explanatory. For example, the type `bag(bool)` is the type of a bag of booleans, and the type `set(list(string))` a set of lists of strings. The type `dictionary(string, set(int))` is the type of a dictionary with keys of type `string` and associated values of type `set(int)`. The type `struct(a: int, b: struct(c: float, d: char))`, is the type of a structure with two fields: the first has label `a` with associated integer values, the second field has label `b` with an associated structure value, which itself has two fields (`c` and `d`).

We find it useful to adopt the following conventions. We will write $\mathrm{Col}(\sigma)$, to denote an arbitrary collection type (array, bag, dictionary, list or set), with elements of type σ. The types `char`, `int`, `real` and `string` are said to be **orderable** types. We will use the symbol $\mathcal{O}$ to range over the subset of types that are orderable.

4 ODL Schema

4.1 Introduction

The Object Definition Language (ODL) is a language to define the specification of object types that conform to the ODMG object model. In more traditional database parlance, ODL is the *data definition language* for ODMG-compliant ODBMSs. We assume that the reader is familiar with ODL (see, e.g. [11]).

To determine the type of an OQL program we obviously need to use certain type information taken from the ODL schema. Primarily this includes the extents, attribute names and their types, the method names and their types, and the relationship names and their types. (The information concerning the type hierarchies is discussed in Section 4.2.)

We shall assume that from a given schema we have constructed a **schema typing environment**, which contains the important typing information from the class definitions.

More precisely, a schema typing environment, written $\mathcal{S}$, is a pair $\langle C, E\rangle$, where C is a partial function mapping class names to their type information, and E is a set of extent names along with their corresponding class identifiers. More formally

$$\begin{array}{rcl} \mathcal{S} = \langle C, E\rangle & & \\ C & : & \text{classId} \rightharpoonup \text{Att} \times \text{Rel} \times \text{Meth} \\ E & : & \wp(\text{Id} \times \text{Type}) \qquad \text{Att} : \wp(\text{Id} \times \text{Type}) \\ \text{Rel} & : & \wp(\text{Id} \times \text{Type}) \qquad \text{Meth} : \wp(\text{Id} \times \text{Type}) \end{array}$$

For example, consider the following schema.

```
class C extends D
( extent Cs)
{ attribute int a;
  float m (in int x)
};
```

This yields the following schema typing environment.

$$\begin{array}{rcl} \mathcal{S} & = & \langle C, E\rangle \\ C & = & \{\texttt{C} \mapsto \{\texttt{a}\colon \texttt{int}\} \times \emptyset \times \{\texttt{m}\colon \texttt{int} \to \texttt{float}\}\} \\ E & = & \{\texttt{Cs}\colon \texttt{C}\} \end{array}$$

In following sections we will find it useful to employ some shorthand. Given a schema typing environment, $\mathcal{S}$, and a class $\texttt{C}$, the collection of its attributes is denoted by $A(\mathcal{S}, \texttt{C})$, the collection of its relationships is denoted by $R(\mathcal{S}, \texttt{C})$, and the collection of its methods is denoted by $M(\mathcal{S}, \texttt{C})$. The union of these three sets is denoted by $I(\mathcal{S}, \texttt{C})$. Given a schema typing environment, $\mathcal{S}$, the collection of extents is denoted by $E(\mathcal{S})$.

4.2 Subtyping

The ODL class definitions also contain the details of the type hierarchies, that is, the sub-classing information. We can take a collection of class definitions and build a relation, $\sqsubseteq_C$, where $\texttt{C} \sqsubseteq_C \texttt{D}$ when class $\texttt{C}$ extends class $\texttt{D}$ (as in the example above). From this sub-class relation we build a general **subtype** relation between types.[1] The idea is that σ is a subtype of τ (conversely τ is a supertype of σ) if a value of type σ can be used in any context in which a value of type τ is expected. This is written $\sigma \leq \tau$.

The rules for building a subtype relation from a sub-class relation are as follows.

[1]In Java subtyping is often referred to as **widening**.

$$\frac{}{\mathtt{C} \leq \mathtt{Object}}\ \text{Top}$$

$$\frac{\mathtt{C} \sqsubseteq_{\mathrm{C}} \mathtt{C}'}{\mathtt{C} \leq \mathtt{C}'}\ \text{Sub-Class} \qquad \frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}\ \text{Sub-Fun}$$

$$\frac{\sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_{\mathtt{k}} \leq \tau_{\mathtt{k}}}{\sigma_1 \times \cdots \times \sigma_{\mathtt{k}} \leq \tau_1 \times \cdots \times \tau_{\mathtt{k}}}\ \text{Sub-Tuple} \qquad \frac{\sigma \leq \tau}{\mathrm{Col}(\sigma) \leq \mathrm{Col}(\tau)}\ \text{Sub-Coll}$$

$$\frac{\sigma_1 \leq \tau_1 \quad \cdots \quad \sigma_{\mathtt{k}} \leq \tau_{\mathtt{k}}}{\mathtt{struct}(\mathtt{l}_1{:}\,\sigma_1, \ldots, \mathtt{l}_{\mathtt{k}}{:}\,\sigma_{\mathtt{k}}, \ldots, \mathtt{l}_{\mathtt{k+n}}{:}\,\sigma_{\mathtt{k+n}}) \leq \mathtt{struct}(\mathtt{l}_1{:}\,\tau_1, \ldots, \mathtt{l}_{\mathtt{k}}{:}\,\tau_{\mathtt{k}})}\ \text{Sub-Struct}$$

$$\frac{}{\sigma \leq \sigma}\ \text{Sub-Refl} \quad \frac{\sigma \leq \sigma' \qquad \sigma' \leq \sigma''}{\sigma \leq \sigma''}\ \text{Sub-Trans}$$

These rules are relatively straightforward, but it is perhaps worth explaining in some detail the Sub-Fun rule. Another way of thinking about subtyping is to treat the relation $\sigma \leq \tau$ as an assertion of a well-behaved 'coercion' function from values of type σ to values of type τ [3]. Let us assume that $\tau \leq \tau'$, i.e. we have a coercion function from values of type τ to values of type τ'. We can see that $\sigma \to \tau \leq \sigma \to \tau'$, i.e. we can coerce a function of type $\sigma \to \tau$ to be one of type $\sigma \to \tau'$ by applying the function to a given value of type σ and then coercing the result (which will be of type τ) to a value of type τ'.

The coercion of the domain type is more tricky. If we assume that $\sigma' \leq \sigma$, then we can show that $\sigma \to \tau \leq \sigma' \to \tau'$, which is perhaps the reverse of what one might first expect. We can coerce a function of type $\sigma \to \tau$ to be one of type $\sigma' \to \tau$ by first coercing a given value of type σ' to be one of type σ, and then applying the function to produce a value of type τ.

In later sections we will use the notion of a least upper bound of two types, as described informally in the Standard [§4.10].[2] This notion can be defined formally as follows.

Definition 1 *Given types σ and σ', their* **least upper bound** *(lub) is a type τ such that*

1. *$\sigma \leq \tau$ and $\sigma' \leq \tau$, and*
2. *$\forall \tau'.\, \sigma \leq \tau'$ and $\sigma' \leq \tau'$ then $\tau \leq \tau'$.*

[2]It should be noted that given any pair of types, it is not necessarily the case that they have a *least* upper bound, but rather several distinct upper bounds. Casting may be required, as is the case for Java.

5 The type system of OQL

This section contains the main technical contribution of our paper. In it we develop the type system of OQL. In other words we give formal rules for determining the type of a given OQL program. As this formal approach may not be that widely known in the database community, we give a brief introduction in Appendix A. (A more complete introduction, albeit one biased to programming languages, can be found in [4].)

5.1 Typing judgements

Let us now consider forming typing judgements for OQL programs, definitions and queries. We shall give three type systems for these three syntactic categories (and we shall use different symbols for the turnstile for clarity).

First let us consider OQL programs. As discussed in §4, to type an OQL program, we need to use typing information extracted from the class definitions. In addition the Standard [§4.10.2] states that query definitions are persistent. Thus we need to know the type information from any previous query definitions. Also the Standard [§2.10] allows named objects to be inserted into the database from a language binding, and moreover to be referenced *directly* in OQL by their names. Hence we need the type information concerning these named objects.

Thus rather than have just the one typing environment, we find it convenient to keep the three typing environments (for the schema, definitions and named objects) separate. A **program typing judgement** for a given OQL program, p, is written

$$\mathcal{S};\mathcal{D};\mathcal{N} \blacktriangleright \mathtt{p}\colon\sigma;\mathcal{D}'.$$

where $\mathcal{S}$ is a schema typing environment, $\mathcal{D}$ is a definition typing environment and $\mathcal{N}$ is a named object typing environment. As the program p may extend the definition typing environment $\mathcal{D}$, the program typing judgement has also as a conclusion $\mathcal{D}'$, which is the resulting definition typing environment.

A **definition typing judgement** for an OQL definition, d is written

$$\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{d} \Rightarrow \mathcal{D}'$$

where $\mathcal{S}$, $\mathcal{D}$ and $\mathcal{N}$ are as before. A **query typing judgement** for an OQL query q, is written

$$\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathtt{q}\colon\sigma$$

where $\mathcal{S}$, $\mathcal{D}$ and $\mathcal{N}$ are as before. $\mathcal{Q}$ contains the types of any free identifiers in q, and is known as the **query typing environment**.

In the following subsections we give complete type systems for OQL programs, definitions and queries.

5.2 Typing OQL programs

The rules for forming program typing judgements are as follows.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{d} \Rightarrow \mathcal{D}' \qquad \mathcal{S};\mathcal{D}';\mathcal{N};\emptyset \vdash \mathtt{q}\colon \sigma}{\mathcal{S};\mathcal{D};\mathcal{N} \blacktriangleright \mathtt{d}\,\mathtt{q}\colon \sigma;\mathcal{D}'}\text{Prog-Def} \qquad \frac{\mathcal{S};\mathcal{D};\mathcal{N};\emptyset \vdash \mathtt{q}\colon \sigma}{\mathcal{S};\mathcal{D};\mathcal{N} \blacktriangleright \mathtt{q}\colon \sigma;\mathcal{D}}\text{Prog-Query}$$

Reading the first rule bottom-up, it states that to type the program $\mathtt{d}\,\mathtt{q}$, we first type the definition $\mathtt{d}$, to get a updated definition environment, $\mathcal{D}'$, which we use to type $\mathtt{q}$, the resulting type of which is the overall type of the program. (The second rule is just a nullary version where there are no definitions.)

The reader will also note that in typing the query $\mathtt{q}$ in both rules, we have insisted that the query environment is *empty*. In other words a top-level query may not have any free query identifiers.[3]

5.3 Typing OQL definitions

The three rules for forming definition typing judgements are as follows.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\emptyset \vdash \mathtt{q}\colon \tau}{\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{define}\ \mathtt{f}\ \mathtt{as}\ \mathtt{q} \Rightarrow \mathcal{D}, \mathtt{f}\colon \tau}\text{Def-Nullary}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathtt{x_1}\colon \sigma_1,\ldots,\mathtt{x_n}\colon \sigma_n \vdash \mathtt{q}\colon \tau}{\begin{array}{c}\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{define}\ \mathtt{f}(\mathtt{x_1}\colon \sigma_1,\ldots,\mathtt{x_n}\colon \sigma_n)\ \mathtt{as}\ \mathtt{q} \\ \Rightarrow \mathcal{D}, \mathtt{f}\colon \sigma_1 \times \cdots \times \sigma_n \to \tau\end{array}}\text{Def}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{d_1} \Rightarrow \mathcal{D}' \qquad \mathcal{S};\mathcal{D}';\mathcal{N} \rhd \mathtt{d_2} \Rightarrow \mathcal{D}''}{\mathcal{S};\mathcal{D};\mathcal{N} \rhd \mathtt{d_1};\mathtt{d_2} \Rightarrow \mathcal{D}''}\text{Def-Comp}$$

Consider the rule Def—reading the rule top-down it states that if a query $\mathtt{q}$ has type τ and free query identifiers $\mathtt{x_1},\ldots,\mathtt{x_n}$ of types $\sigma_1,\ldots,\sigma_n$ respectively, then the type of the definition $\mathtt{f}$ is the function type $\sigma_1 \times \cdots \times \sigma_n \to \tau$.

It is important to note that there are three side conditions to the rules Def and Def-Nullary. Firstly, we are not permitted to overload definition identifiers (the Standard [§4.10.2]).

[3]Rather unfortunately the Standard [§4.10.1] has the erroneous statement that a query is an "expression with no bound variables".

The second side condition is the definition identifier can not be an existing class name. (Algebraically $\forall\sigma.(f\colon\sigma) \notin \mathrm{dom}(\mathcal{S})$.) The third side condition is that the definition identifier can not coincide with an existing named object. (Algebraically $\forall\sigma.(f\colon\sigma) \notin \mathcal{N}$.)

The rule Def-comp handles the case of multiple definitions, which are typed in the order they are given.

5.4 Typing OQL queries

We now consider the rules for deriving query typing judgements. First are the axioms for literals.

$$\frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash b\colon\texttt{bool}} \quad \frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash i\colon\texttt{int}} \quad \frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash f\colon\texttt{float}}$$

$$\frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash c\colon\texttt{char}} \quad \frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash s\colon\texttt{string}}$$

Next are the axioms for identifiers. We have several axioms depending on what sort of identifier we are dealing with: it is either an extent defined in a schema, a definition, a object name or a query identifier.

$$\frac{\texttt{e}\colon\texttt{C}\in E(\mathcal{S})}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \texttt{e}\colon\texttt{set(C)}}\ \text{Extent-Id} \qquad \frac{}{\mathcal{S};\mathcal{D},\texttt{d}\colon\sigma;\mathcal{N};\mathcal{Q}\vdash \texttt{d}\colon\sigma}\ \text{Def-Id}$$

$$\frac{}{\mathcal{S};\mathcal{D};\mathcal{N},\texttt{n}\colon\sigma;\mathcal{Q}\vdash \texttt{n}\colon\sigma}\ \text{Named-Object-Id} \qquad \frac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\texttt{x}\colon\sigma\vdash \texttt{x}\colon\sigma}\ \text{Query-Id}$$

It is worth noting here our use of a convenient shorthand. Rather than writing an extended typing environment, for example $\mathcal{Q}\cup\{\texttt{x}\colon\sigma\}$, we write instead $\mathcal{Q},\texttt{x}\colon\sigma$. In other words the comma can be interpreted as set union.

Next we shall give the type rules for subtyping, which are as follows.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \texttt{q}\colon\texttt{C} \qquad \texttt{C}\le\texttt{D}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \texttt{(D)q}\colon\texttt{D}}\ \text{Casting} \qquad \frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \texttt{q}\colon\sigma \qquad \sigma\le\tau}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \texttt{q}\colon\tau}\ \text{Subtyping}$$

The Casting rule allows an expression of object type `C` to be explicitly cast to a supertype `D`. In the Standard [§4.10.12.5] it is also allowed that an object type be downcast to a *subtype*. This would entail changing our rule to:

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon \mathtt{C} \qquad \mathtt{C}\leq \mathtt{D} \text{ or } \mathtt{D}\leq \mathtt{C}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash (\mathtt{D})\mathtt{q}\colon \mathtt{D}}\text{Casting}$$

It is important to note that this rule has the consequence of requiring *run-time* type checking.

The Subtyping rule embodies our understanding of subtyping, i.e. if $\sigma \leq \tau$ then a value of type σ can be used in any context in which a value of type τ is expected. Consequently if we deduce that an expression is of type σ, we can also deduce that it is of type τ (if $\sigma \leq \tau$).

The rules for structures are straightforward and are as follows.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \sigma_1 \quad \cdots \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_k}\colon \sigma_k}{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{struct}(\mathtt{l_1}\colon \mathtt{q_1},\ldots,\mathtt{l_k}\colon \mathtt{q_k})\\ \qquad\qquad\qquad\colon \mathtt{struct}(\mathtt{l_1}\colon \sigma_1,\ldots,\mathtt{l_k}\colon \sigma_\mathtt{k})\end{array}}\text{Struct}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon \mathtt{struct}(\mathtt{l_1}\colon \sigma_1,\ldots,\mathtt{l_k}\colon \sigma_\mathtt{k})}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q.l_i}\colon \sigma_i}\text{Struct-Select}$$

There are several rules to deal with various collections, which are given in Figure 1. The important rule here is Collection. Reading the rule bottom-up, it reads that to assert that the collection Col($\mathtt{q_1},\ldots,\mathtt{q_k}$) has type Col($\sigma$), we must assert that each $\mathtt{q_i}$ has the type σ.

The Subtyping rule given earlier allows for such a simple typing rule (c.f. [§§4.10.4.4–4.10.5.6] of the Standard). For example, imagine that expressions $\mathtt{q_1}$ and $\mathtt{q_2}$ are of type σ_1 and σ_2 respectively, where σ_1 and σ_2 have a lub τ. Thus we can form the following typing derivation:

$$\frac{\dfrac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \sigma_1}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \tau}\text{Sub.} \qquad \dfrac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon \sigma_2}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon \tau}\text{Sub.}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{set}(\mathtt{q_1},\mathtt{q_2})\colon \mathtt{set}(\tau)}\text{Colln.}$$

The rules for queries dealing with quantification over collections are as follows.

$$\frac{\begin{array}{cc} \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\sigma & \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_k}\colon\sigma \\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q'_1}\colon\tau \quad \cdots & \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q'_k}\colon\tau \end{array}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{dictionary}((\mathtt{q_1},\mathtt{q'_1}),\ldots,(\mathtt{q_k},\mathtt{q'_k}))\colon\mathtt{dictionary}(\sigma,\tau)}\text{ Dictionary}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\sigma \quad \cdots \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_k}\colon\sigma}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathrm{Col}(\mathtt{q_1},\ldots,\mathtt{q_k})\colon\mathrm{Col}(\sigma)}\text{ Collection}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon\mathtt{list}(\sigma)}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{first}(\mathtt{q})\colon\sigma}\text{ First-list} \qquad \frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon\mathtt{array}(\sigma)}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{first}(\mathtt{q})\colon\sigma}\text{ First-array}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon\mathtt{list}(\sigma)}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{last}(\mathtt{q})\colon\sigma}\text{ Last-list} \qquad \frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon\mathtt{array}(\sigma)}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{last}(\mathtt{q})\colon\sigma}\text{ Last-array}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\mathtt{list}(\sigma) \qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon\mathtt{int}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}[\mathtt{q_2}]\colon\sigma}\text{ Index-list}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\mathtt{array}(\sigma) \qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon\mathtt{int}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}[\mathtt{q_2}]\colon\sigma}\text{ Index-array}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\mathtt{dictionary}(\sigma,\tau) \qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon\sigma}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}[\mathtt{q_2}]\colon\tau}\text{ Index-Dict}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon\sigma \qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_2}\colon\mathrm{Col}(\sigma)}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\ \mathtt{in}\ \mathtt{q_2}\colon\mathtt{bool}}\text{ Membership}$$

Figure 1: Typing rules for collections

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \mathrm{Col}(\sigma) \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x}\colon\sigma \vdash \mathtt{q_2}\colon \mathtt{bool}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{forall\ x\ in\ q_1}\colon \mathtt{q_2}\colon \mathtt{bool}}\ \text{Forall}$$

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \mathrm{Col}(\sigma) \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x}\colon\sigma \vdash \mathtt{q_2}\colon \mathtt{bool}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{exists\ x\ in\ q_1}\colon \mathtt{q_2}\colon \mathtt{bool}}\ \text{Exists}$$

Consider the Forall rule, reading from the top-down. First we have the judgement that the expression $\mathtt{q_1}$ is of type $\mathrm{Col}(\sigma)$, with free query identifiers contained in the set $\mathcal{Q}$. We then have the judgement that the expression $\mathtt{q_2}$ is of type `bool`, with free query identifiers contained in the extended set $\mathcal{Q},\mathtt{x}\colon\sigma$. We can then infer that the expression $\mathtt{forall\ x\ in\ q_1}\colon\mathtt{q_2}$ is of type `bool`, with free query identifiers contained in $\mathcal{Q}$. It is important to note that `x` may be a free identifier in the expression $\mathtt{q_2}$, but it is *bound* in the expression $\mathtt{forall\ x\ in\ q_1}\colon\mathtt{q_2}$.

The rules for object creation and method invocation are as follows.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_1}\colon \sigma_1 \quad \cdots \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q_k}\colon \sigma_k}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{C}(\mathtt{l_1}\colon\mathtt{q_1},\ldots,\mathtt{l_k}\colon\sigma_{\mathtt{k}})\colon\mathtt{C}}\ \text{Object} \qquad \frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}\colon\mathtt{C}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q.l}\colon\sigma}\ \text{Path}$$

The rule Object has an important side condition: to construct an object directly in OQL, the values for the attributes and relationships must be of the expected types, as declared in the schema. Put algebraically:

$$\forall 1\leq i\leq k.\ (\mathtt{l}_i\colon\sigma_i)\in A(\mathcal{S},\mathtt{C})\ \text{or}\ (\mathtt{l}_i\colon\sigma_i)\in R(\mathcal{S},\mathtt{C})$$

(Note that the Standard [§4.4.1] allows an object to be created with certain fields undefined—these undefined attributes and relationships "are given a default value".)

The rule Path also contains a side condition that the label `l` produces a valid path expression for an object of type `C`. This can be expressed algebraically: $(\mathtt{l}\colon\sigma)\in I(\mathcal{S},\mathtt{C})$. The case where the label `l` represents a method which takes a number of parameters is handled by a combination of the Path rule and the App rule, which is explained later in this section.

Now we turn our attention to the most important construct in OQL, the select query. Rather than give a single rule handling all the forms at once, we tackle each form separately. The rules for the various forms of select queries are given in Figure 2.

Consider the rule Vanilla-Select, reading from the top-down. First we have the judgement that expression $\mathtt{q_1}$ is of type $\mathrm{Col}(\sigma_1)$, with free query identifiers contained in $\mathcal{Q}$. Thus query identifier $\mathtt{x_1}$ is of type σ_1 (which respects the semantic explanation given in the Standard [§4.10.9]) and may occur free in expression $\mathtt{q_2}$, along with any identifiers contained

$$\frac{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathsf{q_1}\colon \mathrm{Col}(\sigma_1)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1\vdash \mathsf{q_2}\colon \mathrm{Col}(\sigma_2)\\ \ldots\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1,\ldots,\mathtt{x_{k-1}}\colon\sigma_{\mathtt{k-1}}\vdash \mathsf{q_k}\colon \mathrm{Col}(\sigma_k)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}''\colon\mathtt{bool}\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}'\colon\tau\end{array}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{select}\;\mathsf{q}'\;\mathtt{from}\;(\mathsf{q_1}\;\mathtt{as}\;\mathtt{x_1},\ldots,\mathsf{q_k}\;\mathtt{as}\;\mathtt{x_k})\;\mathtt{where}\;\mathsf{q}''\colon\mathtt{bag}(\tau)}\;\text{Vanilla-Select}$$

$$\frac{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathsf{q_1}\colon \mathrm{Col}(\sigma_1)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1\vdash \mathsf{q_2}\colon \mathrm{Col}(\sigma_2)\\ \ldots\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1,\ldots,\mathtt{x_{k-1}}\colon\sigma_{\mathtt{k-1}}\vdash \mathsf{q_k}\colon \mathrm{Col}(\sigma_k)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}''\colon\mathtt{bool}\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}_1'''\colon\mathcal{O}_1\\ \ldots\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}_{\mathsf{j}}'''\colon\mathcal{O}_j\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}'\colon\tau\end{array}}{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{select}\;\mathsf{q}'\;\mathtt{from}\;(\mathsf{q_1}\;\mathtt{as}\;\mathtt{x_1},\ldots,\mathsf{q_k}\;\mathtt{as}\;\mathtt{x_k})\;\mathtt{where}\;\mathsf{q}''\\ \qquad\qquad\mathtt{order\,by}\;(\mathsf{q}_1'''\;\mathtt{asc}|\mathtt{desc},\ldots,\mathsf{q}_{\mathsf{j}}'''\;\mathtt{asc}|\mathtt{desc})\colon\mathtt{list}(\tau)\end{array}}\;\text{Order-Select}$$

$$\frac{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathsf{q_1}\colon \mathrm{Col}(\sigma_1)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1\vdash \mathsf{q_2}\colon \mathrm{Col}(\sigma_2)\\ \ldots\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\colon\sigma_1,\ldots,\mathtt{x_{k-1}}\colon\sigma_{\mathtt{k-1}}\vdash \mathsf{q_k}\colon \mathrm{Col}(\sigma_k)\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}''\colon\mathtt{bool}\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma}\vdash \mathsf{q}_1'''\colon\tau_1\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma},\mathtt{l_1}\colon\tau_1\vdash \mathsf{q}_2'''\colon\tau_2\\ \ldots\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma},\mathtt{l_1}\colon\tau_1,\ldots,\mathtt{l_{j-1}}\colon\tau_{\mathtt{j-1}}\vdash \mathsf{q}_{\mathsf{j}}'''\colon\tau_j\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma},\vec{\mathtt{l}}\colon\vec{\tau},\mathtt{partition}\colon\mathtt{bag}(\mathtt{struct}(\vec{\mathtt{x}}\colon\vec{\sigma}))\vdash \mathsf{q}''''\colon\mathtt{bool}\\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\vec{\mathtt{x}}\colon\vec{\sigma},\vec{\mathtt{l}}\colon\vec{\tau},\mathtt{partition}\colon\mathtt{bag}(\mathtt{struct}(\vec{\mathtt{x}}\colon\vec{\sigma}))\vdash \mathsf{q}'\colon\varphi\end{array}}{\begin{array}{ll}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash & \mathtt{select}\;\mathsf{q}'\;\mathtt{from}\;(\mathsf{q_1}\;\mathtt{as}\;\mathtt{x_1},\ldots,\mathsf{q_k}\;\mathtt{as}\;\mathtt{x_k})\;\mathtt{where}\;\mathsf{q}''\\ & \mathtt{group\,by}\;(\mathtt{l_1}\colon\mathsf{q}_1''',\ldots,\mathtt{l_j}\colon\mathsf{q}_{\mathsf{j}}''')\\ & \mathtt{having}\;\mathsf{q}''''\colon\mathtt{bag}(\varphi)\end{array}}\;\text{Group-Having-Select}$$

Figure 2: Typing rules for select queries

in $\mathcal{Q}$. This continues until we have the judgement that expression $\mathtt{q_k}$ is of type $\mathrm{Col}(\sigma_k)$, with free identifiers from $\{\mathtt{x_1}, \ldots, \mathtt{x_{k-1}}\}$ and $\mathcal{Q}$. We then form the judgement that $\mathtt{q}''$ is of type `bool`, with free query identifiers from $\{\mathtt{x_1}, \ldots, \mathtt{x_k}\}$ (which we abbreviate to $\vec{\mathtt{x}}$) and $\mathcal{Q}$. Similarly we form the judgement that the query $\mathtt{q}'$ has type τ, with free query identifiers from $\vec{\mathtt{x}}$ and $\mathcal{Q}$. Given these (k+2) judgements we can conclude that the select query is of type $\mathtt{bag}(\tau)$.

It is important to notice that the free query identifiers of the select query are contained in $\mathcal{Q}$. In other words, the identifiers $\vec{\mathtt{x}}$ are *bound* by the query. This is an important advantage of our use of formal typing rules—the distinctions of bound and free identifiers is explicit in the rules themselves. (The precision and clarity of these rules should be compared to the rather verbose description given in the Standard [§4.10.9, §4.10.15].)

The rule where the select query has a `distinct` clause is omitted for space reasons, but is identical to the Vanilla-Select rule, except that the overall type of the query is a set and not a bag (unless it it has an `order` clause, in which case it remains a list).

Discussion. It is important to point out here that there is a serious error in the informal discussion of the typing of a select query in the Standard [page 111]. There they state that the $\mathtt{q_i}$ are "of type `Collection`", i.e. the *class* type, and not the literal type $\mathrm{Col}(\sigma_i)$ as we have given. We view this as a *mistake*—it certainly contradicts §4.10.4.3 of the Standard, where it is stated that given the declaration "`e as x`", then "`e` is of type `collection(t)` ", as well as their discussions of examples (e.g. [page 90]).

Unfortunately Alagić [1] has taken this erroneous statement at face value, and so proposes the following (simplified) type rule (where `any` is a notional polymorphic class type, no longer part of the ODMG standard).

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathtt{q_1}\!:\mathtt{Collection} \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\!:\mathtt{any} \vdash \mathtt{q}''\!:\mathtt{bool} \quad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q},\mathtt{x_1}\!:\mathtt{any} \vdash \mathtt{q}'\!:\tau}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathtt{select}\,\mathtt{q}'\,\mathtt{from}\,\mathtt{q_1}\,\mathtt{as}\,\mathtt{x_1}\,\mathtt{where}\,\mathtt{q}''\!:\mathtt{Bag}}$$

Given this type rule, he concludes that "OQL queries cannot be type-checked in the ODMG object model" [1, Theorem 2]. Given that the literal types we use in our rules are certainly valid in the ODMG object model (see the Standard [§2.4.1.2]), we refute Alagić's theorem, and in fact assert the opposite.

Fact 1 *OQL queries* ***can*** *be type-checked in the ODMG object model.*

□

Discussion. The reader may have noticed that we have not provided a `select *` form. Primarily this is because we view it as essentially syntactic sugar. More seriously, however, the way it is defined in the Standard [§4.10.9] breaks the convention that the query identifiers in the '`as`' clause are bound by the select query. The Standard [page 112] states that the query

```
select *
from   (Students as x, x.takes  as y, y.taught_by as z)
where  z.rank="Full Professor"
```

should be assigned the type $\texttt{bag}(\texttt{struct}(\texttt{x}\colon \texttt{Student}, \texttt{y}\colon \texttt{Section}, \texttt{z}\colon \texttt{Professor}))$. In other words, the bound query identifiers become labels for the structure, i.e. they change their syntactic category. We take the opinion that this confusion was an unfortunate design error. In core OQL we insist that this query be written as follows.

```
select struct(x: x', y: y', z: z')
from   (Students as x', x'.takes as y', y'.taught_by as z')
where  z'.rank="Full Professor"
```

□

Now we return to the rules for select queries, and consider that concerning an extension with an `order by` clause. This rule is similar to the vanilla select rule. The important thing to note is that the expressions which order the results, the q_i''', must be of an *orderable* type (as defined in Section 3). The Standard [§4.10.9.1, item 3] specifies that the type of an ordered select query is always a list.

Discussion. The Standard does not specify what happens when the sort criterion in the `order` clause does not distinguish completely between results, e.g. in the query

```
select x from Employees as x
       order by x.Salary
```

what if there are employees with the same salary? Presumably their order in the resulting list is system-dependent. To alleviate this problem we suggest that the resultant type in the Order-Select rule be changed to $\texttt{list}(\texttt{bag}(\tau))$ (the duplicates are now stored in bags).

□

There are two remaining type rules which deal with the application of an expression of a function type to appropriately typed arguments.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}\colon \texttt{void} \to \tau}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}()\colon \tau}\ \text{App-Nullary} \qquad \frac{\begin{array}{l}\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}\colon \sigma_1 \times \cdots \times \sigma_n \to \tau \\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q_1}\colon \sigma_1 \\ \cdots \\ \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q_n}\colon \sigma_n\end{array}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}(\mathsf{q_1},\ldots,\mathsf{q_n})\colon \tau}\ \text{App}$$

Consider the App rule, reading from the top-down. First we have the judgement that the expression `q` is of a function type $\sigma_1 \times \cdots \times \sigma_n \to \tau$, i.e. it is a function of n parameters. We then have a judgement for each of the n parameters that they are of the expected type. Given these we can conclude that applying these n arguments to the expression `q`, gives an expression whose type is τ.

We often use this rule in conjunction with the Path rule given earlier. Suppose we have a class `C` which has a method `m`, which expects an input parameter of type `float` and returns a value of type `int`. Then assuming that an expression $\mathtt{q}_1$ is of type `C`, and expression $\mathtt{q}_2$ is of type `float`, we can form the following typing derivation.

$$\dfrac{\dfrac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}_1\colon \mathtt{C}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}_1.\mathtt{m}\colon \mathtt{float}\to\mathtt{int}}\text{Path}\qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}_2\colon\mathtt{float}}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{q}_1.\mathtt{m}(\mathtt{q}_2)\colon\mathtt{int}}\text{App.}$$

To complete our discussion of the OQL type system we need to consider the types of the unary and binary operators. Most of these operators are **overloaded**, in the sense that they are intended to have a number of distinct types. Thus for each operator there are a number of type axioms. For reasons of brevity we shall just give the four axioms for the `union` operator, the others are self-apparent.

$$\dfrac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{union}\colon \mathtt{set}(\sigma)\times\mathtt{set}(\sigma)\to\mathtt{set}(\sigma)}$$

$$\dfrac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{union}\colon \mathtt{bag}(\sigma)\times\mathtt{set}(\sigma)\to\mathtt{bag}(\sigma)}$$

$$\dfrac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{union}\colon \mathtt{bag}(\sigma)\times\mathtt{set}(\sigma)\to\mathtt{bag}(\sigma)}$$

$$\dfrac{}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q}\vdash \mathtt{union}\colon \mathtt{bag}(\sigma)\times\mathtt{bag}(\sigma)\to\mathtt{bag}(\sigma)}$$

6 Related work

After completing the first draft of our paper, Alagić's paper [1] was published. He also considers a formalisation of the OQL type system, and claims that "several negative results are proved about the ability to type-check queries". We have seen earlier that his problems concerning the OQL type system can be overcome by using a different typing rule (one which is consistent with the Standard).

However, Alagić also demonstrates a much more serious problem with the Java language binding, although again we disagree with some of what he says. He gives a type rule [1, Rule 26], which appears to be a rule for typing Java directly, i.e. he considers OQL as a language extension of Java. Clearly there is a problem here as OQL has parameterised types (e.g.`bag(int)`), but Java does not (all it has is covariant arrays).

Using the host language type system to type queries has been a dream of OODBMS designers (see, for example, [13, §2.3].) However, *nowhere* in the Standard does the ODMG propose that the Java type system be used to type OQL. Instead the Standard gives a number of language bindings, for example in [§7.4.2] it gives the following Java interface.

```
public interface OQLQuery{
 public void   create(String query)
        throws QueryInvalidException;
 public void   bind(Object parameter)
        throws QueryParameterCountInvalidException,
               QueryParameterTypeInvalidException;
 public Object execute()
        throws QueryException;}
```

So we would use these methods in our Java program to create and execute OQL queries, for example:

```
OQLQuery example;
example.create("select x.age
                from Persons as x
                where x.name="Pat");
answers=(DBag)example.execute();
```

Thus we would expect the implementation of the `create` method, to actually implement the type system described in this paper. If the query can not be well typed, then the `QueryInvalidException` exception is raised. Alagić's claim that this can not be done in Java is quite wrong.

However, Alagić is correct in pointing out the mismatch between the expressive power of the OQL type system and that for Java—others [8] have criticised Java for this omission. As a consequence, invocations of the `bind` method, which enable Java objects to be passed into OQL queries, clearly require the Java objects to be re-typed under the ODMG type system (again we assume this is why the ODMG specified a `QueryParameterTypeInvalidException` method). How this is achieved, however, is clearly system-specific.

Riedel and Scholl [9] also set out to describe formally the type system underlying the ODMG object model. There are certainly strong similarities between their presentation and ours, but also some differences. Firstly, they study a much earlier version of the Standard (version 1.2), so some of their work is no longer relevant. Secondly, they treat class types quite differently to us (indeed, quite differently from the way they are treated by Java). Finally, (despite their title) they do not give many formal details of the type rules, but rather describe them using examples.

7 Conclusions and future work

In this paper we have studied closely the query language, OQL, proposed by the ODMG for object databases. We have identified a core OQL which is of the same expressive power as the full language, and for this core language we have given a complete set of type rules. These specify precisely the valid judgements one can make concerning the type of a given OQL program. In the process of defining the type rules we have shown what type information needs to be extracted from the ODL schema and how the subtyping relation is generated.

The principal feature of our work is the application of techniques familiar from modern programming language design (see, for example, the work on SML [7], and Java [10, 12]) to database query language design. Of course, this has entailed the use of a modicum of mathematical formalism (although no more than one can reasonably expect a Computer Science graduate to understand). We can give a number of reasons (there are many more!) why we think this is worthwhile.

- **Precision.** The use of mathematical formalism forces one to be completely precise about the type system (certainly more precise than the Standard!).
- **Conciseness.** Typing rules are a very concise method for defining type systems. Certainly the type rules for our core OQL take only a couple of pages. The type rules for Standard ML, a high-level, general-purpose programming language with higher-order functions, polymorphism, exceptions, user-defined recursive datatypes and a powerful module system, takes under two dozen pages to define [7].
- **Correctness.** Given a mathematical description of a type system, one is then able to prove (formally) facts about the system. Clearly it is very difficult to prove any facts about a type system that has only been informally defined, but worse, it is easy to come to false conclusions. For example, the informal description of the type system for Eiffel, an object-oriented programming language, was thought to be correct before a formal study showed it to be faulty [5].
- **Flexibility.** Our formalisation provides a flexible framework to study, for example, possible extensions to the underlying object model (e.g. the possibility of allowing parametric polymorphism for class types [8, 2]).

There are several areas of current work in progress, following on from that described in this paper. One topic we are working on is the problem of type inference. This is the process of discovering, automatically, the type of a given OQL program. The reader will have noticed that nearly all our type rules are **syntax-directed**, in that for each program construct there is only one rule which could be applied to produce a type derivation. The exception is the Subtyping rule. This rule complicates the process of type inference (indeed, without it, type inference would be trivial!).

The solution is to provide an alternative set of rules which are syntax-directed, and thus have the action of the Subtyping rule built into them. For example, here is a (simplified) version of an application rule.

$$\frac{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}_1\colon \sigma \to \tau \qquad \mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}_2\colon \sigma' \quad \sigma' \leq \sigma}{\mathcal{S};\mathcal{D};\mathcal{N};\mathcal{Q} \vdash \mathsf{q}_1(\mathsf{q}_2)\colon \tau}$$

In a forthcoming paper this syntax-directed typing system is proven correct with respect to the system given in this paper.

We have implemented the type system described in this paper in Java. Given our formal description of the type system, we found this a relatively straightforward process (another advantage of our approach!). We used the Poet Object Server (version 6.0) to store the objects and schema that are referenced in the OQL programs. We are currently extending our implementation to cover the full OQL language.

Acknowledgements

We should like to thank Ken Moody for his guidance. Trigoni is funded by the National Scholarships Institute of Greece and the National Bank of Greece.

References

[1] S. Alagić. Type-checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems*, 24(3):319–360, 1999.

[2] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*. ACM, 1998.

[3] V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance and implicit coercion. *Information and Control*, 93(1):172–221, 1991.

[4] L. Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997.

[5] W.R. Cook. A proposal for making Eiffel type-safe. In *Proceedings of the European conference on object-oriented programming*, pages 57–72. Cambridge University Press, 1989.

[6] R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[8] A.C. Myers, J.A. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of Symposium on Principles of Programming Languages*, pages 132–145, 1997.

[9] H. Riedel and M.H. Scholl. A formalization of ODMG queries. In Proceedings of the 7th IFIP 2.6 Working Conference on Database Semantics, 1997.

[10] D. Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, 1997.

[11] J.D. Ullman and J. Widom. *A First Course in Database Systems.* Prentice-Hall International, 1997.

[12] D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. 1999.

[13] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In *Readings in Object-oriented Database Systems*, pages 1–32. Morgan Kaufmann, 1990.

A A brief introduction to type systems

Type systems are specified using a particular formalism. The basic building block is a **typing judgement**. A typical typing judgement is of the form

$$\Gamma \vdash \texttt{e}\colon \sigma$$

which is read as an assertion that from the assumptions contained in the set Γ, the expression e has type σ (the symbol '$\vdash$' is often referred to as a turnstile). Γ contains the types of any free identifiers in the expression e, and is often called a **typing environment**. Elements of a typing environment are written, for example y: bool, which states that the identifier y is of type bool. For example, two typing judgements might be

$$\emptyset \vdash \texttt{true}\colon \texttt{bool}, \text{ and}$$

$$\texttt{x}\colon \texttt{float} \vdash \texttt{set}(\texttt{x}, 3.14)\colon \texttt{set}(\texttt{float}).$$

The former judgement states that from no assumptions we can conclude that the expression true is of type bool; the latter that from the assumption that the identifier x is of type float, then the expression set(x, 3.14) is of type set(float).

Any typing judgement is either valid (such as the two above) or invalid (such as $\emptyset \vdash$ 'c' $+ 3.6$: int). We characterise the set of valid typing judgements by giving a number of axioms and rules for forming these judgements. These axioms are essentially judgements which are intrinsically valid. One example might be:

$$\frac{}{\emptyset \vdash 4\colon \texttt{int}}$$

A **type rule** allows us to build valid typing judgements on the basis of other judgements which are known to be valid. We write these rules in the form

$$\frac{P_1 \ldots P_k}{C}$$

where the P_i are the premise judgements, and C the (single) conclusion judgement. When all of the premises are satisfied, then the conclusion must hold. An example of a possible type rule is:

$$\frac{\Gamma \vdash \texttt{e}\colon \texttt{int} \qquad \Gamma \vdash \texttt{f}\colon \texttt{int}}{\Gamma \vdash \texttt{e} + \texttt{f}\colon \texttt{int}}$$

This states that if the expressions e and f can be shown to be of type int, then the expression e + f is of type int.

A collection of axioms and type rules is called a **type system**. A **typing derivation** is a tree of judgements, where the leaves are axioms and where each typing judgement is obtained from the ones above it using a particular type rule. Given a typing environment Γ, an expression e is said to be **well-typed**, if there exists a type σ such that we can construct a typing derivation with the root $\Gamma \vdash \texttt{e}\colon \sigma$. The process of discovering a derivation (and hence the type) for a given expression is known as **type inference** and is discussed in Section 7.