Formal semantics and analysis of object queries

G.M. Bierman
University of Cambridge Computer Laboratory
J.J. Thomson Avenue
Cambridge, CB3 0DF. UK.

gmb@cl.cam.ac.uk

ABSTRACT

Modern database systems provide not only powerful data models but also complex query languages supporting powerful features such as the ability to create new database objects and invocation of arbitrary methods (possibly written in a third-party programming language).

In this sense query languages have evolved into powerful programming languages. Surprisingly little work exists utilizing techniques from programming language research to specify and analyse these query languages. This paper provides a formal, high-level operational semantics for a complex-value OQL-like query language that can create fresh database objects, and invoke external methods. We define a type system for our query language and prove an important soundness property.

We define a simple effect typing discipline to delimit the computational effects within our queries. We prove that this effect system is correct and show how it can be used to detect cases of non-determinism and to define correct query optimizations.

1. INTRODUCTION

"Database languages ('query languages') are nothing but special-purpose *programming* languages." [6]

Since Codd's pioneering work, database systems have moved beyond the simple relational data model and basic query language. Modern data models typically support complex data types, objects which are collected into classes, and notions of subtyping. Likewise query languages have been extended to include various features including object identity, object creation, and method invocation. To this extent, we can see that Date's quote above is even more true now than when he made it nearly twenty years ago.

Given that query languages are now essentially complex programming languages, it is perhaps surprising to find that the techniques and methodologies of the programming languages community do not find widespread use in the specification and analysis of query languages. This paper applies two dominant themes in current programming language research—type systems and operational semantics—to the study of an object query language and the problems of query optimization.

We believe that a formal, mathematical approach is essential to set a firm foundation for researchers, users and implementors of complex query languages. Without such mathematical precision it is very difficult, for example, to assert correctness. For example, the ODMG [8, p.100] define a notion of *least upper bound* of two types in their object model, and give an informal definition. However a few moment's formality soon reveals that a *least* upper bound of two types need not necessarily exist (because we have both classes and interfaces)! We shall also see in a number of places that another advantage for our formal approach is that it allows us to consider the design space of various features.

In this paper we pay particular attention to object-oriented data models although our techniques apply equally well to both relational and object-relational data models. We define a simple object data model that is essentially a fragment of the ODMG object model. This model provides primitive types and arbitrarily nested collection types, along with classes, and supports single inheritance between classes.

We define a complex query language, broadly based on ODMG OQL. This query language supports path expressions, object creation, and (read-only) method invocation amongst the more familiar features. Our language is similar in spirit with IQL of Abiteboul and Kanellakis [1], although it is different in detail. Moreover the primary concern in their work is in the expressive power of IQL (in particular the implications of its ability to create new objects, see also [7]).

We specify formally the type system for queries, and also provide an operational semantics. This semantics defines precisely the process of evaluation of a query, and is defined recursively over the structure of the query. Given this operational semantics we are able to prove the correctness of our type system. As far as we are aware, no result of this form exists for object query languages (in fact, the opposite negative result has been asserted for ODMG OQL [2]).

We then turn our attention to reasoning about queries. In particular we are concerned with formalizing when the two queries should be considered equivalent, which is at the heart of the optimization problem. Even given our relatively small query language, we see that matters are subtle and complicated. The chief complication is that iteration over sets is non-deterministic—we can have no idea as to the order in which elements are taken from a set. In the pure relational world, this is not a problem as queries are purely declarative and so the order of evaluation is irrelevant. However once we add features familiar from object programming languages to our query language things are much more complicated. For example, consider the following query (written in a version of OQL).

```
SELECT (if size(Fs)<1
        then (new F(name:"Peter",pal:p)).name
        else p.name)
FROM p in Ps;
```

This query assumes a simple class `P` of objects with just a `name` attribute, whose extent is called `Ps`. We also have a class `F` with attributes `name` and `pal`, whose extent is called `Fs`. We shall assume initially there are no `F` objects and just two `P` objects, one with `name` "Jack", and the other "Jill".

Unfortunately this query is observably **non-deterministic**! The result of the query (and its side-effect on the extent of class `F`) is different depending on the order in which the `P` objects are considered.[1] If we visit the "Jack" object first, the result is the set `{"Peter","Jill"}`, otherwise the result is the set `{"Peter","Jack"}`.

Another problem, and one often ignored, e.g. [5, 19], is that method invocation may not terminate. For example, consider the following query which is a variant of the one above, where `P` objects now have a method `loop` that, as the name suggests, does not terminate.

```
SELECT (if size(Fs)<1 and p.name="Jack"
        then p.loop()
        else new F(name:"Peter",pal:p)
FROM p in Ps;
```

We now have quite different non-deterministic behaviour: the query terminates if we visit the "Jill" object first, but fails to terminate if we visit the "Jack" object first.

To address these problems (and also to formulate correct optimizations) we define an effect typing discipline for our query language. Such typing disciplines, originally proposed by Gifford and Lucassen [11], are used to delimit computational effects and have been used in a variety of programming languages. For example, Java contains a simple effects system where each method is labelled with the exceptions it might raise. We define a simple effects system that annotates types with details of the extents that may be used in the evaluation of the query. For example, the source of the non-determinism in the examples above is that the inner query both reads *and* updates the extent of the class `F`.

An advantage of our formal approach is that we can *prove* that our effects system is correct with respect to our operational semantics. (In fact the proof is a rather straightforward structural induction.) Given this result we are able to use our effects system to provably detect all cases of non-determinism.

We conclude this paper by briefly considering database support for method invocation. Using the formalism of operational semantics, we are able to explore and delineate the design space for method invocation support.

[1]This is clearly related to the update problem in SQL. However our solution is quite different. See §7 for further comparisons.

Contributions. In summary, the significant contributions of this paper include: (1) The definition of an OQL-like query language, IOQL, that incorporates comprehensions, object identifiers, path expressions, object creation, and simple method invocation; (2) The formal definition of a type system and operational semantics for IOQL; (3) A proof of type soundness for IOQL; (4) The development of a type-based effect system to infer database access and update behaviour of queries; (5) A proof of correctness for this analysis; and (6) The use of effects to detect non-determinism and define correct optimizations.

2. DATA MODEL

In this section we define our data model which is strongly influenced by ODMG ODL, although the techniques we employ could easily be applied to other complex-value data models, including object-relational data models.

Our data model is a class-based object model. As in ODL, we allow single-inheritance between classes, although for simplicity we have not included interfaces. All objects have a unique object identity (oid), and consist of internal state comprising attributes, and a collection of methods. For simplicity, we shall consider only *read-only* methods, similar to that provided by e.g. PREDATOR [22] or considered in [15]. The impact of more sophisticated method support is considered briefly in §5.

For example, here is a simple class definition of `Employee` objects.

```
class Employee extends Person
(extent Employees)
{ attribute int EmpID;
  attribute int GrossSalary;
  attribute Manager UniqueManager;
  int NetSalary (int TaxRate); }
```

This defines the `Employee` class as a subclass of `Person`. Its extent is called `Employees`. It has three attributes and one method. For simplicity we insist that all class definitions explicitly state a superclass (we also assume a class `Object`, which is the superclass of all classes). Two of the attributes, `EmpID` and `GrossSalary` are integer values. The third, `UniqueManager`, is an object-valued attribute. The method `NetSalary` takes an integer argument and returns an integer.

More formally we define the grammar for class definitions as follows, where ϕ denotes valid types in our data model, which are just class names and primitive types `int` and `bool`.

Class definition

$$\begin{array}{lcl} cd & ::= & \texttt{class}\,\texttt{C}_1\ \texttt{extends}\ \texttt{C}_2 \\ & & (\texttt{extent}\ e) \\ & & \{ad_1 \ldots ad_k \\ & & \ md_1 \ldots md_n\} \end{array}$$

Attribute definition

$$ad \ ::= \ \texttt{attribute}\ \phi\ a;$$

Method definition

$$md \ ::= \ \phi\ m\ (\phi_0\ x_0, \ldots, \phi_m\ x_m);$$

An **object schema** is then a collection of class definitions. Of course, not all collections of class definitions form a valid object schema (e.g. we shouldn't define the same class twice). It is quite straightforward to define formally the well-formedness conditions for a collection of class definitions, but we elide them from this short paper (they are similar, for example, to those for Java [16]).

Note 1. The types allowed in class definitions have been chosen so that they can represented *precisely* in our preferred method language, Java. As generic classes are not available in Java yet, we can not represent the generic collection type $\mathsf{set}(\sigma)$ precisely. Permitting types in the object model that can not be represented precisely in the method language leads to potential type insecurity [2].

3. IDEALIZED OBJECT QUERY LANGUAGE

In this section we define our query language. We first define the syntax for the language, and then define its type system, and finally define an operational semantics, which provides a high-level specification of the dynamics of query evaluation. We are then able to prove some properties of our typed query language.

3.1 Syntax

Our Idealized Object Query Language, IOQL, is essentially a core fragment of ODMG OQL. The main difference is that we use a comprehension syntax for bulk processing, rather than a select-from-where construct. (This is just for ease of presentation, our techniques can be applied to the whole of OQL without significant problems.) An IOQL **program** consists of a sequence of definitions, followed by a query. An IOQL definition is defined by the following grammar:

Query definition

$$def \quad ::= \quad \texttt{define}\, d(x_0 : \sigma_0, \ldots, x_n : \sigma_n)\ \texttt{as}\ q;$$

Such a definition associates a name with a parameterized query (thus definitions are non-recursive). We require that the types of the parameters are given (we do not provide type *inference* for definitions; this has been considered elsewhere for ODMG OQL [23]).

IOQL queries are defined by the following grammar:

Query expression

q	$::=$	i	integer
	$\mid$	$\texttt{true} \mid \texttt{false}$	booleans
	$\mid$	x	identifier
	$\mid$	$\{q_0, \ldots, q_k\}$	set
	$\mid$	$q_1\ \texttt{sop}\ q_2$	set ops
	$\mid$	$q_1\ \texttt{iop}\ q_2$	int ops
	$\mid$	$q_1 \texttt{=} q_2$	int equality
	$\mid$	$q_1 \texttt{==} q_2$	object equality
	$\mid$	$\langle l_1 : q_1, \ldots, l_k : q_k \rangle$	record
	$\mid$	$q.l$	record access
	$\mid$	$d(q_0, \ldots, q_k)$	definition access
	$\mid$	$\texttt{size}(q)$	set size
	$\mid$	$(\texttt{C})q$	cast
	$\mid$	$q.a$	attribute access
	$\mid$	$q.m(q_0, \ldots, q_k)$	method invocation
	$\mid$	$\texttt{new}\ \texttt{C}(a_0 : q_0, \ldots, a_k : q_k)$	object creation
	$\mid$	$\texttt{if}\ q_1\ \texttt{then}\ q_2\ \texttt{else}\ q_3$	conditional
	$\mid$	$\{q \mid cq_0, \ldots, cq_k\}$	comprehension

Comprehension qualifier

cq	$::=$	q	predicate
	$\mid$	$x \leftarrow q$	generator

We assume a countable set of identifiers, and assume for convenience a number of designated subsets: record labels, l, object attributes, a, definition identifiers, d, and extent identifiers, e, and by convention these are never mixed up, nor used as variables in generators, nor as parameters in definitions.

To simplify our presentation we have only provided one collection type, set, although we could have easily added others (bags, lists, etc.), or even moved to a monoid setting as proposed in [10] (although, as is noted, the monoid setting can not model the whole of ODMG OQL, which is one of our longer-term aims). However IOQL provides a reasonably rich collection of types, including booleans, integers, classes, records as well as sets. (The type system is presented formally in the next section.) We assume a collection, `sop`, of set operators, and a collection, `iop`, of integer operations, although to save space in what follows we shall consider only one candidate from each set, namely set union ($\cup$) and integer addition ($+$).

IOQL queries have a number of object-oriented features. We can create new objects using the `new` expression. This takes a number of arguments which are pairs of attribute name and value—we insist (unlike the ODMG) that *all* attributes are defined, and our type system will ensure that they are correct. The expression returns a fresh oid, and we assume that this object is included immediately in its class extent. The size of a set can be checked using the `size` expression. Given a query that denotes an object, we can access an attribute, or invoke a method, using the familiar 'dot' notation. As we permit object-valued attributes, we can thus form so-called path expressions, e.g. `x.foo.bar`. Again our type system ensures correctness. We can cast a query that denotes an object to its superclass. This is written $(\texttt{C})q$, and the type system will ensure that the query q denotes an object of type $\texttt{C}'$ where $\texttt{C}'$ is a subclass of $\texttt{C}$. Incorrect casting is detected by the type system.

Note 2. The ODMG [8, §4.8.2] suggest that OQL, like Java, also permits downcasting, i.e. casting to a *sub*class. However this is an inherently unsafe operation, and leads to an *insecure* type system where, e.g. we can invoke methods that are not supported by an object. This is handled in Java by the exception mechanism. It is unclear that exceptions are desirable in a general query language.

3.2 Type system

IOQL supports a complex type system, where types are given by the following grammar:

IOQL type

$$\sigma \quad ::= \quad \phi \mid \mathsf{set}(\sigma) \mid \langle l_1 : \sigma_1, \ldots, l_k : \sigma_k \rangle$$

IOQL supports the types of the underlying data model (bool, int and class types) as well as a single collection type, set, and record types (which is the same as OQL's struct type except that we do not specify a name for the type). Thus a conventional table can be represented as a set of records.

The class definitions in an object schema specify a subclass hierarchy. When typing IOQL programs we assume this subclass information in the form of a relation, which is written **extends**. (For example, given the class definition in §1 we would have `Employee` **extends** `Person`.)

Given this subclass relation, we can define a *subtyping* relation on IOQL types, written $\sigma \leq \sigma'$. This is given by the following rules:

$$\frac{\texttt{C}\ \textbf{extends}\ \texttt{C}'}{\texttt{C} \leq \texttt{C}'} \qquad \frac{}{\sigma \leq \sigma} \qquad \frac{\sigma \leq \sigma' \quad \sigma' \leq \sigma''}{\sigma \leq \sigma''}$$

$$\frac{\sigma_1 \leq \sigma_1' \ldots \sigma_k \leq \sigma_k'}{\langle l_1 : \sigma_1, \ldots l_k : \sigma_k \rangle \leq \langle l_1 : \sigma_1', \ldots l_k : \sigma_k' \rangle}$$

Note 3. We have given a moderately straightforward definition of subtyping, matching that suggested by the ODMG. It could easily be extended with other interesting features, for example to allow width subtyping between records. Again our formal specification makes it easy to investigate such alternatives.

In the rest of this subsection we provide typing judgements for IOQL programs. Typing judgements are logical statements that are written $\Gamma \vdash X : Y$, which is intended to mean that given assumptions Γ, X has type Y. We will present rules for forming valid judgements about IOQL programs, definitions and queries, and to aid readability we annotate the turnstile accordingly. The form of the typing environment Γ varies depending on the type system being defined. One part of the typing environment is defined by the object schema: we write E to denote a partial function from extent names to their class (thus given the example in §1, E will be the function $\{\texttt{Employees} \mapsto \texttt{Employee}\}$).

First let us consider IOQL definitions. Definitions are clearly functions, and we write function types thus: $\vec{\sigma} \to \sigma'$, where $\vec{\sigma}$ is a sequence of types. A definition typing judgement is written $E; D \vdash_{\text{def}} \mathit{def} : \vec{\sigma} \to \sigma'$, where D is a partial function that maps the previously defined definition identifiers to their types. The rule for forming valid definition typing judgements is as follows:

$$\frac{E; D; x_0 : \sigma_0, \ldots, x_k : \sigma_k \vdash_{\text{ioql}} q : \sigma'}{E; D \vdash_{\text{def}} \texttt{define}\ f(x_0 : \sigma_0, \ldots, x_k : \sigma_k)\ \texttt{as}\ q : \sigma_0, \ldots, \sigma_k \to \sigma'}$$

A query typing judgement is written $E; D; Q \vdash_{\text{ioql}} q : \sigma$, where E and D are as before, and Q is a partial function that maps the free identifiers of the query q to their types.

The rules for forming valid query typing judgements are given in Figure 1. For some of the rules we make use of auxiliary functions, *atype*, *atypes* and *mtype*. These functions are generated by the object schema. The first takes a class name and an attribute and returns that attribute's type, the second takes a class name and returns a list of all its attributes with their type, and the third takes a class name and a method, and returns the type of that method (which is also a function type).[2]

Given these two systems, we can finally define the rule for forming a valid program typing judgement, which is written $E \vdash_{\text{prog}} \mathit{def}_0 \ldots \mathit{def}_k\ q : \sigma$. (We have used a little shorthand here, where the function types are abbreviated as τ_i and def_i has been expanded to $\texttt{define}\ f_i(\vec{x_i})\ \texttt{as}\ q_i$.)

[2]The *mtype* function is actually slightly more complicated in that it has to handle method inheritance and overriding. We elide the details in this short paper.

$$\frac{\begin{array}{l} E; \emptyset \vdash_{\text{def}} \texttt{define}\ f_0(\vec{x_0})\ \texttt{as}\ q_0 : \tau_0 \\ E; f_0 : \tau_0 \vdash_{\text{def}} \texttt{define}\ f_1(\vec{x_1})\ \texttt{as}\ q_1 : \tau_1 \\ \ldots \\ E; f_0 : \tau_0, \ldots, f_{k-1} : \tau_{k-1} \vdash_{\text{def}} \texttt{define}\ f_k(\vec{x_k})\ \texttt{as}\ q_k : \tau_k \\ E; f_0 : \tau_0, \ldots, f_k : \tau_k; \emptyset \vdash_{\text{ioql}} q : \sigma \end{array}}{E \vdash_{\text{prog}} \mathit{def}_0 \ldots \mathit{def}_k\ q : \sigma}$$

Note 4. Our type system attempts to reflect the informal intentions of the ODMG OQL type system. Others, e.g. [21], have suggested quite different type systems. Whilst interesting these type systems are radically different from those existing in programming languages, and it is unclear how they will interact with arbitrary method calls. For example, our type system combines well with the type system of Java and other object-oriented programming languages.

3.3 Operational Semantics

In this section we give a precise, mathematical specification of the dynamic behaviour of IOQL queries.

There are a number of ways of presenting the dynamics of queries. We shall use structural operational semantics, or just operational semantics. This is a popular techniques in the programming language community, and has been successfully used to specify, for example, the entirety of SML [18] and significant portions of MSIL, Microsoft's bytecode language for .NET [13].

One presentation of an operational semantics is based on normalization ("big-step"), but we shall follow the approach of [25] and use an operational semantics based on reduction ("single-step"). This has the advantage of both making proofs simpler, and as we shall see makes the nondeterministic nature of query evaluation explicit.

First we define a subset of queries that are values, i.e. expressions for which no further reduction is possible. If a query terminates, then it will return a value. Values are defined by the following grammar. (Note this requires the addition of oids to the grammar of queries. For simplicity we'll assume that the set of oids is a designated subset of the program identifiers. Thus their types will be contained in the typing environment Q.)

Query value

v	::=	i	integer
	|	$\texttt{true} \mid \texttt{false}$	booleans
	|	$\{v_0, \ldots, v_k\}$	set
	|	$\langle l_1 : v_1, \ldots, l_k : v_k \rangle$	record
	|	o	oid

Queries are evaluated given a Definition Environment (*DE*), Extent Environment (*EE*) and an Object Environment (*OE*) (this is essentially the heart of the database!). *DE* is a function that maps definition identifiers to the definitions themselves. We use λ-notation to represent definitions; thus a definition $\texttt{define}\ f(x : \sigma)\ \texttt{as}\ q$ is represented as $\lambda x : \sigma . q$ (we will often drop the types of the bound variables to aid presentation). The extent environment maps an extent identifier to a pair containing the class name and a set of oids which are the current objects in that extent. The object environment, *OE*, is a function that maps an oid to the runtime representation of that object. This representation contains the class of the object and the values of its attributes, and is written e.g. $\ll \texttt{C}, a_1 : v_1, \ldots, a_k : v_k \gg$.

$$\frac{}{E; D; Q \vdash_{\text{ioql}} i\colon \mathsf{int}}\,(\text{Int}) \qquad \frac{}{E; D; Q \vdash_{\text{ioql}} \mathtt{true}\colon \mathsf{bool}}\,(\text{Bool}) \qquad \frac{}{E; D; Q \vdash_{\text{ioql}} \mathtt{false}\colon \mathsf{bool}}\,(\text{Bool})$$

$$\frac{}{E; D; Q, x\colon \sigma \vdash_{\text{ioql}} x\colon \sigma}\,(\text{Id}) \qquad \frac{}{E, e\colon \mathtt{C}; D; Q \vdash_{\text{ioql}} e\colon \mathsf{set}(\mathtt{C})}\,(\text{Extent})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} e\colon \mathsf{set}(\sigma)}{E; D; Q \vdash_{\text{ioql}} \mathtt{size}(e)\colon \mathsf{int}}\,(\text{Size})$$

$$\frac{E; D, d\colon \sigma'_0, \ldots, \sigma'_k \to \sigma; Q \vdash_{\text{ioql}} q_0\colon \sigma_0 \quad \cdots \quad E; D, d\colon \sigma'_0, \ldots, \sigma'_k \to \sigma; Q \vdash_{\text{ioql}} q_k\colon \sigma_k \qquad \forall i \in 0..k.\sigma_i \leq \sigma'_i}{E; D, d\colon \sigma'_0, \ldots, \sigma'_k \to \sigma; Q \vdash_{\text{ioql}} d(q_0, \ldots, q_k)\colon \sigma}\,(\text{Def})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_0\colon \sigma_0 \quad \cdots \quad E; D; Q \vdash_{\text{ioql}} q_k\colon \sigma_k \quad \forall i \in 0..k.\sigma_i \leq \sigma}{E; D; Q \vdash_{\text{ioql}} \{q_0, \ldots, q_k\}\colon \mathsf{set}(\sigma)}\,(\text{Set})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \mathsf{set}(\sigma) \quad E; D; Q \vdash_{\text{ioql}} q_2\colon \mathsf{set}(\sigma)}{E; D; Q \vdash_{\text{ioql}} q_1 \cup q_2\colon \mathsf{set}(\sigma)}\,(\text{Union})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \mathsf{int} \quad E; D; Q \vdash_{\text{ioql}} q_2\colon \mathsf{int}}{E; D; Q \vdash_{\text{ioql}} q_1 + q_2\colon \mathsf{int}}\,(\text{Add})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \mathsf{int} \quad E; D; Q \vdash_{\text{ioql}} q_2\colon \mathsf{int}}{E; D; Q \vdash_{\text{ioql}} q_1 \mathtt{=} q_2\colon \mathsf{bool}}\,(\text{Int eq}) \qquad \frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \mathtt{C} \quad E; D; Q \vdash_{\text{ioql}} q_2\colon \mathtt{C}'}{E; D; Q \vdash_{\text{ioql}} q_1 \mathtt{==} q_2\colon \mathsf{bool}}\,(\text{Obj eq})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \sigma_1 \quad \cdots \quad E; D; Q \vdash_{\text{ioql}} q_k\colon \sigma_k}{E; D; Q \vdash_{\text{ioql}} \langle l_1\colon q_1, \ldots, l_k\colon q_k\rangle\colon \langle l_1\colon \sigma_1, \ldots, l_k\colon \sigma_k\rangle}\,(\text{Rec}) \qquad \frac{E; D; Q \vdash_{\text{ioql}} q\colon \langle l_1\colon \sigma_1, \ldots, l_k\colon \sigma_k\rangle \quad i \in 1..k}{E; D; Q \vdash_{\text{ioql}} q.l_i\colon \sigma_i}\,(\text{Rec access})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_1\colon \mathsf{bool} \quad E; D; Q \vdash_{\text{ioql}} q_2\colon \sigma_2 \quad E; D; Q \vdash_{\text{ioql}} q_3\colon \sigma_3 \qquad \sigma_2 \leq \sigma \text{ and } \sigma_3 \leq \sigma}{E; D; Q \vdash_{\text{ioql}} \mathtt{if}\ q_1\ \mathtt{then}\ q_2\ \mathtt{else}\ q_3\colon \sigma}\,(\text{Cond})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q\colon \mathtt{C} \quad \mathit{atype}(\mathtt{C}, a) = \phi}{E; D; Q \vdash_{\text{ioql}} q.a\colon \phi}\,(\text{Attribute}) \qquad \frac{E; D; Q \vdash_{\text{ioql}} q\colon \mathtt{C} \quad \mathtt{C} \leq \mathtt{C}'}{E; D; Q \vdash_{\text{ioql}} (\mathtt{C}')q\colon \mathtt{C}'}\,(\text{Upcast})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_0\colon \phi_0 \quad \cdots \quad E; D; Q \vdash_{\text{ioql}} q_k\colon \phi_k \quad \mathit{atypes}(\mathtt{C}) = [a_0\colon \phi'_0, \ldots, a_k\colon \phi'_k] \quad \forall i \in 0..k.\phi_i \leq \phi'_i}{E; D; Q \vdash_{\text{ioql}} \mathtt{new}\,\mathtt{C}(a_0\colon q_0, \ldots, a_k\colon q_k)\colon \mathtt{C}}\,(\text{New})$$

$$\frac{\begin{array}{l} E; D; Q \vdash_{\text{ioql}} q\colon \mathtt{C} \\ E; D; Q \vdash_{\text{ioql}} q_0\colon \phi_0 \quad \cdots \quad E; D; Q \vdash_{\text{ioql}} q_k\colon \phi_k \quad \mathit{mtype}(\mathtt{C}, m) = \phi'_0, \ldots, \phi'_k \to \phi \quad \forall i \in 0..k.\phi_i \leq \phi'_i \end{array}}{E; D; Q \vdash_{\text{ioql}} q.m(q_0, \ldots, q_k)\colon \phi}\,(\text{Method})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q\colon \sigma}{E; D; Q \vdash_{\text{ioql}} \{q \mid \}\colon \mathsf{set}(\sigma)}\,(\text{Comp1}) \qquad \frac{E; D; Q \vdash_{\text{ioql}} q_2\colon \mathsf{set}(\sigma) \quad E; D; Q, x\colon \sigma \vdash_{\text{ioql}} \{q_1 \mid \overrightarrow{cq}\}\colon \sigma'}{E; D; Q \vdash_{\text{ioql}} \{q_1 \mid x \leftarrow q_2, \overrightarrow{cq}\}\colon \sigma'}\,(\text{Comp2})$$

$$\frac{E; D; Q \vdash_{\text{ioql}} q_2\colon \mathsf{bool} \quad E; D; Q \vdash_{\text{ioql}} \{q_1 \mid \overrightarrow{cq}\}\colon \sigma'}{E; D; Q \vdash_{\text{ioql}} \{q_1 \mid q_2, \overrightarrow{cq}\}\colon \sigma'}\,(\text{Comp3})$$

Figure 1: Type system for IOQL queries

A single reduction step is written $DE \vdash EE, OE, q \rightarrow EE', OE', q'$, where EE, OE and q are the extent environment, object environment and query before the step, and EE', OE' and q' their values after. (For convenience we abbreviate a number of the steps $DE \vdash EE, OE, q \rightarrow q'$, which is shorthand for $DE \vdash EE, OE, q \rightarrow EE, OE, q'$.)

The operational semantics of IOQL are defined in Figure 2. An evaluation context, $\mathcal{E}$, is a query with a single hole, written $\bullet$, in place of the next subexpression to be evaluated. The result of placing a query q in that hole is written $\mathcal{E}[q]$. Evaluation contexts are defined so that they specify the *order* of evaluation, e.g. unions are evaluated left-to-right, and arguments to definitions/methods are evaluated left-to-right and are call-by-value. The fundamental property of evaluation contexts is that any given query is either a value, or can be *uniquely* decomposed into a evaluation context and a subexpression that will match a reduction step. We write $q[x := v]$ for the substitution of value v for all free instances of identifier x in query q.

Along with the single reduction steps, there is a rule, (Context), which forms the contextual closure of reduction with respect to evaluation contexts. We define $\twoheadrightarrow$ to be the reflexive, transitive closure of the $\rightarrow$ relation.

Let us look at a couple of the reduction steps in detail. The (New) rule generates a fresh oid, which is the result of the reduction, along with appropriately updated extent and object environments. The (Method) rule makes use of an *mbody* function which returns the code for the method—as with definitions, we represent methods using λ-notation. We have assumed an auxiliary relation $\Downarrow$ which is used to denote the (deterministic) evaluation of the method body: it relates method code along with the object environment to a final value (e.g. [20] gives a similar such relation for Java). The exact definition of this relation is not important for now; what is important is that the method body does not update the object environment—methods are *read-only*.

The other rule of interest is (ND comp): this explicitly embodies the non-deterministic nature of query evaluation. An element is picked at random from the generator set. As evaluation of a union expression is explicitly left-to-right (by the definition of evaluation contexts), this means that this element really is used first.

Note 5. We have explicitly made IOQL a call-by-value language. The ODMG is vague on this issue. Fegaras and Maier [10] assume a call-by-name semantics (their optimizations are *invalid* for a call-by-value semantics) but note that this is inefficient and suggest a call-by-need implementation, although no analysis is made of the runtime implications of this suggestion. Moreover as most programming languages are call-by-value, it is not clear how this suggestion would interact with method invocation.

3.4 Properties

In this section we consider some properties of our type system and operational semantics of IOQL. In particular we prove the type system correct—as we have noted before, it has been claimed that this property fails for ODMG OQL [2].

Our principal result is a type soundness property or, more informally, that well-typed queries "do not go wrong". As demonstrated by Wright and Felleisen [25], the essence of this property is contained in two important theorems: the subject reduction theorem and the progress theorem.

We shall only consider *closed* queries, that is queries whose only free variables are definition or extent identifiers, or oids. We need to relate the runtime environments, DE, EE and OE with the typing environments E, D and Q. We shall elide the details here, as they are straightforward but rather lengthy to state formally. We shall write $E, D, Q \vdash EE, DE, OE, q \colon \sigma$ to mean that the query, runtime and typing environments correspond correctly.

We can now prove the subject reduction theorem, i.e. reductions preserve types (up to subtyping).

THEOREM 1 (SUBJECT REDUCTION). *If $E, D, Q \vdash EE, DE, OE, q \colon \sigma$ and $DE \vdash EE, OE, q \rightarrow EE', OE', q'$ then $\exists Q' \supseteq Q. E, D, Q' \vdash EE', DE, OE', q' \colon \sigma'$ where $\sigma' \leq \sigma$.*

The proof is by case analysis on the reduction step, where we make frequent use of the following lemma (which is proved by induction on the first typing derivation).

LEMMA 1 (SUBSTITUTION). *If $E; D; Q, x \colon \sigma \vdash_{ioql} q \colon \tau$ and $E; D; Q \vdash_{ioql} v \colon \sigma'$ for some $\sigma' \leq \sigma$ then $\exists \tau'$ such that $E; D; Q \vdash_{ioql} q[x := v] \colon \tau'$ and $\tau' \leq \tau$.*

Next we show that well-typed expressions that are not values can always make a reduction step.

THEOREM 2 (PROGRESS). *If $E, D, Q \vdash EE, DE, OE, q \colon \sigma$ then either q is a value or $\exists EE', OE', q'$ such that $DE \vdash EE, OE, q \rightarrow EE', OE', q'$.*

The proof is by induction on the typing derivation of the query. From these two theorems, it is quite easy to deduce a type soundness theorem for IOQL, which essentially states that well-typed queries "do not go wrong", i.e. a query can never get into a state where it is not a value and can not reduce further.

THEOREM 3 (TYPE SOUNDNESS OF IOQL). *If $E, D, Q \vdash EE, DE, OE, q \colon \sigma$ then it is never the case that $DE \vdash EE, OE, q \twoheadrightarrow EE', OE', q'$ such that q' is not a value and $\neg \exists q'', EE'', OE''. DE \vdash EE', OE', q \rightarrow EE'', OE'', q''$.*

Let us call an IOQL query *functional* if it contains no instance of the `new` expression (and all of the definitions it invokes are functional). It should be clear from the operational semantics that any potentiality for observable non-determinism has been removed.

THEOREM 4 (FUNCTIONAL QUERIES). *If $E, D, Q \vdash DE, EE, OE, q \colon \sigma$ and q is functional, then $\forall EE', EE'', OE', OE'', v', v''$ such that $DE \vdash EE, OE, q \twoheadrightarrow EE', OE', v'$ and $DE \vdash EE, OE, q \twoheadrightarrow EE'', OE'', v''$ it is the case that $EE' = EE''$, $OE' = OE''$ and $v = v'$.*

4. EFFECT ANALYSIS

IOQL queries are permitted to side-effect the database: they can create fresh objects, which are then stored in the extent. In §1 we saw that this can introduce non-determinism in query evaluation. However side-effects also impact on query optimization. Assume a database with just one `Person` object (with say name "Jack" and address "Utah") and one `Employee` object (with say name "Jill" and address "NYC"), where `Employee` is a subclass of `Person`. Consider the following query.

Evaluation context

$$\begin{array}{lll} \mathcal{E} & ::= & \bullet \mid \{\vec{v}, \mathcal{E}, \vec{q}\} \mid \mathcal{E} + q \mid v + \mathcal{E} \mid \mathcal{E} \cup q \mid v \cup \mathcal{E} \mid \mathcal{E}\texttt{=}q \mid v\texttt{=}\mathcal{E} \mid \mathcal{E}\texttt{==}q \mid v\texttt{==}\mathcal{E} \mid \langle \vec{l \colon v}, \mathcal{E}, \vec{l \colon q} \rangle \mid \mathcal{E}.l \mid d(\vec{v}, \mathcal{E}, \vec{q}) \mid \texttt{size}(\mathcal{E}) \\ & \mid & (\texttt{C})\mathcal{E} \mid \mathcal{E}.a \mid \mathcal{E}.m(\vec{q}) \mid v.m(\vec{v}, \mathcal{E}, \vec{q}) \mid \texttt{new C}(\vec{a \colon v}, \mathcal{E}, \vec{a \colon q}) \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ q_1\ \texttt{else}\ q_2 \mid \{q \mid \mathcal{E}, \vec{cq}\} \mid \{q \mid x \leftarrow \mathcal{E}, \vec{cq}\} \mid \{\mathcal{E} \mid\ \} \end{array}$$

Reduction steps

(Definition)	$DE \vdash EE, OE, d(\vec{v}) \rightarrow q[\vec{x} := \vec{v}]$	where $DE(d) = \lambda\vec{x}.q$
(Extent)	$DE \vdash EE, OE, e \rightarrow v$	where $EE(e) = (\texttt{C}, v)$
(Size)	$DE \vdash EE, OE, \texttt{size}(\{v_0, \ldots, v_k\}) \rightarrow k$	
(Union)	$DE \vdash EE, OE, v_1 \cup v_2 \rightarrow v_3$	where $v_3 = v_1 \cup v_2$
(Addition)	$DE \vdash EE, OE, i_1 + i_2 \rightarrow i_3$	where $i_3 = i_1 + i_2$
(Int eq)	$DE \vdash EE, OE, i_1\texttt{=}i_2 \rightarrow b$	where $b \overset{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } i_1 = i_2 \\ \texttt{false} & \text{otherwise} \end{cases}$
(Object eq)	$DE \vdash EE, OE, o_1\texttt{==}o_2 \rightarrow b$	where $b \overset{\text{def}}{=} \begin{cases} \texttt{true} & \text{if } o_1 = o_2 \\ \texttt{false} & \text{otherwise} \end{cases}$
(Cond1)	$DE \vdash EE, OE, \texttt{if true then}\ q_1\ \texttt{else}\ q_2 \rightarrow q_1$	
(Cond 2)	$DE \vdash EE, OE, \texttt{if false then}\ q_1\ \texttt{else}\ q_2 \rightarrow q_2$	
(Record)	$DE \vdash EE, OE, \langle l_1 \colon v_1, \ldots, l_k \colon v_k \rangle.l_i \rightarrow v_i$	$1 \leq i \leq k$
(Attribute)	$DE \vdash EE, OE, o.a_i \rightarrow v_i$	where $OE(o) = \ll \texttt{C}, a_1 \colon v_1, \ldots, a_k \colon v_k \gg$ and $1 \leq i \leq k$
(Upcast)	$DE \vdash EE, OE, (\texttt{C}')o \rightarrow o$	where $OE(o) = \ll \texttt{C}, \ldots \gg$ and $\texttt{C} \leq \texttt{C}'$
(New)	$DE \vdash EE, OE, \texttt{new C}(a_0 \colon v_0, \ldots, a_k \colon v_k) \rightarrow EE', OE', o$	where fresh $o \notin \text{dom}(OE)$ and $OE' \overset{\text{def}}{=} OE[o \mapsto \ll \texttt{C}, a_0 \colon v_0, \ldots, a_k \colon v_k \gg]$ and $EE(e) = (\texttt{C}, v)$ and $EE' \overset{\text{def}}{=} EE[e \mapsto (\texttt{C}, v \cup \{o\})]$
(Method)	$DE \vdash EE, OE, o.m(\vec{v}) \rightarrow v$	where $OE(o) = \ll \texttt{C}, \ldots \gg$ and $\mathit{mbody}(\texttt{C}, m) = \lambda\vec{x}.\mathit{body}$ and $OE, \mathit{body}[\vec{x} := \vec{v}, \texttt{this} := o] \Downarrow v$
(Empty comp)	$DE \vdash EE, OE, \{v \mid\ \} \rightarrow \{v\}$	
(True comp)	$DE \vdash EE, OE, \{q \mid \texttt{true}, \vec{cq}\} \rightarrow \{q \mid \vec{cq}\}$	
(False comp)	$DE \vdash EE, OE, \{q \mid \texttt{false}, \vec{cq}\} \rightarrow \{\ \}$	
(Triv comp)	$DE \vdash EE, OE, \{q \mid x \leftarrow \{\ \}, \vec{cq}\} \rightarrow \{\ \}$	
(ND comp)	$DE \vdash EE, OE, \{q \mid x \leftarrow \{v_1, \ldots, v_k\}, \vec{cq}\} \rightarrow (\{q \mid \vec{cq}\}[x := v_i]) \cup \{q \mid x \leftarrow v_{\text{rest}}, \vec{cq}\}$	for some $i \in 1..k$ and $v_{\text{rest}} \overset{\text{def}}{=} \{v_1, \ldots, v_k\} - v_i$

$$\frac{DE \vdash EE, OE, q \rightarrow EE', OE', q'}{DE \vdash EE, OE, \mathcal{E}[q] \rightarrow EE', OE', \mathcal{E}[q']}\ \text{(Context)}$$

Figure 2: Operational semantics for IOQL

```
select new Person(name:e.name,address:"Utah")
       from   e in Employees
intersect
select p from p in Persons
where  p.address="Jill" and p.name="Utah";
```

There is no non-determinism in this query: the only result possible is the singleton set containing the "Jill", "Utah" object. However a common query optimization is to *commute* the order of set intersections. Unfortunately this "optimization" would result in a query that returns a different result: the empty set!

Again the problem is the side-effecting behaviour of the query. To address these problems we shall define an *effects system*, which is an adjunct to the type system and essentially delimits the computational effects of a query. Effects systems have been used in a variety of programming languages, e.g. [14].

We define a new typing relation $E; D; Q \vdash_{\text{ioql}} q \colon \sigma \mathrel{!} \epsilon$ to mean that not only does q yield a value of type σ (or some subtype thereof), but that it has *effects* delimited by ϵ. In our system we shall be interested in delimiting two computational effects: $\mathsf{R(C)}$ is the effect that the extent of class $\mathtt{C}$ has been read, and $\mathsf{A(C)}$ is the effect that the extent of class $\mathtt{C}$ has been added to (by the creation of a new $\mathtt{C}$ object).

Formally an effect is either the empty effect, written $\emptyset$, the union of two effects, or the $\mathsf{R(C)}$ or $\mathsf{A(C)}$ effect. Equality of effects is modulo the assumption that $\cup$ is associative, commutative, idempotent, and has $\emptyset$ as a unit. As a shorthand we sometimes treat effects as sets and write e.g. $\mathsf{R(C)} \in \epsilon$. A subeffecting relation, $\subseteq$, is naturally induced: $\epsilon \subseteq \epsilon' \stackrel{\text{def}}{=} \exists \epsilon''.\epsilon' = \epsilon \cup \epsilon''$.

Types are as before except that the function types used to represent definitions now come labelled with the effect that occurs when that definition is used. For example, $\mathsf{int} \xrightarrow{\mathsf{R(C)}} \mathsf{int}$ denotes a definition which takes an integer argument, and returns an integer value and has the side effect that should it terminate it may read the extent of class $\mathtt{C}$.

The effects system for IOQL is given in Figure 3. Let us consider a couple of these rules in detail. The (Extent) rule generates the Read effect, and the (New) generates the Add effect. In the (Method) rule we assume that methods have also been typed using an effects systems, and that the method's effect, ϵ'', is included in the overall effect of the method. Of course, we have assumed that methods both can not read the extents and can not side-effect the database, so the value of ϵ'' will always be $\emptyset$. (If we allow more sophisticated methods, then this may not necessarily be true, see §5.)

Finally we have included an extra rule, (Does). This provides a notion of subeffect, that is given a derivation that a query has an effect ϵ we can safely conclude that it has a less precise effect ϵ'. Two properties of the effects system are immediate:

Lemma 2. *1. For all values v, $E; D; Q \vdash_{ioql} v \colon \sigma \mathrel{!} \emptyset$.*

2. If $E; D; Q, x \colon \sigma \vdash_{ioql} q \colon \tau \mathrel{!} \epsilon$ and $E; D; Q \vdash_{ioql} v \colon \sigma' \mathrel{!} \emptyset$ where $\sigma' \leq \sigma$ then $\exists \tau', \epsilon'$ such that $E; D; Q \vdash_{ioql} q[x := v] \colon \tau' \mathrel{!} \epsilon$ and $\tau' \leq \tau$, $\epsilon' \subseteq \epsilon$.

We can *instrument* the IOQL operational semantics so that it traces the effects of query evaluation. A reduction step is now written $DE \vdash EE, OE, q \xrightarrow{\epsilon} EE', OE', q'$ where ϵ is the effect. The instrumented semantics for IOQL is given in Figure 4. (The evaluation contexts are as in Figure 2.)

Again we define $\twoheadrightarrow$ to be the reflexive, transitive closure of the instrumented reduction relation. For example, transitivity is given by the following rule.

$$\frac{\begin{array}{l} DE \vdash EE, OE, q \overset{\epsilon_1}{\twoheadrightarrow} EE', OE', q' \\ DE \vdash EE', OE', q' \overset{\epsilon_2}{\twoheadrightarrow} EE'', OE'', q'' \end{array}}{DE \vdash EE, OE, q \overset{\epsilon_1 \cup \epsilon_2}{\twoheadrightarrow} EE'', OE'', q''} \text{(Transitivity)}$$

The main result in this section is correctness of our effects system. As before we need to relate the typing and runtime environments, which we write $E, D, Q \vdash EE, DE, OE, q \colon \sigma ! \epsilon$. As in §3.4, correctness is given by two theorems. First that our instrumented semantics both preserves types and also is consistent with the effects.

Theorem 5 (Subject reduction). *If $E, D, Q \vdash EE, DE, OE, q \colon \sigma \mathrel{!} \epsilon$ and $DE \vdash EE, OE, q \xrightarrow{\epsilon'} EE', OE', q'$ then $\exists Q' \supseteq Q. E, D, Q' \vdash EE', DE, OE', q' \colon \sigma' ! \epsilon$ where $\sigma' \leq \sigma$ and $\epsilon' \subseteq \epsilon$.*

(Note that the inferred effect may be less precise than actual runtime effect.)

Theorem 6 (Progress). *If $E, D, Q \vdash EE, DE, OE, q \colon \sigma \mathrel{!} \epsilon$ then either q is a value or $\exists EE', OE', q', \epsilon'$ such that $DE \vdash EE, OE, q \xrightarrow{\epsilon'} EE', OE', q'$.*

We can now consider how to use the effects information. We saw in §1 that object creation can result in non-determinism. Let us consider this in detail, with the following example

$$\{q \mid x \leftarrow \{v_1, \ldots, v_k\}, q'\}$$

In essence this query reduces to an arbitrarily ordered union of the k subqueries: $\{q|q'\}[x := v_1]$, ..., $\{q|q'\}[x := v_k]$. Non-determinism will occur if there is *interference* between these subqueries. Interference occurs if there is both a read *and* an add to a common extent. If there is no interference, then the ordering of the subqueries clearly does not matter, and the query will be deterministic (up to a possible bijection on the oids).

We can formalise this idea by considering a new type system, written $\vdash'$, which is identical to that given in Figure 3 except that the (Comp2) rule is replaced by the following

$$\frac{\begin{array}{l} E; D; Q \vdash'_{\text{ioql}} q_2 \colon \mathsf{set}(\sigma) \mathrel{!} \epsilon_2 \\ E; D; Q, x \colon \sigma \vdash'_{\text{ioql}} \{q_1 \mid \overrightarrow{cq}\} \colon \sigma' \mathrel{!} \epsilon_1 \quad \mathit{nonint}(\epsilon_1) \end{array}}{E; D; Q \vdash'_{\text{ioql}} \{q_1 \mid x \leftarrow q_2, \overrightarrow{cq}\} \colon \sigma' \mathrel{!} \epsilon_1 \cup \epsilon_2} \text{(Comp2)}$$

where the non-interference predicate is defined as follows.

$$\mathit{nonint}(\epsilon) \stackrel{\text{def}}{=} \forall \mathsf{R(C)} \in \epsilon. \neg \exists \mathsf{A(C)} \in \epsilon$$

Theorem 7 (Determinism). *If $E, D, Q \vdash' EE, DE, OE, q \colon \sigma \mathrel{!} \epsilon$ then $\forall v, v''$ such that $DE \vdash EE, OE, q \twoheadrightarrow EE', OE', v$ and $DE \vdash EE, OE, q \twoheadrightarrow EE'', OE'', v'$ $\exists$ a bijection $\sim$ such that $EE' \sim EE''$, $OE' \sim OE''$ and $v \sim v'$.*

This is proved by a careful analysis of the possible reduction paths. The key observation is that if a query does not read an extent then its evaluation is invariant under removing those objects from the object environment, or adding

$$\frac{}{E; D; Q \vdash_{\mathrm{ioql}} i \colon \mathsf{int} \,!\, \emptyset}\,(\text{Int}) \qquad \frac{}{E; D; Q \vdash_{\mathrm{ioql}} \texttt{true} \colon \mathsf{bool} \,!\, \emptyset}\,(\text{Bool}) \qquad \frac{}{E; D; Q \vdash_{\mathrm{ioql}} \texttt{false} \colon \mathsf{bool} \,!\, \emptyset}\,(\text{Bool})$$

$$\frac{}{E; D; Q, x \colon \sigma \vdash_{\mathrm{ioql}} x \colon \sigma \,!\, \emptyset}\,(\text{Id}) \qquad \frac{}{E, e \colon \mathtt{C}; D; Q \vdash_{\mathrm{ioql}} e \colon \mathsf{set}(\mathtt{C}) \,!\, \mathsf{R}(\mathtt{C})}\,(\text{Extent})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \mathsf{set}(\sigma) \,!\, \epsilon}{E; D; Q \vdash_{\mathrm{ioql}} \texttt{size}(q) \colon \mathsf{int} \,!\, \epsilon}\,(\text{Size})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_0 \colon \sigma_0 \,!\, \epsilon_0 \quad \cdots \quad E; D; Q \vdash_{\mathrm{ioql}} q_k \colon \sigma_k \,!\, \epsilon_k \quad d \colon \sigma'_0, \ldots, \sigma'_k \xrightarrow{\epsilon} \sigma \in D \quad \forall i \in 0..k.\sigma_i \leq \sigma'_i}{E; D; Q \vdash_{\mathrm{ioql}} d(q_0, \ldots, q_k) \colon \sigma \,!\, \epsilon_0 \cup \cdots \cup \epsilon_k \cup \epsilon}\,(\text{Def})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_0 \colon \sigma_0 \,!\, \epsilon_0 \quad \cdots \quad E; D; Q \vdash_{\mathrm{ioql}} q_k \colon \sigma_k \,!\, \epsilon_k \quad \forall i \in 0..k.\sigma_i \leq \sigma}{E; D; Q \vdash_{\mathrm{ioql}} \{q_0, \ldots, q_k\} \colon \mathsf{set}(\sigma) \,!\, \epsilon_0 \cup \cdots \cup \epsilon_k}\,(\text{Set})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \mathsf{set}(\sigma) \,!\, \epsilon_1 \quad E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathsf{set}(\sigma) \,!\, \epsilon_2}{E; D; Q \vdash_{\mathrm{ioql}} q_1 \cup q_2 \colon \mathsf{set}(\sigma) \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Union})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \mathsf{int} \,!\, \epsilon_1 \quad E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathsf{int} \,!\, \epsilon_2}{E; D; Q \vdash_{\mathrm{ioql}} q_1 + q_2 \colon \mathsf{int} \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Add})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \mathsf{int} \,!\, \epsilon_1 \quad E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathsf{int} \,!\, \epsilon_2}{E; D; Q \vdash_{\mathrm{ioql}} q_1 \texttt{=} q_2 \colon \mathsf{bool} \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Int eq}) \qquad \frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \mathtt{C} \,!\, \epsilon_1 \quad E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathtt{C}' \,!\, \epsilon_2}{E; D; Q \vdash_{\mathrm{ioql}} q_1 \texttt{==} q_2 \colon \mathsf{bool} \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Obj eq})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \sigma_1 \,!\, \epsilon_1 \quad \cdots \quad E; D; Q \vdash_{\mathrm{ioql}} q_k \colon \sigma_k \,!\, \epsilon_k}{E; D; Q \vdash_{\mathrm{ioql}} \langle l_1 \colon q_1, \ldots, l_k \colon q_k \rangle \colon \langle l_1 \colon \sigma_1, \ldots, l_k \colon \sigma_k \rangle \,!\, \epsilon_1 \cup \cdots \cup \epsilon_k}\,(\text{Rec})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \langle l_1 \colon \sigma_1, \ldots, l_k \colon \sigma_k \rangle \,!\, \epsilon \quad i \in 1..k}{E; D; Q \vdash_{\mathrm{ioql}} q.l_i \colon \sigma_i \,!\, \epsilon}\,(\text{Rec access})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_1 \colon \mathsf{bool} \,!\, \epsilon_1 \quad E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \sigma_2 \,!\, \epsilon_2 \quad E; D; Q \vdash_{\mathrm{ioql}} q_3 \colon \sigma_3 \,!\, \epsilon_3 \quad \sigma_2 \leq \sigma \text{ and } \sigma_3 \leq \sigma}{E; D; Q \vdash_{\mathrm{ioql}} \texttt{if}\ q_1\ \texttt{then}\ q_2\ \texttt{else}\ q_3 \colon \sigma \,!\, \epsilon_1 \cup \epsilon_2 \cup \epsilon_3}\,(\text{Cond})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \mathtt{C} \,!\, \epsilon \quad \mathit{atype}(\mathtt{C}, a) = \phi}{E; D; Q \vdash_{\mathrm{ioql}} q.a \colon \phi \,!\, \epsilon}\,(\text{Attribute}) \qquad \frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \mathtt{C} \,!\, \epsilon \quad \mathtt{C} \leq \mathtt{C}'}{E; D; Q \vdash_{\mathrm{ioql}} (\mathtt{C}')q \colon \mathtt{C}' \,!\, \epsilon}\,(\text{Upcast})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_0 \colon \phi_0 \,!\, \epsilon_0 \quad \cdots \quad E; D; Q \vdash_{\mathrm{ioql}} q_k \colon \phi_k \,!\, \epsilon_k \quad \mathit{atypes}(\mathtt{C}) = [a_0 \colon \phi'_0, \ldots, a_k \colon \phi'_k] \quad \forall i \in 0..k.\phi_i \leq \phi'_i}{E; D; Q \vdash_{\mathrm{ioql}} \texttt{new}\,\mathtt{C}(a_0 \colon q_0, \ldots, a_k \colon q_k) \colon \mathtt{C} \,!\, \epsilon_0 \cup \cdots \cup \epsilon_k \cup \mathsf{A}(\mathtt{C})}\,(\text{New})$$

$$\frac{\begin{array}{l} E; D; Q \vdash_{\mathrm{ioql}} q \colon \mathtt{C} \,!\, \epsilon' \\ E; D; Q \vdash_{\mathrm{ioql}} q_0 \colon \phi_0 \,!\, \epsilon_0 \quad \cdots \quad E; D; Q \vdash_{\mathrm{ioql}} q_k \colon \phi_k \,!\, \epsilon_k \quad \mathit{mtype}(\mathtt{C}, m) = \phi'_0, \ldots, \phi'_k \xrightarrow{\epsilon''} \phi \quad \forall i \in 0..k.\phi_i \leq \phi'_i \end{array}}{E; D; Q \vdash_{\mathrm{ioql}} q.m(q_0, \ldots, q_k) \colon \phi \,!\, \epsilon_0 \cup \cdots \cup \epsilon_k \cup \epsilon' \cup \epsilon''}\,(\text{Method})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \sigma \,!\, \epsilon}{E; D; Q \vdash_{\mathrm{ioql}} \{q \mid \} \colon \mathsf{set}(\sigma) \,!\, \epsilon}\,(\text{Comp1}) \qquad \frac{E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathsf{set}(\sigma) \,!\, \epsilon_2 \quad E; D; Q, x \colon \sigma \vdash_{\mathrm{ioql}} \{q_1 \mid \overrightarrow{cq}\} \colon \sigma' \,!\, \epsilon_1}{E; D; Q \vdash_{\mathrm{ioql}} \{q_1 \mid x \leftarrow q_2, \overrightarrow{cq}\} \colon \sigma' \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Comp2})$$

$$\frac{E; D; Q \vdash_{\mathrm{ioql}} q_2 \colon \mathsf{bool} \,!\, \epsilon_2 \quad E; D; Q \vdash_{\mathrm{ioql}} \{q_1 \mid \overrightarrow{cq}\} \colon \sigma' \,!\, \epsilon_1}{E; D; Q \vdash_{\mathrm{ioql}} \{q_1 \mid q_2, \overrightarrow{cq}\} \colon \sigma' \,!\, \epsilon_1 \cup \epsilon_2}\,(\text{Comp3}) \qquad \frac{E; D; Q \vdash_{\mathrm{ioql}} q \colon \sigma \,!\, \epsilon \quad \epsilon \subseteq \epsilon'}{E; D; Q \vdash_{\mathrm{ioql}} q \colon \sigma \,!\, \epsilon'}\,(\text{Does})$$

Figure 3: Effect type system for IOQL

(Definition)	$DE \vdash EE, OE, d(\vec{v}) \xrightarrow{\emptyset} q[\vec{x} := \vec{v}]$	where $DE(d) = \lambda\vec{x}.q$
(Extent)	$DE \vdash EE, OE, e \xrightarrow{\mathsf{R(C)}} v$	where $EE(e) = (\mathtt{C}, v)$
(Size)	$DE \vdash EE, OE, \mathtt{size}(\{v_0, \dots, v_k\}) \xrightarrow{\emptyset} k$	
(Union)	$DE \vdash EE, OE, v_1 \cup v_2 \xrightarrow{\emptyset} v_3$	where $v_3 = v_1 \cup v_2$
(Addition)	$DE \vdash EE, OE, i_1 + i_2 \xrightarrow{\emptyset} i_3$	where $i_3 = i_1 + i_2$
(Int eq)	$DE \vdash EE, OE, i_1 \mathtt{=} i_2 \xrightarrow{\emptyset} b$	where $b \overset{\text{def}}{=} \begin{cases} \mathtt{true} & \text{if } i_1 = i_2 \\ \mathtt{false} & \text{otherwise} \end{cases}$
(Object eq)	$DE \vdash EE, OE, o_1 \mathtt{==} o_2 \xrightarrow{\emptyset} b$	where $b \overset{\text{def}}{=} \begin{cases} \mathtt{true} & \text{if } o_1 = o_2 \\ \mathtt{false} & \text{otherwise} \end{cases}$ and $OE(o_1) = \ll \mathtt{C}_1, \dots \gg$ and $OE(o_2) = \ll \mathtt{C}_2, \dots \gg$
(Cond1)	$DE \vdash EE, OE, \mathtt{if\ true\ then}\ q_1\ \mathtt{else}\ q_2 \xrightarrow{\emptyset} q_1$	
(Cond 2)	$DE \vdash EE, OE, \mathtt{if\ false\ then}\ q_1\ \mathtt{else}\ q_2 \xrightarrow{\emptyset} q_2$	
(Record)	$DE \vdash EE, OE, \langle l_1 : v_1, \dots, l_k : v_k \rangle.l_i \xrightarrow{\emptyset} v_i$	$1 \leq i \leq k$
(Attribute)	$DE \vdash EE, OE, o.a_i \xrightarrow{\emptyset} v_i$	where $OE(o) = \ll \mathtt{C}, a_1 : v_1, \dots, a_k : v_k \gg$ and $1 \leq i \leq k$
(Upcast)	$DE \vdash EE, OE, (\mathtt{C}')o \xrightarrow{\emptyset} o$	where $OE(o) = \ll \mathtt{C}, \dots \gg$ and $\mathtt{C} \leq \mathtt{C}'$
(New)	$DE \vdash EE, OE, \mathtt{new}\ \mathtt{C}(a_0 : v_0, \dots, a_k : v_k) \xrightarrow{\mathsf{A(C)}} EE', OE', o$	where fresh $o \notin \mathrm{dom}(OE)$ and $OE' \overset{\text{def}}{=} OE[o \mapsto \ll \mathtt{C}, a_0 : v_0, \dots, a_k : v_k \gg]$ and $EE(e) = (\mathtt{C}, v)$ and $EE' \overset{\text{def}}{=} EE[e \mapsto (\mathtt{C}, v \cup \{o\})]$
(Method)	$DE \vdash EE, OE, o.m(\vec{v}) \xrightarrow{\emptyset} v$	where $OE(o) = \ll \mathtt{C}, \dots \gg$ and $mbody(\mathtt{C}, m) = \lambda\vec{x}.body$ and $OE, body[\vec{x} := \vec{v}, \mathtt{this} := o] \Downarrow v \,!\, \epsilon$
(Empty comp)	$DE \vdash EE, OE, \{v \mid \} \xrightarrow{\emptyset} \{v\}$	
(True comp)	$DE \vdash EE, OE, \{q \mid \mathtt{true}, \overrightarrow{cq}\} \xrightarrow{\emptyset} \{q \mid \overrightarrow{cq}\}$	
(False comp)	$DE \vdash EE, OE, \{q \mid \mathtt{false}, \overrightarrow{cq}\} \xrightarrow{\emptyset} \{\,\}$	
(Triv comp)	$DE \vdash EE, OE, \{q \mid x \leftarrow \{\,\}, \overrightarrow{cq}\} \xrightarrow{\emptyset} \{\,\}$	
(ND comp)	$DE \vdash EE, OE, \{q \mid x \leftarrow \{v_1, \dots, v_k\}, \overrightarrow{cq}\} \xrightarrow{\emptyset}$ $(\{q \mid \overrightarrow{cq}\}[x := v_i]) \cup \{q \mid x \leftarrow v_{\text{rest}}, \overrightarrow{cq}\}$	 for some $i \in 1..k$ and $v_{\text{rest}} \overset{\text{def}}{=} \{v_1, \dots, v_k\} - v_i$

$$\frac{DE \vdash EE, OE, q \xrightarrow{\epsilon} EE', OE', q'}{DE \vdash EE, OE, \mathcal{E}[q] \xrightarrow{\epsilon} EE', OE', \mathcal{E}[q']}\ \text{(Context)}$$

Figure 4: Instrumented operational semantics for IOQL

more objects to that extent. (The bijection is necessary to handle the fresh oid generation.)

We can also use the effect information to enable query optimizations. As we saw at the start of this section, common optimizations such as commutativity of set intersection or union are no longer straightforwardly applicable. However we can see that if the two components of the commutative binary set operators do not interfere, then it is safe to commute their order. Again consider a new type system $\vdash''$ where e.g. the (Union) rule ensures a check of non-interference.

THEOREM 8 (SAFE COMMUTATIVITY). *If* $E, D, Q \vdash'' EE, DE, OE, q \cup q' : \sigma \mathrel{!} \epsilon$ *then* $\forall v$ *such that* $DE \vdash EE, OE, q \cup q' \twoheadrightarrow EE', OE', v$, *then* $\exists v'$ *and a bijection* $\sim$ *such that* $DE \vdash EE, OE, q' \cup q \twoheadrightarrow EE'', OE'', v'$ *and* $EE' \sim EE'', OE' \sim OE''$ *and* $v \sim v'$.

5. EXTENDING IOQL WITH METHODS

Thus far IOQL can only invoke read-only methods. Clearly there is a design-space where we can vary to what degree the method language can access and side-effect the database. One advantage of the techniques employed in this paper is that it is quite easy to formally specify the various options in this design space.

For example, consider an extreme point in this design space where we allow methods to read, add to and update the database. We should then specify an operational semantics for the method language, which would take the form $EE, OE, \mathit{code} \Downarrow EE', OE', \mathit{result}$, where EE and OE are the extent and object environments, *code* is the method body, and *result* is the result of executing the method body.

All that is then needed is a small change to the IOQL semantics that models method invocation.

$$\begin{array}{ll}(\text{Method}) & DE \vdash EE, OE, o.m(\vec{v}) \to EE', OE', v \\ & \quad \text{where } OE(o) = \ll \mathtt{C}, \ldots \gg \\ & \quad \text{and } \mathit{mbody}(\mathtt{C}, m) = \lambda\vec{x}.\mathit{body} \\ & \quad \text{and } EE, OE, \mathit{body}[\vec{x} := \vec{v}, \mathtt{this} := o] \\ & \qquad \Downarrow EE', OE', v\end{array}$$

The point here is that the method body can update both the extent and object environments.

Note 6. This semantics models ODMG OQL where queries can invoke arbitrary side-effecting methods, which are written in any one of the languages for which there is a binding. (It is therefore perhaps a little misleading to suggest that OQL is a functional language, c.f. [8, p.89]!)

In an extended version of this paper we detail a method language that is a valid fragment of Java. We allow the methods to read, add to and update the database. We prove a type soundness property for the version of IOQL that can invoke these Java methods.

6. RELATED WORK

In this section we briefly review other related work. We divide this work into the three areas covered by this work: type systems, semantics and update analysis.

6.1 Type systems

Alagić [2] considered aspects of the ODMG OQL type system. He showed, by providing a counter-example, that if collections are typed using a non-generic class type, e.g. `Set`, then type soundness does not hold. (It is not clear to us however that this is what the ODMG intended.) In this paper, we *prove* that using generic literal types (e.g. $\mathtt{set}(\sigma)$) for collections—perhaps what the ODMG intended—yields a sound type system.

Riedell and Scholl [21] give type rules for a fragment of ODMG OQL. Whilst interesting, their type system is quite different from that proposed by the ODMG, and moreover, is radically different from those used in conventional programming languages. The resulting problematic issue of inter-language working, and type checking method calls is not addressed.

6.2 Formal semantics

When defining the "semantics" of a query language, other authors typically provide a simple set-theoretic interpretation, e.g. [5]. Unfortunately, this approach does *not* work if queries might not terminate: what denotation is given to a non-terminating query? As most query languages provide access to methods written in third-party programming languages, and checking for termination of code is undecidable, this is a serious shortcoming. To handle non-termination, one has to move from a set-theoretic setting to a *domain*-theoretic setting [24]. Operational semantics requires much less mathematical overheads than domain theory.

Fegaras [9] considered a query language that can update database objects. However his objects are really SML-like references; object features such as classes, subtyping and so on are not supported. He defined a set-theoretic semantics (again, avoiding the issue of non-termination) that simply maps a query to the set of all possible answers. No consideration was given to the *dynamics* of query evaluation.

The only use of operational semantics to define the dynamics of query evaluation that we are aware of is in the draft definition of XQuery [26]. However this considers a quite different data model, and defines a deterministic query language (the iteration is over sequences).

6.3 Update analysis

A number of papers have considered the problem of non-determinism resulting from combining queries and updates. Andries et al. [3] consider the case of conflicts between a set of queries that could update the database, and define three classes of update independence. (Unfortunately, as the authors observe, all three notions are undecidable in general.) The main positive result in their paper is to show that these notions are decidable for updates written in a weak fragment of relational algebra. It is unclear how this work can be applied to the object setting considered here, and how it applies to potential non-termination. Moreover we are interested in update languages that are more powerful than weak fragments of the relational algebra.

Liefke and Davidson [17] present a detailed analysis of queries written in an object-oriented language called CPL+, that contains specific update primitives. It would be interesting to see if their system could be proved correct with respect to an operational semantics.

7. CONCLUSIONS AND FUTURE WORK

In this paper we have applied some dominant themes in programming language research: type systems and operational semantics, to a prototypical object query language.

We have proved a type soundness property for our language; a property that has been claimed to fail for ODMG OQL.

Unlike other object query languages, we have included a number of realistic features, including object identity, object creation and invocation of methods that need not terminate. In this rich setting we have seen that the dynamics of query execution is subtle, and that our operational semantics can aid reasoning.

To address issues such as non-determinism and query optimization we have defined an effects system. This is an adjunct to the type system, that is trivial to implement, and provides information about the computational effects of a query. It is important to recall that this is a static, compile-time analysis. This effect information can be used to remove non-determinism and to help in formulating query optimizations. It is perhaps worth pointing out that other analyses could be used instead to infer similar information. However, one advantage of the effects approach is that we are able to establish correctness relatively easily.

Clearly this is preliminary work, and much more remains to make this a viable technique to assist in building real-work query compilers and optimizers. In future work we plan to address query optimization more thoroughly, verifying the optimizations suggested e.g. by Cluet and Moerkette [5] and Fegaras and Maier [10]. We also plan to develop notions of query equivalence based upon "contextual equivalence", which is a common notion for programming languages [12].

We conclude by pointing out that whilst this paper has considered only object database systems, these techniques readily apply to other paradigms, including object-relational. In their compelling analysis, Carey and deWitt [4] identify *client integration* as an important research challenge for future-generation database systems. In our opinion, tighter integration between programming languages and query languages, along with full object querying and navigation, requires precise, formal specification to enable the exploration of the design space and identification of options to be implemented. We have shown in this paper that some of the techniques common in the analysis of programming languages can be applied successfully to query languages. In future work we plan to analyse and formally specify the object model of SQL:1999.

Acknowledgements. I am grateful to Matthew Parkinson for many discussions about objects, and the referees for suggesting significant improvements to the paper.

8. REFERENCES

[1] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1999.

[2] S. Alagić. Type-checking OQL queries in the ODMG type systems. *ACM TODS*, 24(3):319–360, 1999.

[3] M. Andries, L. Cabibbo, J. Paredaens, and J. Van den Bussche. Applying an update method to a set of receivers. *ACM TODS*, 26(1):1–40, 2001.

[4] M. Carey and D. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of VLDB*, 1996.

[5] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proceedings of Workshop on Database Programming Languages*, pages 226–242, 1993.

[6] C.J. Date. Some principles of good language design. *ACM SIGMOD Record*, 14(3):1–7, 1984.

[7] J. Van den Bussche and D. Van Gucht. A semideterministic approach to object creation and nondeterminism in database queries. *Journal of Computer and System Sciences*, 54(1):34–47, 1997.

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

[9] L. Fegaras. Optimizing queries with object updates. *Journal of Intelligent Information Systems*, 12(2–3):219–242, 1999.

[10] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM TODS*, 25(4):457–516, 2000.

[11] D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, 1986.

[12] A. Gordon, P. Hankin, and S. Lassen. Compilation and equivalence of imperative objects. *Journal of Functional Programming*, 9(4):373–426, 1999.

[13] A. Gordon and D. Syme. Typing a multilanguage intermediate code. In *Proceedings of POPL*, 2001.

[14] A. Greenhouse and J. Boyland. An object-oriented effects system. In *Proceedings of ECOOP*, 1999.

[15] J. Hellerstein. Optimization techniques for queries with expensive methods. *ACM TODS*, 23(2):113–157, 1998.

[16] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[17] H. Liefke and S. Davidson. Updates and nondeterminism in object-oriented databases. Technical Report MS-CIS-99-11, University of Pennsylvania, 1999.

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

[19] G. Mitchell, S. Zdonik, and U. Dayal. Object-oriented query optimization: what's the problem? Technical Report CS-91-41, Brown University, 1991.

[20] T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe—definitely. In *Proceedings of POPL*, 1998.

[21] H. Riedell and M.H. Scholl. The CROQUE model: Formalization of the data model and the query language. Technical Report 23, University of Konstänz, 1996.

[22] P. Seshadri and M. Paskin. PREDATOR: An OR-DBMS with enhanced data types. In *Proceedings of ACM SIGMOD*, pages 568–571, 1997.

[23] A. Trigoni and G.M. Bierman. Inferring the principal type and schema requirements of an OQL query. In *Proceedings of BNCOD*, 2001.

[24] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

[25] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[26] XQuery 1.0 formal semantics. Working draft June 2001.