Inferring the Principal Type and the Schema Requirements of an OQL Query

A. Trigoni and G.M. Bierman

University of Cambridge Computer Laboratory, UK

Abstract. In this paper, we present an inference algorithm for OQL which both identifies the most general type of a query in the absence of schema type information, and derives the minimum type requirements a schema should satisfy to be compatible with this query. Our algorithm is useful in any database application where heterogeneity is encountered, for example, schema evolution, queries addressed against multiple schemata, inter-operation or reconciliation of heterogeneous schemata. Our inference algorithm is technically interesting as it concerns an object functional language with a rich semantics and complex type system. More precisely, we have devised a set of constraints and an algorithm to resolve them. Our resulting type inference system for OQL should be useful in any open distributed, or even semi-structured, database environment.

1 Introduction

The ODMG Standard [6] (hereafter referred to as simply the Standard) presents, rather informally, some details of a type system for checking OQL queries using type information about the classes, extents, named objects and query definitions from a given database schema. Recently there have been some efforts to formalise this type system [2, 3]. This paper builds on our earlier work [3] and considers the problem of inferring the most general type of an OQL query in the absence of any schema information.

For example, consider the following OQL definition and query:

```
define Dept_Managers(dept) as
  select e
  from   Employees as e
  where  e.position="manager" and e.department=dept;

select d
from   Departments as d
where  count(Dept_Managers(d))>5
```

This query yields those departments that have more than five managers. It is interesting to notice that this information could be drawn by running the query against databases with significantly different schemata. For instance, consider schema A, which has two classes, `Employee` and `Department`, defined as follows.

```
class Employee (extent Employees)       class Department (extent Departments)
{ attribute string     name;            { attribute string id;}
  attribute string     position;
  attribute int        year_of_birth;
  attribute float      salary;
  attribute Department department;}
```

On the other hand, consider a second schema B, which has a class `Employee` and a named collection object `Departments` of type `List(int)`.

```
class Employee (extent Employees)
{ attribute string name;
  attribute string position;
  attribute int    department;}
```

The query could potentially run against both A and B without causing any type errors. In the case of schema A, the result of the query would be a bag of `Department` objects. In a database with schema B, the result of the query would be a bag of integers. Two vital questions arise at this point. First, how we can draw limits, or put restrictions, on the properties of a schema, so that a certain query is well-typed with respect to it? Second, what information we can derive about the type of the result of the query, supposing that we have no specific schema in mind? In this paper, we study these two questions in detail, but first let us consider the setting where this could be important.

For example, this information could be exploited in distributed database applications. Suppose we have time critical queries addressed against multiple schemata. If frequent updates on parts of these schemata are likely to occur, then many of the queries will inevitably fail to be executed. In order to avoid this situation, we should register interest in specific updates of each schema –at least in those that would affect the critical queries– and resolve the type incompatibility in due course and not at the time the queries get executed.

Our work is equally useful in contexts where we need to achieve interoperation between heterogeneous sources. There has been a lot of research on reconciling schemata with semantic heterogeneity [4, 7]. One approach to this problem identifies the semantic inconsistencies of the ontologies in different domains and creates a global ontology that combines all of them. Another approach identifies the intersection of domains where the inconsistencies occur and tries to resolve them by introducing matching rules between them. In both cases, queries that are initially written to be executed on one domain need to be rephrased to fit the needs of more domains. Knowing the schema requirements of a query and the schema mappings to a (global or just different) ontology, the task of rephrasing queries becomes a trivial automatic process. Suppose that a group of airline companies cooperate to create a single uniform system for booking tickets. In order to do that they define a global ontology that is very close to each of the distinct ontologies. Each query is initially phrased to conform to the global ontology and is then transformed to appropriate queries addressed to the individual schemata. The transformation is much easier to perform if besides the schema mappings (from one ontology to the other), we are aware of the query schema requirements. The latter effectively point out the exact mappings we need to use.

This paper is organised as follows. In section 2 we recall our earlier [3] definition of a core OQL—a fragment of the language defined in the Standard, but which has the same expressive power. We give a brief overview of the type system of OQL, including the notion of subtyping. In section 3, we study the type system of our inference model introducing a new relation between types, called *more specific*. In section 4, we describe the kinds of constraints generated by our type inference algorithm, and in section 5, we present an algorithm for resolving these constraints. The core of our inference system, the inference rules, are given in section 6. Finally, in section 7, we present the inference algorithm which yields the most general type of a query along with its type schema requirements.

2 Core OQL

In this section we fix the syntax and type system for OQL. This is explained in greater detail in an earlier paper [3]; space restrictions mean that here we simply give the syntax for queries and definitions[1] in Figure 1. An OQL **program** consists of a number (maybe zero) of named definitions followed by a query.

The syntax for OQL types is also given in Figure 1. In what follows we will write $\mathrm{Col}(\sigma)$, to denote an arbitrary collection type (set, bag, list or array), with elements of type σ.

Implicit in the ODMG model is a notion of subtyping; the underlying idea is that σ is said to be a subtype of τ, if a value of type σ can be used in any context in which a value of type τ is expected. This we shall write $\sigma \leq \tau$ and define as the least relation closed under the rules given in Figure 1.

We use the $\sqsubseteq$ symbol to denote single inheritance between two classes, referred to in the Standard as the "*derives from*" relation. To simplify our presentation we do not consider interfaces.

An interesting feature of our subtype relation is the treatment of structures. A type $\sigma = \mathtt{struct}(\mathtt{l}_1\colon\sigma_1,\ldots,\mathtt{l}_\mathtt{m}\colon\sigma_\mathtt{m})$ is considered to be a subtype of $\tau = \mathtt{struct}(\mathtt{l}_1\colon\tau_1,\ldots,\mathtt{l}_\mathtt{n}\colon\tau_\mathtt{n})$ if τ is obtained from σ by dropping some labels. (In fact, we generalise this a little and also allow subtyping between the label types). This so-called width-subtyping is an extension to the Standard, but we feel it offers considerable flexibility.

The type system and the subtype relation are given in detail in an earlier paper [3]. In that work, we aimed at deriving the type of an OQL query given specific schema information. In order to do that, we defined typing judgements of the form:

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

where $\mathcal{S}$ are the class definitions, $\mathcal{D}$ are the persistent query definitions and $\mathcal{N}$ are the named objects of a specific schema. $\mathcal{Q}$ represents the query typing

[1] Naturally as we are interested in *inferring* types we drop the requirement that definition parameters be explicitly typed.

$$
\begin{array}{rcl}
\textbf{Queries}\ \mathtt{q} & ::= & b \mid f \mid i \mid c \mid s \\
& \mid & \mathtt{x} \\
& \mid & \mathtt{bag}(\mathtt{q},\dots,\mathtt{q}) \mid \mathtt{set}(\mathtt{q},\dots,\mathtt{q}) \mid \mathtt{list}(\mathtt{q},\dots,\mathtt{q}) \mid \mathtt{array}(\mathtt{q},\dots,\mathtt{q}) \\
& \mid & \mathtt{struct}(\mathtt{l}\colon \mathtt{q},\dots,\mathtt{l}\colon \mathtt{q}) \\
& \mid & \mathtt{C}(\mathtt{l}\colon \mathtt{q},\dots,\mathtt{l}\colon \mathtt{q}) \mid \mathtt{q.l} \mid (\mathtt{C})\mathtt{q} \\
& \mid & \mathtt{q}[\mathtt{q}] \mid \mathtt{q}\ \mathtt{in}\ \mathtt{q} \mid \mathtt{q}() \mid \mathtt{q}(\mathtt{q},\dots,\mathtt{q}) \\
& \mid & \mathtt{forall}\ \mathtt{x}\ \mathtt{in}\ \mathtt{q}\colon \mathtt{q} \mid \mathtt{exists}\ \mathtt{x}\ \mathtt{in}\ \mathtt{q}\colon \mathtt{q} \\
& \mid & \mathtt{q}\ \mathit{binop}\ \mathtt{q} \mid \mathit{unop}(\mathtt{q}) \\
& \mid & \mathtt{select}\ [\mathtt{distinct}]\ \mathtt{q} \\
& & \mathtt{from}\ (\mathtt{q}\ \mathtt{as}\ \mathtt{x},\cdots,\mathtt{q}\ \mathtt{as}\ \mathtt{x}) \\
& & \mathtt{where}\ \mathtt{q} \\
& & [\mathtt{group\ by}\ (\mathtt{l}\colon \mathtt{q},\cdots,\mathtt{l}\colon \mathtt{q})] \\
& & [\mathtt{having}\ \mathtt{q}] \\
& & [\mathtt{order\ by}\ (\mathtt{q}\ \mathtt{asc}|\mathtt{desc},\cdots,\mathtt{q}\ \mathtt{asc}|\mathtt{desc})]
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{Definitions}\ d & ::= & \mathtt{define}\ \mathtt{x}\ \mathtt{as}\ \mathtt{q} \\
& \mid & \mathtt{define}\ \mathtt{x}(\mathtt{x},\dots,\mathtt{x})\ \mathtt{as}\ \mathtt{q}
\end{array}
$$

Here b, f, i, c, s range over booleans, floats, integers, characters and strings respectively, $\mathtt{x}$ is taken from a countable set of identifiers, $\mathtt{l}$ is taken from a countable set of labels, and $\mathtt{C}$ ranges over a countable set of class names. We assume sets of unary and binary operators, ranged over by *unop* and *binop* respectively.

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

We assume a distinguished class name `Object`.

Sub-typing

$$
\frac{}{\mathtt{C} \leq \mathtt{Object}}\ \text{Top} \qquad \frac{\mathtt{C} \sqsubseteq \mathtt{C}'}{\mathtt{C} \leq \mathtt{C}'}\ \text{Sub-Class}
$$

$$
\frac{\sigma_1' \leq \sigma_1 \cdots \sigma_k' \leq \sigma_k \qquad \tau \leq \tau'}{\sigma_1 \times \cdots \times \sigma_k \to \tau \leq \sigma_1' \times \cdots \times \sigma_k' \to \tau'}\ \text{Sub-Fun} \qquad \frac{\sigma \leq \tau}{\text{Col}(\sigma) \leq \text{Col}(\tau)}\ \text{Sub-Coll}
$$

$$
\frac{\sigma_1 \leq \tau_1 \qquad \cdots \qquad \sigma_k \leq \tau_k}{\mathtt{struct}(\mathtt{l}_1\colon \sigma_1,\dots,\mathtt{l}_k\colon \sigma_k,\dots,\mathtt{l}_{k+n}\colon \sigma_{k+n}) \leq \mathtt{struct}(\mathtt{l}_1\colon \tau_1,\dots,\mathtt{l}_k\colon \tau_k)}\ \text{Sub-Struct}
$$

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

Fig. 1. Syntax, Types and Subtyping for Core OQL

environment, i.e. it contains the types of any free identifiers in q. A simple example of the typing rules used to derive the type of a query is the following:

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

In the current context, we have no information about the classes, the query definitions, and the named objects, as we have no schema information. The problem we address can thus be written $? \vdash \mathtt{q}\colon ?$, i.e. given an arbitrary query q, can we infer its type and the type of any supporting schemata?

3 Type and Schema Inference

In this section, we present the extended type system behind our inference algorithm and a new relation between types called *Generalisation-Specialisation* relation. We also discuss the notions of *Least Upper Bound* and *Greatest Lower Bound* of two types that occur frequently in our inference algorithm.

3.1 Extended Type System

It turns out to be convenient to extend the notion of type given in Figure 1; an example should make this clear. Consider the following:

```
define q1 as select x from Students as x;
define q2 as set(first(Students));
q1 union q2
```

Considering the query `q1 union q2` first, we can infer immediately that q1 and q2 should be either sets or bags of elements. To represent this we introduce a new type constructor set/bag(−). Moreover, the elements of this collection cannot be of a *function* type, since the Standard does not allow functions to be members of a collection; thus we introduce the types `nonfunctional` and `function`. From the definition q1 we infer that `Students` is some collection (set, bag, list or array) of elements of any (non-functional) type. Considering the definition q2 we can infer further information about `Students`. As it is the argument of a `first` operation it must be an *ordered* collection, i.e. a list or an array. Again we introduce a new type constructor list/array(−). In summary, the algorithm should infer that `Students` is a list or an array of a nonfunctional member type τ, and that the query `q1 union q2` is of type $\mathtt{bag}(\tau)$.[2]

The above example motivates our need to extend the initial *specific* types (given in Figure 1) with the following so-called *general* types.

$$\{\mathtt{any}, \mathtt{nonfunctional}, \mathtt{atomic}, \mathtt{orderable}, \mathtt{int/float}\} \cup$$
$$\{\mathtt{collection}(\tau), \mathtt{set/bag}(\tau), \mathtt{list/array}(\tau), \mathtt{constructor}(l_i\colon \tau_i)\} \cup$$
$$\{\texttt{all types from the core type system with at least one component type being a general type, e.g. set(any)}\}$$

[2] The Standard [§4.10.11] states that merging a set and a bag results in a bag.

where τ, τ_i are *specific* or *general* types.

Given these general types the resulting type system is then as follows:

$$
\begin{aligned}
\sigma ::= \; & \mathtt{int} \mid \mathtt{float} \mid \mathtt{bool} \mid \mathtt{char} \mid \mathtt{string} \mid \mathtt{void} \\
\mid \; & \sigma \times \cdots \times \sigma \rightarrow \sigma \\
\mid \; & \mathtt{bag}(\sigma) \mid \mathtt{set}(\sigma) \mid \mathtt{list}(\sigma) \mid \mathtt{array}(\sigma) \mid \mathtt{struct}(\mathtt{l}\colon \sigma, \cdots, \mathtt{l}\colon \sigma) \\
\mid \; & \mathtt{C} \\
\mid \; & \mathtt{any} \mid \mathtt{nonfunctional} \\
\mid \; & \mathtt{atomic} \mid \mathtt{orderable} \mid \mathtt{int/float} \\
\mid \; & \mathtt{constructor}(\mathtt{l}\colon \sigma, \cdots, \mathtt{l}\colon \sigma) \\
\mid \; & \mathtt{collection}(\sigma) \mid \mathtt{set/bag}(\sigma) \mid \mathtt{list/array}(\sigma)
\end{aligned}
\tag{1}
$$

This extended type system (1) is coupled with the type hierarchy illustrated in Figure 2. It is worth noting that the *general* types, which are the internal nodes of the tree, are not types that can be found in a database schema, but rather abstractions or families of types that encapsulate the common features of their children.

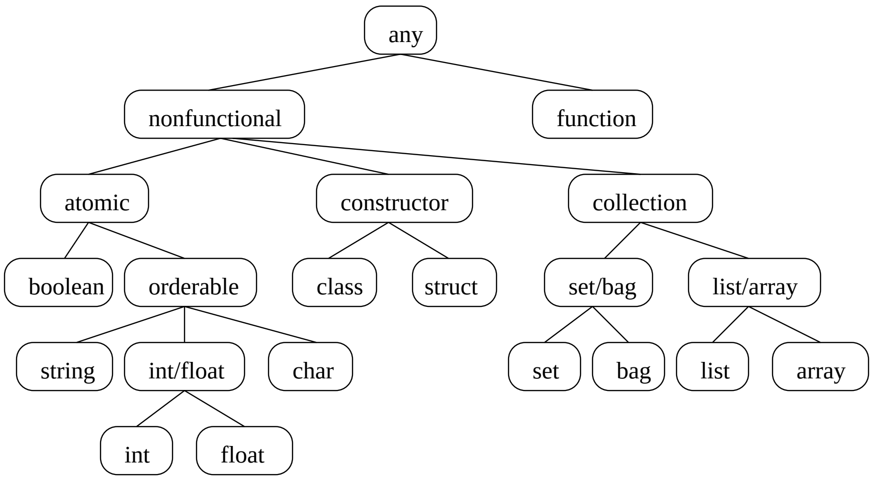

Fig. 2. Type Hierarchy

A type is said to be *specific*, if it can be derived by the type system given in Figure 1. Otherwise, it is said to be *general*. All the leaves of the hierarchy tree (in Figure 2) are specific types, if they are nullary (non parametric) types (`int`, `float`, `char`, `string`, `bool`) or if they are parametric types with all the parameter types being specific types (e.g. `set(int)`).

Given this more general type system, we need to extend our notion of subtyping given earlier. We define a new relation, GSR (Generalisation-Specialisation Relationship). Given types σ, τ, we write $\sigma \subseteq \tau$ to express that σ is *more specific* than τ.

$$\frac{\sigma \le \tau}{\sigma \subseteq \tau}\ \mathrm{GSR-Type} \qquad \frac{\sigma_1' \subseteq \sigma_1 \cdots \sigma_k' \subseteq \sigma_k \quad \tau \subseteq \tau'}{\sigma_1 \times \cdots \times \sigma_k \to \tau \subseteq \sigma_1' \times \cdots \times \sigma_k' \to \tau'}\ \mathrm{GSR-Fun}$$

$$\frac{\mathtt{coll_1\ is\ child\ of\ coll_2} \quad \sigma \subseteq \tau}{\mathtt{coll_1}(\sigma) \subseteq \mathtt{coll_2}(\tau)}\ \mathrm{GSR-Coll}$$

where $\mathtt{coll_1}, \mathtt{coll_2}$ are nodes in the sub-tree (figure 2) with root `collection` and `is child of` signifies that $\mathtt{coll_2}$ is a direct or indirect parent of $\mathtt{coll_1}$ or that $\mathtt{coll_1}$ and $\mathtt{coll_2}$ are the same.

$$\frac{\mathtt{constr_1\ is\ child\ of\ constr_2} \quad \sigma_1 \subseteq \tau_1 \cdots \sigma_k \subseteq \tau_k}{\mathtt{constr_1}(\mathtt{l_1}{:}\,\sigma_1, \ldots, \mathtt{l_k}{:}\,\sigma_k, \ldots, \mathtt{l_{k+n}}{:}\,\sigma_{k+n}) \subseteq \mathtt{constr_2}(\mathtt{l_1}{:}\,\tau_1, \ldots, \mathtt{l_k}{:}\,\tau_k)}\ \mathrm{GSR-Constr}$$

where $\mathtt{constr_1}, \mathtt{constr_2}$ are nodes in the sub-tree (figure 2) with root `constructor`, and `is child of` signifies that $\mathtt{constr_2}$ is a direct parent of $\mathtt{constr_1}$ or $\mathtt{constr_1}$ and $\mathtt{constr_2}$ are the same.

$$\frac{\mathtt{atom_1\ is\ child\ of\ atom_2}}{\mathtt{atom_1} \subseteq \mathtt{atom_2}}\ \mathrm{GSR-Atomic}$$

where $\mathtt{atom_1}$ and $\mathtt{atom_2}$ are nodes in the sub-tree (figure 2) with root `atomic` and `is child of` signifies that $\mathtt{atom_2}$ is direct or indirect parent of $\mathtt{atom_1}$ or $\mathtt{atom_1}$ and $\mathtt{atom_2}$ are the same.

$$\frac{}{\sigma \subseteq \sigma}\ \mathrm{GSR-Refl} \qquad \frac{\sigma_1 \subseteq \sigma_2 \quad \sigma_2 \subseteq \sigma_3}{\sigma_1 \subseteq \sigma_3}\ \mathrm{GSR-Trans}$$

$$\frac{\sigma \ne \tau_1 \to \tau_2}{\sigma \subseteq \mathtt{nonfunctional}}\ \mathrm{GSR-NonFun} \qquad \frac{}{\sigma \subseteq \mathtt{any}}\ \mathrm{GSR-All}$$

Given this definition, we can define the *Greatest Lower Bound* (`GLB`) and the *Least Upper Bound* (`LUB`) of two types τ_1 and τ_2. Insight into these concepts can be gained through the following simple example. Consider the types $\tau_1 = \mathtt{set(atomic)}$ and $\tau_2 = \mathtt{set/bag(int)}$. The `GLB` of the two types is derived by taking the most specific of the collection constructors, `set`, and the most specific of the parameter types, `int`. Thus $\mathtt{GLB}(\tau_1, \tau_2) = \mathtt{set(int)}$. Likewise, for the `LUB`, we take the most general of the two collection constructors, `set/bag`, and the most general of the two parameter types, `atomic`. Thus, $\mathtt{LUB}(\tau_1, \tau_2) = \mathtt{set/bag(atomic)}$.

We may now formally present `GLB` and `LUB`. In the following definitions, we assume that $\mathtt{constr_1}$, $\mathtt{constr_2}$ are nodes in the sub-tree (figure 2) with root `constructor` and that $\mathtt{constr_1}$ is a child of $\mathtt{constr_2}$ or $\mathtt{constr_1} = \mathtt{constr_2}$. Moreover, $\mathtt{coll_1}$ and $\mathtt{coll_2}$ are nodes in the sub-tree (figure 2) with root `collection` and $\mathtt{coll_1}$ is a child of $\mathtt{coll_2}$ or $\mathtt{coll_1} = \mathtt{coll_2}$.

$$\mathtt{GLB}(\tau_1,\tau_2) \stackrel{\text{def}}{=} \tau_1 \text{ } \mathtt{if} \text{ } \tau_1 \subseteq \tau_2 \wedge \tau_2 \subseteq \mathtt{atomic}$$

$$\mathtt{LUB}(\tau_1,\tau_2) \stackrel{\text{def}}{=} \tau \text{ if } \tau_1 \subseteq \mathtt{atomic} \wedge \tau_2 \subseteq \mathtt{atomic} \wedge \\ \text{(there exists no } \tau' \text{ s.t. } (\tau' \neq \tau \wedge \tau_1 \subseteq \tau' \wedge \tau_2 \subseteq \tau' \wedge \tau \subseteq \tau'))$$

$$\mathtt{GLB}(\mathtt{constr}_1(\mathtt{l}_1{:}\,\sigma_1,\ldots,\mathtt{l}_\mathtt{k}{:}\,\sigma_\mathtt{k}), \mathtt{constr}_2(\mathtt{l}_1{:}\,\sigma'_1,\ldots,\mathtt{l}_\mathtt{k}{:}\,\sigma'_\mathtt{k},\ldots,\mathtt{l}_{\mathtt{k+n}}{:}\,\sigma'_{\mathtt{k+n}})) \stackrel{\text{def}}{=} \\ \mathtt{constr}_1(\mathtt{l}_1{:}\,\mathtt{GLB}(\sigma_1,\sigma'_1),\ldots,\mathtt{l}_\mathtt{k}{:}\,\mathtt{GLB}(\sigma_\mathtt{k},\sigma'_\mathtt{k}),\ldots,\mathtt{l}_{\mathtt{k+n}}{:}\,\sigma_{\mathtt{k+n}})$$

$$\mathtt{LUB}(\mathtt{constr}_1(\mathtt{l}_1{:}\,\sigma_1,\ldots,\mathtt{l}_\mathtt{k}{:}\,\sigma_\mathtt{k}), \mathtt{constr}_2(\mathtt{l}_1{:}\,\sigma'_1,\ldots,\mathtt{l}_\mathtt{k}{:}\,\sigma'_\mathtt{k},\ldots,\mathtt{l}_{\mathtt{k+n}}{:}\,\sigma'_{\mathtt{k+n}})) \stackrel{\text{def}}{=} \\ \mathtt{constr}_2(\mathtt{l}_1{:}\,\mathtt{LUB}(\sigma_1,\sigma'_1),\ldots,\mathtt{l}_\mathtt{k}{:}\,\mathtt{LUB}(\sigma_\mathtt{k},\sigma'_\mathtt{k}))$$

$$\mathtt{GLB}(\mathtt{coll}_1(\sigma_1),\mathtt{coll}_2(\sigma_2)) \stackrel{\text{def}}{=} \mathtt{coll}_1(\mathtt{GLB}(\sigma_1,\sigma_2))$$

$$\mathtt{LUB}(\mathtt{coll}_1(\sigma_1),\mathtt{coll}_2(\sigma_2)) \stackrel{\text{def}}{=} \mathtt{coll}_2(\mathtt{LUB}(\sigma_1,\sigma_2))$$

$$\mathtt{GLB}(\tau_1 \times \cdots \times \tau_\mathtt{k} \to \sigma, \tau'_1 \times \cdots \times \tau'_\mathtt{k} \to \sigma') \stackrel{\text{def}}{=} \mathtt{LUB}(\tau_1,\tau'_1) \times \cdots \times \mathtt{LUB}(\tau_\mathtt{k},\tau'_\mathtt{k}) \to \mathtt{GLB}(\sigma,\sigma')$$

$$\mathtt{LUB}(\tau_1 \times \cdots \times \tau_\mathtt{k} \to \sigma, \tau'_1 \times \cdots \times \tau'_\mathtt{k} \to \sigma') \stackrel{\text{def}}{=} \mathtt{GLB}(\tau_1,\tau'_1) \times \cdots \times \mathtt{LUB}(\tau_\mathtt{k},\tau'_\mathtt{k}) \to \mathtt{LUB}(\sigma,\sigma')$$

$$\mathtt{LUB}(\sigma,\tau) \stackrel{\text{def}}{=} \mathtt{any}, \text{ if } \sigma \subseteq \mathtt{function} \wedge \tau \subseteq \mathtt{nonfunctional}$$

$$\mathtt{LUB}(\sigma_1,\sigma_2) \stackrel{\text{def}}{=} \mathtt{nonfunctional}, \text{ if } \forall \tau_1,\tau_2.\sigma_1 \subseteq \tau_1 \wedge \sigma_2 \subseteq \tau_2 \wedge \tau_1 \neq \tau_2 \wedge \\ \tau_1,\tau_2 \in \{\mathtt{atomic},\mathtt{constructor}(),\mathtt{collection}(\mathtt{any})\}$$

4 Type compatibility - Constraints

The inference algorithm we present in section 7 analyses an OQL construct and infers the most general type of the query and the schema requirements that should be satisfied so that the query is well-typed. Before being able to present the inference algorithm, we first discuss an important mechanism that the algorithm is based upon—the generation of constraints. When the inference algorithm analyses a certain query construct, it often infers several relations or associations amongst the types of the query and its subqueries. These associations are given in the form of constraints.

For example, the analysis of a query $\mathtt{q}_1$ `union` $\mathtt{q}_2$ would generate the constraint that the type τ of the query is the merge result of the types τ_1 and τ_2 of the two subqueries, i.e. $\tau = \mathtt{Merge\_Result}(\tau_1,\tau_2)$. In section 5, we show how this constraint is simplified and is *assimilated* in the set of the existing constraints.

Analysing the query `exists x in Customers: x.income` $> 40,000$ our algorithm generates the following constraints. First, it introduces the constraint $\tau_1 = \mathtt{Member\_Type}(\tau_2)$, where τ_1 is the type of `x` and τ_2 is the type of `Customers`. The type of `x` is expected to be the same as that of the members of the collection `Customers`. Second, the constraint $\tau_3 = \mathtt{Constructor\_Member\_Type}(\tau_1, \mathtt{income})$ is generated, where τ_3 is the type of `x.income`. This signifies that `x` is a constructor type (a structure or a class) with at least one member `income` of type τ_3. Another interesting constraint is that the types of `x.income` (τ_3) and of the literal $40,000$ (`int`) should be compatible in the sense that they than can be compared for inequality. This is expressed by the constraint $\texttt{Greater\_Less\_Than\_Compatible}(\tau_3, \texttt{int})$. Later, we will show how this constraint is simplified to the constraint $\tau_3 \subseteq$ $\texttt{int/float}$.

In section 6, we give a set of inference rules, one for each query construct. Each rule starts with an existing set of constraints and generates a number (possibly zero) of new constraints. The different kinds of constraints generated by the rules in our inference algorithm are given below:

1. $\texttt{Equality\_Compatible}(\tau_1, \ldots, \tau_n)$	6. $\tau_0 = \texttt{Merge\_Result}(\tau_1, \tau_2)$
2. $\texttt{Greater\_Less\_Than\_Compatible}(\tau_1, \ldots, \tau_n)$	7. $\tau_0 = \texttt{Distinct\_Result}(\tau_1)$
3. $\tau_1 = \tau_2$	8. $\tau = \texttt{Member\_Type}(\sigma)$
4. $\tau_1 \subseteq \tau_2$	9. $\tau = \texttt{Constructor\_Member\_Type}(\sigma, \texttt{l})$
5. $\tau_0 = \texttt{Arith\_Result}(\tau_1, \tau_2)$	

We briefly explain the constraints used in our inference model. The constraint $\texttt{Equality\_Compatible}(\texttt{q}_1, \ldots, \texttt{q}_n)$ is analysed in section 4.1. The constraint $\texttt{Greater\_Less\_Than\_Compatible}(\texttt{q}_1, \texttt{q}_2)$ is useful for ensuring typability for queries like $\texttt{q}_1 < \texttt{q}_2$. Type equality, and GSR (Generalisation-Specialisation Relationship) are handled by constraints 3 and 4. $\tau_0 = \texttt{Arith\_Result}(\tau_1, \tau_2)$ is needed for the type inference of queries of the form $\texttt{q}_1\, op\, \texttt{q}_2$ where $op \in \{+, -, /, *\}$. Likewise, $\tau_0 = \texttt{Merge\_Result}(\tau_1, \tau_2)$ arises as a constraint from inferencing the type of `union`, `intersect` or `except` query expressions. The constraint $\tau_0 = \texttt{Distinct\_Result}(\tau_1)$ implies that both τ_0 and τ_1 are collection types and the collection constructor of τ_0 is the distinct equivalent of the collection constructor of τ_1. Moreover, $\tau = \texttt{Member\_Type}(\sigma)$ implies that σ is a collection type and that τ is the type of its members. Finally, the constraint $\tau = \texttt{Constructor\_Member\_Type}(\sigma, \texttt{l})$ is used to denote that σ is a `class` or `struct` type with at least one member `l` of type τ. If σ is a `class` type then `l` can be any of its properties, relationships or methods.

4.1 Collections - Membership Type Compatibility

The first two constraints refer to type compatibility w.r.t. equality or non-equality comparison. These constraints arise in OQL constructs that involve a merge of two or more elements, or a membership test. For instance, the first constraint results from considering a query of the form $\texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_n)$, which includes the merge of n query results.

First of all, we should stress the fact that in order for two values (objects or literals) to be eligible as members of the same collection, they should be eligible for *equality comparison*. If two types are compatible (membership-wise), two values of these types may be members of a set. In order to insert an element into a set, we need to test if its value is equal to any existing value. Thus, we need to ensure that these values have types which are compatible (equality-wise). Inversely, if two types are compatible equality-wise, then their values may be inserted into any collection, therefore these types are also compatible membership-wise.

The Standard [§4.10] defines recursively when two types are compatible, and thus when elements of these types can be put in the same collection. The Standard then defines the notion of least upper bound (LUB) of two types to derive the type of the collection elements. In a context where we need to check and derive the type of a query based on *specific* type information (from a schema), this approach is sufficient and straightforward. However, in our context, where we aim to infer the type of a query without any schema information, the compatibility issue becomes more complicated. The use of LUB to infer the type of a query like $\mathtt{set}(\mathtt{q_1}, \ldots, \mathtt{q_n})$ does not yield the appropriate result, for example consider the following.

```
define q1 as struct(x:12,y:30);
define q2 as element(select z from People as z where z.x=14);
set(q1,q2)
```

If we call the inference algorithm on q1 and q2 the inferred types (IT) would be $\mathtt{struct}(\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int})$ and $\mathtt{constructor}(\mathtt{x} : \mathtt{int})$ respectively. The least upper bound of these two types is $\mathtt{constructor}(\mathtt{x} : \mathtt{int})$. Thus, the inferred type of the query set(q1,q2) would be $\mathtt{set}(\mathtt{constructor}(\mathtt{x} : \mathtt{int}))$.

However, the correct inferred type should be $\mathtt{set}(\mathtt{struct}(\mathtt{x} : \mathtt{int}))$, since we may not merge objects and structures in the same collection, and therefore we know that the constructor should be a struct and not a class.

To overcome this problem we define another relation between types, namely CUB (*Compatible Upper Bound*). Intuitively, CUB combines the behaviour of both LUB and GLB (Greatest Lower Bound). In the previous example, the CUB of the two types $\mathtt{IT}(\mathtt{q_1})$ and $\mathtt{IT}(\mathtt{q_2})$ would be derived by taking the most specific of the two constructor types (constructor and struct), but the least general of the element types ($(\mathtt{x} : \mathtt{int})$ and $(\mathtt{x} : \mathtt{int}, \mathtt{y} : \mathtt{int})$). Before we define CUB, we define *compatibility* (Membership- or Equality- wise) for our typing system.

Compatibility is recursively defined as follows:

- τ is compatible with τ
- if σ is compatible with τ and $\mathtt{coll_1}, \mathtt{coll_2} \in \{\mathtt{collection}, \mathtt{set/bag}, \mathtt{list/array}, \mathtt{set}, \mathtt{bag}, \mathtt{list}, \mathtt{array}\}$ and either the collection constructors are the same or one is child of the other in the hierarchy tree then $\mathtt{coll_1}(\sigma)$ is compatible with $\mathtt{coll_2}(\tau)$.
- Any two class types $\mathtt{class\_name_1}$ and $\mathtt{class\_name_2}$ are compatible.
- If σ_i is compatible with τ_i, $\forall\ i = 1, \ldots, n$ and $\mathtt{constr_1}, \mathtt{constr_2} \in \{\mathtt{constructor}, \mathtt{struct}\}$ and no labels other than $\mathtt{l_1}, \ldots, \mathtt{l_n}$ are common in both constructor types then $\mathtt{constr_1}(\mathtt{l_1}{:}\, \sigma_1, \ldots, \mathtt{l_n}{:}\, \sigma_\mathtt{n}, \mathtt{l_{11}}{:}\, \sigma_{11}, \ldots, \mathtt{l_{1k}}{:}\, \sigma_\mathtt{1k})$ and $\mathtt{constr_2}(\mathtt{l_1}{:}\, \tau_1, \ldots, \mathtt{l_n}{:}\, \tau_\mathtt{n}, \mathtt{l_{21}}{:}\, \tau_{21}, \ldots, \mathtt{l_{2m}}{:}\, \tau_\mathtt{2m})$ are compatible.
- If σ_i is compatible with τ_i, $\forall\ \mathtt{i} = 1, \ldots, \mathtt{n}$ and class_name is a class type such that $\mathtt{Constructor\_Member\_Type}(\mathtt{class\_name}, \mathtt{l_i}) = \tau_\mathtt{i}$, $\forall\ \mathtt{i} = 1, \ldots, \mathtt{n}$, then the types class_name and $\mathtt{constructor}(\mathtt{l_1}{:}\, \sigma_1, \ldots, \mathtt{l_n}{:}\, \sigma_\mathtt{n}, \mathtt{l_{11}}{:}\, \sigma_{11}, \ldots, \mathtt{l_{1k}}{:}\, \sigma_\mathtt{1k})$ are compatible, provided that no labels other than $\mathtt{l_1}, \ldots, \mathtt{l_n}$ are common members in the two types.
- If $\sigma, \tau \in \{\mathtt{atomic}, \mathtt{orderable}, \mathtt{int/float}, \mathtt{int}, \mathtt{float}, \mathtt{char}, \mathtt{string}, \mathtt{bool}\}$ and either they are the same, or one is a child of the other in the hierarchy

tree, or one is $\texttt{int}$ and the other is $\texttt{float}$
then σ is compatible with τ.

- If either of the types is $\texttt{nonfunctional}$ and the other type is not a function type then these types are compatible.
- If either of two types is $\texttt{any}$, then these types are compatible.

Note that we do not define compatibility for function types, as no two function values may be members of the same collection or may be compared for equality. Only the results of function application may be considered for compatibility.

Given that two types are compatible (based on the recursive definition above), their $\texttt{CUB}$ is defined recursively and in accordance with the compatibility category that they fall into.

- $\texttt{CUB}(\tau, \tau) = \tau$.
- If $\texttt{coll}_1(\sigma)$ is compatible with $\texttt{coll}_2(\tau)$ and $\texttt{coll}_1$ is a child of (or the same as) $\texttt{coll}_2$, then $\texttt{CUB}(\texttt{coll}_1(\sigma), \texttt{coll}_2(\tau)) = \texttt{coll}_1(\texttt{CUB}(\sigma, \tau))$.
- If the types τ_1 and τ_2 are class types, then $\texttt{CUB}(\tau_1, \tau_2)$ is the least common superclass of the two classes.
- If the types $\sigma = \texttt{constr}_1(\texttt{l}_1\colon \sigma_1, \ldots, \texttt{l}_\texttt{n}\colon \sigma_\texttt{n}, \texttt{l}_{11}\colon \sigma_{11}, \ldots, \texttt{l}_{1\texttt{k}}\colon \sigma_{1\texttt{k}})$ and $\tau = \texttt{constr}_2(\texttt{l}_1\colon \tau_1, \ldots, \texttt{l}_\texttt{n}\colon \tau_\texttt{n}, \texttt{l}_{21}\colon \tau_{21}, \ldots, \texttt{l}_{2\texttt{m}}\colon \tau_{2\texttt{m}})$ are compatible, where $\texttt{constr}_1, \texttt{constr}_2 \in \{\texttt{constructor}, \texttt{struct}\}$ then $\texttt{CUB}(\sigma, \tau) = \texttt{constr}_1(\texttt{l}_1\colon \texttt{CUB}(\sigma_1, \tau_1), \ldots, \texttt{l}_\texttt{n}\colon \texttt{CUB}(\sigma_\texttt{n}, \tau_\texttt{n}))$.
- If $\sigma = \texttt{class\_name}_1$ and $\texttt{Constructor\_Member\_Type}(\texttt{class\_name}_1, \texttt{l}_\texttt{i}) = \sigma_\texttt{i}$, $\forall\ \texttt{i} = 1, \ldots, \texttt{n}$ and $\tau = \texttt{constructor}(\texttt{l}_1\colon \tau_1, \ldots, \texttt{l}_\texttt{n}\colon \tau_\texttt{n}, \texttt{l}_{21}\colon \tau_{21}, \ldots, \texttt{l}_{2\texttt{m}}\colon \tau_{2\texttt{m}})$ then $\texttt{CUB}(\sigma, \tau)$ is the least superclass of $\texttt{class\_name}_1$, say $\texttt{class\_name}_2$, satisfying the following condition: For all $\texttt{l}'_\texttt{j}$, $\texttt{l}'_\texttt{j}$ is a property or a relationship of $\texttt{class\_name}_2$, if $\texttt{Constructor\_Member\_Type}(\texttt{class\_name}_2, \texttt{l}'_\texttt{j}) = \phi_\texttt{j}$ then there exists $\texttt{k}$, $1 \leq \texttt{k} \leq \texttt{n}$, s.t. $\texttt{l}'_\texttt{j} = \texttt{l}_\texttt{k} \wedge \texttt{CUP}(\tau_\texttt{k}, \sigma_\texttt{k}) \subseteq \phi_\texttt{j}, \forall\ \texttt{j} = 1, \ldots, \texttt{m},\ \texttt{m} \leq \texttt{n}$.
- If σ is compatible with τ, then
  1. if either of them is $\texttt{int/float}$ or one is $\texttt{int}$ and the other is $\texttt{float}$ then $\texttt{CUB}(\sigma, \tau) = \texttt{int/float}$
  2. else
     $\texttt{CUB}(\sigma, \tau) = \texttt{GLB}(\sigma, \tau)$.
- If $\sigma = \texttt{nonfunctional}$ and $\tau \subseteq \texttt{nonfunctional}$ then $\texttt{CUB}(\sigma, \tau) = \tau$.
- If $\sigma = \texttt{any}$ then, for any type τ, $\texttt{CUB}(\sigma, \tau) = \tau$.

As discussed earlier, the notion of $\texttt{CUB}$ is used for the inference of the types of queries like $\texttt{set}(\texttt{q}_1, \ldots, \texttt{q}_\texttt{n})$. However, the OQL construct $\texttt{q}_1\ \texttt{in}\ \texttt{q}_2$ raises another issue of a slighly different nature. The Standard [§4.10.8.3] states that if the type of $\texttt{q}_2$ is $\texttt{coll}(\tau)$ then the type of $\texttt{q}_1$ should be τ. This is not the case in our context. Suppose that the type of $\texttt{q}_2$ is inferred to be $\texttt{bag}(\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{string}))$; then according to the Standard $\texttt{q}_1$ should have the type $\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{string})$. Since a value of type $\texttt{struct}(\texttt{x}\colon \texttt{int})$ could potentially be added in the collection $\texttt{q}_2$ (that is, since $\texttt{struct}(\texttt{x}\colon \texttt{int})$ and $\texttt{struct}(\texttt{x}\colon \texttt{int}, \texttt{y}\colon \texttt{float})$ are compatible types), there is no reason why $\texttt{q}_1$ could not be of type $\texttt{struct}(\texttt{x}\colon \texttt{int})$ or even $\texttt{struct}()$.

The same situation occurs when dealing with a collection of objects of different classes. Suppose $\mathtt{lub\_class}(\mathtt{l}_1\!:\sigma_1,\ldots,\mathtt{l}_n\!:\sigma_n)$ is the LUB of all classes of the objects in the collection and Object is the most general class that all other classes derive from (the top of the class hierarchy). We should be able to check whether an object of type $\mathtt{object\_class}(\mathtt{l}'_1\!:\sigma'_1,\ldots,\mathtt{l}'_m\!:\sigma'_m)$ is a member of the collection, even if its class is not a subclass of $\mathtt{lub\_class}(\mathtt{l}_1\!:\sigma_1,\ldots,\mathtt{l}_n\!:\sigma_n)$. This allows more queries to (safely) type-check, for example:

```
select x
from   People as x
where  x.father in School_Teachers
```

does not type-check according to the Standard. In order to be type-correct, `x.father` requires an explicit type cast, i.e. `(School_Teacher)x.father`. The problem is that this query, despite being well-typed, can generate a run-time error (if the cast does not succeed). We choose not to enforce that `x.father` has a type which is more specific than the member type or the collection `School_Teachers`. Rather, we simply ensure that `x.father` could potentially be a member of `School_Teachers`. To do this we add the constraint $\mathtt{Equality\_Compatible}(\sigma,\tau)$, where σ is the type of `x.father` and τ is the member type of `School_Teachers`.

5 Resolving constraints

Now that we have studied the kinds of constraints that are generated by our inference algorithm, we can discuss how these constraints are resolved. When a constraint is generated by an inference rule, it is added to the set of existing constraints. If this was a simple insertion procedure, we would end up having a huge set of constraints, that would include redundant and often incomprehensible type information; there is obviously a need to resolve the inserted constraints. Due to the complexity of the type system and the expressiveness of the language, we have a wide variety of constraints, that cannot be solved using a standard unification mechanism alone [10]. In our system, the insertion of a new constraint in a set of existing constraints may have one of the following effects:

- A constraint is deleted, if it is always satisfied, e.g. the constraint
  $\mathtt{Equality\_Compatible}(\mathtt{set}(\mathtt{int}),\mathtt{set}(\mathtt{float}))$
  is always true, so it does not need to be maintained.
- A constraint raises a type error or exception, if it is never satisfied, e.g. $\mathtt{set}(\mathtt{Employee}(\mathtt{name}:\mathtt{string})) \subseteq \mathtt{list/array}(\mathtt{Employee}(\mathtt{name}:\mathtt{string}))$.
- A constraint might be maintained as it is. This usually occurs when some of the types involved are *general*; it may be that when refined, these types no longer satisfy the constraint. Therefore, they must be preserved as required schema information. For instance, if $\tau_\mathtt{i} \subseteq \mathtt{set/bag}(\tau) \forall\ \mathtt{i} = 0,1,2$ are in the set of already produced constraints, the constraint $\tau_0 = \mathtt{Merge\_Result}(\tau_1,\tau_2)$ needs to be preserved.
- A constraint is often simplified, i.e. replaced by one or more simpler constraints. For instance, the constraint $\mathtt{set}(\sigma) = \mathtt{set}(\tau)$ is replaced by the simpler one $\sigma = \tau$.

– A constraint occasionally implies one or more constraints. The latter need to be added to the set of constraints already produced. For example, the constraint $\texttt{Greater\_Less\_Than\_Compatible}(\tau_1, \tau_2)$ is inserted as a new constraint along with the implied constraints $\tau_1 \subseteq \texttt{orderable}$ and $\tau_2 \subseteq \texttt{orderable}$.

The effect varies depending on the constraint kind, the types involved in the constraint and the already existing constraints on these types. The details of the constraint resolution algorithm will appear in [11].

It is worth pointing out that the resolution of constraints could take place either at the time each constraint is generated (*gradual resolution*) or at the time all the constraints have been produced (*accumulative resolution*).

The *gradual resolution* is very simple, since it usually concerns the insertion of a few constraints whose simplification (*unification*) is straightforward. If their simplification produces new constraints then these are simplified as well, until no more constraints are produced.

The *accumulative resolution* starts from the constraints of the form $\tau_1 = \tau_2$. It simplifies them to constraints of the form $\texttt{type\_var} = \tau$ and replaces `type_var` by τ in all other constraints that involve `type_var`. Then it proceeds to simplify all other kinds of constraints. If a simplification leads to more constraints, the latter are added to the set of unprocessed constraints and are simplified in due course.

6 Type Inference Rules

Having explained the type system underlying our inference model and the various constraints generated and resolved by our algorithm, we are now in a position to present the backbone of our work, the *inference rules*. Note that there is a single rule for each OQL construct, and, therefore, the use of the rules by the inference algorithm is syntax driven. In the remainder of this section, we present a substantial part of the inference rules; the complete set will be given in [11].

In the following rules, $\mathcal{H}$ signifies the *type environment*, that is $\mathcal{H} = \{var_i : \tau_i\}$, and $\mathcal{C}$ denotes the *constraints* added so far. The inference rules for the literal and the identifier queries are given first:

$$\frac{}{\mathcal{H};\mathcal{C} \vdash b\colon \texttt{bool} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash i\colon \texttt{int} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash f\colon \texttt{float} \Rightarrow \mathcal{C}}$$

$$\frac{}{\mathcal{H};\mathcal{C} \vdash c\colon \texttt{char} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H};\mathcal{C} \vdash s\colon \texttt{string} \Rightarrow \mathcal{C}} \qquad \frac{}{\mathcal{H} \cup \{\texttt{x}\colon \sigma\};\mathcal{C} \vdash \texttt{x}\colon \sigma \Rightarrow \mathcal{C}}$$

There are several rules to deal with various collections (sets, bags, lists, arrays). We just give one representative rule, that concerns the query construct $\texttt{set}(\mathtt{q}_1, \ldots, \mathtt{q}_\mathtt{n})$. As expected, the rule generates a constraint that ensures that the types of the queries are compatible equality-wise (or membership-wise). We also give the rules for accessing the first, last or i-th member of an ordered collection, as well as checking whether an element belongs in a certain collection.

The constraint $\texttt{Member\_Type}(\sigma) = \tau$ denotes that σ is a collection (set, bag, list or array) with members of type τ.

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \ \ldots\ \mathcal{H};\mathcal{C}_{n-1} \vdash \texttt{q}_\texttt{n} : \sigma_\texttt{n} \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \texttt{set}(\texttt{q}_1,\ldots,\texttt{q}_\texttt{n}) : \texttt{set}(\texttt{CUB}(\sigma_1,\ldots,\sigma_\texttt{n})) \Rightarrow \mathcal{C}_n \wedge \{\texttt{Equality\_Compatible}(\sigma_1,\ldots,\sigma_\texttt{n})\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \texttt{first}(\texttt{q}_1) : \phi \Rightarrow \mathcal{C}_1 \wedge \{\texttt{Member\_Type}(\sigma) = \phi\} \wedge \{\sigma \subseteq \texttt{list/array}(\phi)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \texttt{q}_2 : \tau \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1[\texttt{q}_2] : \phi \Rightarrow \mathcal{C}_2 \wedge \{\tau = \texttt{int}\} \wedge \{\texttt{Member\_Type}(\sigma) = \phi\} \wedge \{\sigma \subseteq \texttt{list/array}(\phi)\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma \Rightarrow \mathcal{C}_1 \quad \mathcal{H};\mathcal{C}_1 \vdash \texttt{q}_2 : \tau \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 \texttt{ in } \texttt{q}_2 : \texttt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\texttt{Equality\_Compatible}(\sigma, \texttt{Member\_Type}(\tau))\}}$$

The rules for constructing a structure or an object, as well as for accessing a member of a structure or an object are given below. The constraint $\texttt{Constructor\_Member\_Type}(\sigma, \ell) = \tau$ denotes that type σ is a class or a structure with a member called ℓ of type τ.

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \ \ldots\ \mathcal{H};\mathcal{C}_{n-1} \vdash \texttt{q}_\texttt{n} : \sigma_\texttt{n} \Rightarrow \mathcal{C}_n}{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \texttt{class\_name}(\texttt{l}_1 : \texttt{q}_1,\ldots,\texttt{l}_\texttt{n} : \texttt{q}_\texttt{n}) : \texttt{class\_name} \Rightarrow \mathcal{C}_n \wedge \\ \{\sigma_1 \subseteq \texttt{Constructor\_Member\_Type}(\texttt{class\_name}, \texttt{l}_1)\} \wedge \ldots \wedge \\ \{\sigma_\texttt{n} \subseteq \texttt{Constructor\_Member\_Type}(\texttt{class\_name}, \texttt{l}_\texttt{n})\}\end{array}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \ \ldots\ \mathcal{H};\mathcal{C}_{n-1} \vdash \texttt{q}_\texttt{n} : \sigma_\texttt{n} \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \texttt{struct}(\texttt{l}_1 : \texttt{q}_1,\ldots,\texttt{l}_\texttt{n} : \texttt{q}_\texttt{n}) : \texttt{struct}(\texttt{l}_1 : \sigma_1,\ldots,\texttt{l}_\texttt{n} : \sigma_\texttt{n}) \Rightarrow \mathcal{C}_n}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \tau \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1.\texttt{l} : \sigma \Rightarrow \mathcal{C}_1 \wedge \{\texttt{Constructor\_Member\_Type}(\tau, \texttt{l}) : \sigma\}}$$

The inference rules for the existential and the universal quantification follow. It is worth noting that the variable $\texttt{x}$ is bound in query $\texttt{q}_2$ to the member type of the collection $\texttt{q}_1$.

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H} \cup \{\texttt{x} : \phi\};\mathcal{C}_1 \vdash \texttt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \texttt{exists x in } \texttt{q}_1 \texttt{: } \texttt{q}_2 : \texttt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\sigma_2 = \texttt{bool}\} \wedge \{\texttt{Member\_Type}(\sigma_1) = \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \texttt{q}_1 : \sigma_1 \Rightarrow \mathcal{C}_1 \quad \mathcal{H} \cup \{\texttt{x} : \phi\};\mathcal{C}_1 \vdash \texttt{q}_2 : \sigma_2 \Rightarrow \mathcal{C}_2}{\mathcal{H};\mathcal{C} \vdash \texttt{forall x in } \texttt{q}_1 \texttt{: } \texttt{q}_2 : \texttt{bool} \Rightarrow \mathcal{C}_2 \wedge \{\sigma_2 = \texttt{bool}\} \wedge \{\texttt{Member\_Type}(\sigma_1) = \phi\}}$$

An interesting set of rules concerns the application of methods with or without parameters. The inferred types of the queries used as arguments are not constrained to be the same as the types of the parameters of the method involved. They only need to be their subtypes.

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1}\colon \sigma \Rightarrow \mathcal{C}_1}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_1}()\colon \phi \Rightarrow \mathcal{C}_1 \wedge \{\sigma = \mathtt{unit} \rightarrow \phi\}}$$

$$\frac{\mathcal{H};\mathcal{C} \vdash \mathtt{q_0}\colon \sigma_0 \Rightarrow \mathcal{C}_0 \ \ldots\ \mathcal{H};\mathcal{C}_{n-1} \vdash \mathtt{q_n}\colon \sigma_\mathtt{n} \Rightarrow \mathcal{C}_n}{\mathcal{H};\mathcal{C} \vdash \mathtt{q_0}(\mathtt{q_1},\ldots,\mathtt{q_n})\colon \phi \Rightarrow \mathcal{C}_n \wedge \{\sigma_0 = \tau_1 \times \ldots \times \tau_n \rightarrow \phi\} \wedge \{\sigma_1 \subseteq \tau_1\} \wedge \ldots \wedge \{\sigma_\mathtt{n} \subseteq \tau_\mathtt{n}\}}$$

The rules concerning the query constructs `q binop q` or `unop q` are omitted for space reasons. We finish by giving the rule for a simple `select` query; the judgements dealing with a `group by` or an `order by` clause can be found in [11].

$$\frac{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{q_1}\colon \sigma_1 \Rightarrow \mathcal{C}_1 \\ \mathcal{H} \cup \{\mathtt{x_1}\colon \tau_1\};\mathcal{C}_1 \vdash \mathtt{q_2}\colon \sigma_2 \Rightarrow \mathcal{C}_2 \\ \ldots \\ \mathcal{H} \cup \{\mathtt{x_1}\colon \tau_1,\ldots,\mathtt{x_{n-1}}\colon \tau_{\mathtt{n-1}}\};\mathcal{C}_{n-1} \vdash \mathtt{q_n}\colon \sigma_\mathtt{n} \Rightarrow \mathcal{C}_n \\ \mathcal{H} \cup \{\mathtt{x_1}\colon \tau_1,\ldots,\mathtt{x_n}\colon \tau_\mathtt{n}\};\mathcal{C}_n \vdash \mathtt{q_{01}}\colon \sigma_{01} \Rightarrow \mathcal{C}_{01} \\ \mathcal{H} \cup \{\mathtt{x_1}\colon \tau_1,\ldots,\mathtt{x_n}\colon \tau_\mathtt{n}\};\mathcal{C}_{01} \vdash \mathtt{q_{00}}\colon \sigma_{00} \Rightarrow \mathcal{C}_{00}\end{array}}{\begin{array}{c}\mathcal{H};\mathcal{C} \vdash \mathtt{select}\ \mathtt{q_{00}}\ \mathtt{from}\ \mathtt{q_1}\ \mathtt{as}\ \mathtt{x_1},\ldots,\mathtt{q_n}\ \mathtt{as}\ \mathtt{x_n}\ \mathtt{where}\ \mathtt{q_{01}}\colon \mathtt{bag}(\sigma_{00}) \Rightarrow \mathcal{C}_{00} \wedge \\ \{\mathtt{Member\_Type}(\sigma_1) = \tau_1\} \wedge \ldots \wedge \{\mathtt{Member\_Type}(\sigma_\mathtt{n}) = \tau_\mathtt{n}\} \wedge \{\sigma_{01} = \mathtt{bool}\}\end{array}}$$

7 Inference Algorithm

Having given an overview of the type system, the constraints and the rules involved in our inference model, we may now present the core of our work, which is the inference algorithm. The algorithm takes as input a query `q` and returns its inferred type, as well as a pair $(\mathcal{H},\mathcal{C})$ of a type environment and its constraints. This pair is a synopsis of the requirements a schema should satisfy so that the query `q` can be executed against it without any type-errors.

1. For each free variable `var` in the query q, $\mathcal{H} = \mathcal{H} \cup \{\mathtt{var}\colon \mathtt{new\_type\_var}\}$. Initially $\mathcal{C} = \{\}$.
2. Based on the construct of the query `q` recursively apply the appropriate inference rule.
3. Depending on the unification strategy, either simplify the constraints as soon as they are produced (gradual unification) or simplify them all in the end after having applied all the inference rules. If the unification process produces a type-error then the query is not typable and the algorithm is interrupted.
4. The final $\mathcal{H},\mathcal{C}$ include the requirements a schema should have to be compatible with the query `q`. The type of q, which is the type inferred by the outer inference rule, also satisfies the constraints $\mathcal{C}$.

8 Related Work and Conclusions

Fundamental to the work described in this report is the type system for ODMG OQL described in an earlier paper [3]. Alagić [2] independently gave a number of typing rules for OQL; see our earlier paper for a comparison. The canonical reference for work on type systems for database programming languages is the work of Buneman and Ohori [5]. The goals of their type inference algorithm are identical to ours; both approaches infer the most general type of an expression (if one exists) without accessing any schema information, and in this sense determine the constraints placed on the schema by the query. However, the underlying languages, the type systems, and some parts of the inference algorithms differ considerably. Buneman and Ohori introduce kinded types to infer the type of a record based on selections of fields on this record. Instead, we use the notion of *general* types; in this way we are able to express general type information not only for records, but also for parametric collection types, structures and classes. Moreover, due to the syntax of OQL, we define a wider variety of constraints than those introduced in their framework and therefore a different algorithm to resolve them.

The work most related to ours arises from studying type systems for object oriented programming languages, see for example [8, 9, 1]. However none of these studies consider the various issues arising from studying *database* type systems; for example, the complications arising from combining parametric collection types with subtyping.

Much research on schema evolution, schema inter-operation, distributed or semi-structured database applications has pointed out that there is a need to run queries in the presence of changing or heterogeneous schemata, or even in the absence of specific schema information. Our work addresses this problem, by proposing an inference algorithm for the ODMG query language OQL. This algorithm infers the most general type of an OQL query and derives the schema information required so that the query can be executed against it without any type errors. In contrast to other work, we deal with a rather complex type system, which includes atomic types, structures, classes, various (parameterised) collection types (set, bag, list, array) and function types. This, in connection with the rich semantics of OQL, results in the generation of a wide variety of constraints by the inference rules. We discuss the semantics of these constraints and provide a mechanism for their solution. Finally, we present a set of inference rules for OQL, which is the core of our type inference algorithm. Based on our experience, this algorithm, as well as all the formalisms prior to it, are easy to implement, and hence, we believe that they could prove to be useful in many applications.

Acknowledgements

Trigoni is funded by the State Scholarships Foundation of Greece and the National Bank of Greece.

References

1. O. Agesen and U. Holzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *OOPSLA*, pages 91–

107, 1995.
2. S. Alagić. Type checking OQL queries in the ODMG type systems. *ACM Transactions on Database Systems*, 24(3):319–360, September 1999.
3. G.M. Bierman and A. Trigoni. Towards a formal type system for ODMG OQL. Technical Report 497, University of Cambridge, Computer Laboratory, October 2000.
4. M.W. Bright, A.R. Hurson, and S. Pakzad. Automated resolution of semantic heterogeneity in multidatabases. *ACM Transactions on Database Systems*, 19(2), 1994.
5. P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.
6. R.G.G. Cattell et al. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
7. R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *PODS*, 1997.
8. J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA*, pages 146–161, 1991.
9. J. Plevyak and A.A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.
10. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
11. A. Trigoni. Phd thesis, to appear. 2001.