The Translation of Object-Oriented Queries to Optimized Datalog Programs

Tore Risch

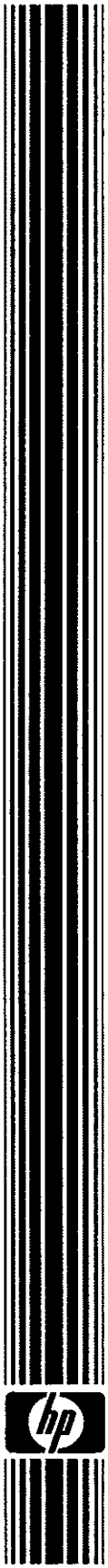
Technical Report HPL–DTD–91–9

Feb 20 1991

The advent of object-oriented DBMSs has created a demand for object-oriented (OO) declarative query languages. Analogous to the relational environments, the query processor has the responsibility of translating queries into efficient execution plans. In this paper we address the translation of queries in the OSQL language, which is the lingua franca of HP's IRIS system. The usefulness of the optimization of query execution cannot be understated based on the relational experience.

We infer the need for certain optimization methods from the query constructs and usage patterns that are particular to the OO paradigm. Further, we observe the correspondence between the extensive optimization technology developed in the context of relational/Datalog queries and this new arena. Accordingly, we translate OSQL queries into optimized Datalog programs, wherein traditional optimization techniques can be utilized straightforwardly.

This approach is demonstrated in a fully functional prototype implementation of OSQL.



HEWLETT
PACKARD
Database Technology Department
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

The Translation of Object-Oriented Queries to Optimized Datalog Programs*

Tore Risch
HP Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94303

February 20, 1991

Abstract

The advent of object-oriented DBMSs has created a demand for object-oriented (OO) declarative query languages. Analogous to the relational environments, the query processor has the responsibility of translating queries into efficient execution plans. In this paper we address the translation of queries in the OSQL language, which is the lingua franca of HP's IRIS system. The usefulness of the optimization of query execution cannot be understated based on the relational experience.

We infer the need for certain optimization methods from the query constructs and usage patterns that are particular to the OO paradigm. Further, we observe the correspondence between the extensive optimization technology developed in the context of relational/Datalog queries and this new arena. Accordingly, we translate OSQL queries into optimized Datalog programs, wherein traditional optimization techniques can be utilized straightforwardly.

This approach is demonstrated in a fully functional prototype implementation of OSQL.

*This is a working paper. The author greatly appreciates comments.

1 Introduction

OSQL [1] is a high-level query language for the object-oriented DBMS Iris [6]. Iris has a functional data model similar to DAPLEX[13], and OSQL can be regarded as an extension and modification of SQL that works over such an object-oriented data model. In particular, OSQL supports declarative set-oriented queries similar to SQL, extended with abstract data types, inheritance, and object identifiers [6]. The basic query primitive of OSQL is a **select** statement related to the corresponding SQL statement.

OSQL provides a declarative access language to object databases and thus it must be optimized before execution. There has been substantial research on optimizing relational query languages, and the work on optimizing logical query languages, e.g., $\mathcal{LDL}$ [3, 4] and NAIL! [14, 15], is closely related. One may actually regard a logical query language such as Datalog [14, 15] as a canonical relational query language from which relational algebra can be easily generated for subsequent interpretation [14]. It would be advantageous if some of the work invested in optimizing and interpreting Datalog could also be applied to OSQL.

This paper is based on an optimizing translator from OSQL into a Datalog dialect. By optimizing the generated Datalog programs, we are able to leverage some of the vast amount of research invested in optimizing relational and logical data languages. We will show how OSQL queries can be translated rather easily into Datalog rules. As part of the translation into Datalog rules, we apply some optimizations on OSQL expressions relying on object-oriented properties of OSQL. In a second optimization step, the generated Datalog rules are transformed into an equivalent set of more efficient Datalog rules. We will describe some optimization techniques we have found useful to optimize Datalog expressions that are generated from object-oriented OSQL query patterns.

We distinguish between two kinds of optimizations:

- *Ameliorations* [2] are unambiguous transformations that guarantee improved final execution speed irrespective of any other decisions.
- *Optimizations* are transformations that aim at improving the execution speed by choosing among several execution plans.

We have a fully implemented system that does the transformations described in this pa-

per. The system, called ADB (Active DataBase), includes a main memory interpreter for Datalog rules. The system allows OSQL-based databases to be completely represented, updated, and queried. The interpreter supports referential integrity so that a database update invalidates definitions that depend on deleted objects. The referential integrity system guarantees that critical assumptions made by the code optimizer are always valid, or otherwise the code is invalidated.

The system is designed for efficient execution in main memory. To achieve good performance we have carefully optimized the representation of critical system data structures, e.g., object representation, type information, and the representation of function definitions. For these cases we use tailored main memory data structure representations, rather than using relations. For example, our object identifiers are represented as variable-length records, where one field points to the object's type information and another points to its function definition. It is critical that this information is represented efficiently since it is extensively looked up both during compilation and during interpretation of OSQL functions.

We use an extended foreign function mechanism to give transparent access to special-purpose data structures such as the type system. The architecture relies on optimization of such foreign function calls.

In section 2 we give a short introduction to the basic concepts of objects, types, and functions in OSQL. Section 3 describes the abstract query representations used during the transformations. Section 4 and 5 describe ameliorations and optimizations that are used. Section 6 contains some performance measures indicating the importance of our optimizations, and finally, in section 7, we summarize our experiences and point out some possible future work.

2 Basic OSQL Concepts

Throughout this paper we will use OSQL [6, 1] as a query language to model object-oriented databases. OSQL models objects based on the three concepts of *types*, *objects*, and *functions*. In OSQL, objects are atomic object identifiers (OIDs), types classify OIDs into groups, and functions associate properties and relationships between objects. For a more complete description of OSQL and Iris, see [6]. Here we will review some of the basic OSQL constructs for maintaining and searching object-oriented databases.

2.1 Types

The statement `create type` creates a new OSQL type, optionally as a subtype of one or more other types. For example,

```
create type Person;
create type Student subtype of Person;
                                /* Illustrates inheritance */
create type Teacher subtype of Person;
create type TeachingAssistant subtype of Teacher, Student;
                                /* Illustrates multiple inheritance */
```

With the `subtype of` construct, types will be partially ordered in an acyclic type graph.

2.2 Objects

A special variant of `create` is used for creating and initiating objects of prespecified types. For example,

```
create Student;
```

will create a new object of type `Student`. Objects may have one or several *properties* (or attributes) which are actually modeled as typed *functions*. For example, a person may have a property, `Name`, which is actually represented as a function that, given an object of type *Person*, returns a string. The properties may be set as part of the `create` statement, e.g.:

```
create Student (Name) instance ("Karl");
```

will create a new `Student` and also set the property `Name` to `"Karl"`.

The system maintains referential integrity of objects so that if an object is deleted all references to that object are also deleted. If a type is deleted all subtypes are deleted as well as all functions defined over the deleted types.

2.3 Functions

OSQL functions model both object attributes as well as relationships between objects.

OSQL functions are of three kinds:

1. *Stored functions* are stored as database tables.
2. *Foreign functions* are programmer-defined operators implemented by escaping to an underlying procedural language.
3. *Derived functions* are derived combinations of other functions; derived functions are related to views in relational databases.

Figure 1 gives some examples of OSQL function definitions that will be used in our examples in the rest of this paper.

Methods of the type found in Smalltalk and C++ are modeled in OSQL as functions whose only argument is bound to an object identifier. In the example, objects of type `Person` have the attributes `Name, Income, Taxes, Parent, NetIncome, GrandSParentNetIncome,` etc., represented as OSQL functions of a single argument bound to objects of type `Person`.

The functions `Income`, `NetIncome` and `BothIncomes` are examples of *overloaded* functions that have different definitions depending on the type of their first argument. For example, there are two variants, or *resolvents*, of the `Income` function, one for incomes of given persons and another one for incomes of given names of persons.

Resolvents can be defined as any of the three basic function types. Overloaded functions handle the cases when different classes have attributes with the same name; the overloading on the first argument (the object instance) does the class dispatch. CLOS[7] represents methods similarly by using overloaded functions.

The function `Minus` in the definition of the OSQL function `NetIncome` is an example of a *foreign* function call.

An OSQL function may return a set of values; e.g. `Parent(Person p)` will normally return two parents.

```
create function Name(Person p) -> Charstring nm;/* stored */
create function Income(Person p) -> Integer i;  /* stored */
create function Taxes(Person p) -> Integer i;   /* stored */
create function Parent(Person c) -> Person p;   /* stored */
create function PersonNamed(Charstring nm) -> Person p as
    select p                                    /* derived */
    for each Person p
    where Name(p) = nm;
create function Income(Charstring nm) -> Integer i as
    select Income(PersonNamed(nm));             /* overloaded */
create function NetIncome(Person p) -> Integer n as
    select Minus(Income(p),Taxes(p));           /* derived */
create function BothIncomes(Person p) -> <Integer i, Integer n> as
    select Income(p), NetIncome(p);             /* two results */
create function BothIncomes(Charstring nm) -> <Integer i, Integer n> as
    select i, n
    for each Integer i, Integer n
    where <i,n> = BothIncomes(PersonNamed(nm)); /* overloaded */
create function NetIncome(Charstring nm) -> Integer n as
    select NetIncome(PersonNamed(nm));          /* overloaded */
create function SParent(Person c) -> Student s as
    select p
    for each Student p
    where p=Parent(c);      /* Parent if parent is student */
create function GrandSParentNetIncome(Person c) -> Integer ni as
    select ni                           /* Net income of grandparent
                                           if grandparent is student */
    for each Integer ni, Person gp, Person p
    where ni = NetIncome(gp) and
          gp = SParent(p) and
          p = Parent(c);
```

Figure 1: Examples of OSQL queries

The arguments and results of a function together with their types are called the *signature* of the function. We denote signatures by

`f(`T_1 P_1`,...,`T_n P_n`) -> `U_1 Q_1`,...,`U_m Q_m

For example:

```
Income(Person p) -> Integer i
BothIncomes(Person p) -> Integer i, Integer n
BothIncomes(Charstring nm) -> Integer i, Integer n
```

The *arguments* of a function are named by $P_1, \ldots, P_n$; their actual values are restricted to types $T_1, \ldots, T_n$. A function can have more than one result, named by $Q_1, \ldots, Q_m$ restricted to types $U_1, \ldots, U_m$. Syntactically multivalued functions are called by using a bracket notation, as in `BothIncomes`.

OSQL has a set-oriented `select` statement to declaratively specify queries over the database. Derived functions are specified using select statements referencing other derived or stored functions. Select expressions allow functions to be used backwards, for example `Name` in the function `PersonNamed`.

The select statement has the syntax:

```
select <results>
for each <declarations>
where <predicate>
```

The `<results>` is a list of expressions denoting the result(s) of the function. The `<results>` can be either a list of result variables or a list of function expressions, as illustrated by the two overloaded definitions of `BothIncomes`.

We will here assume that the predicate is always a conjunct of simple predicates. Our implementation allows arbitrary nesting of conjuncts and disjuncts.

3 Query Abstractions

Our algorithms operate on queries represented in canonical formats. In this section we briefly describe the formats of these abstracted queries. In the next sections we describe our optimization algorithms as transformations on the abstracted queries. We present the query abstractions by means of examples. It is outside the scope of the paper to describe how the system automatically translates between the abstractions.

3.1 Flattened OSQL Queries

The translation of OSQL queries into Datalog assumes `select` expressions where there are no functions references in the result list and where no function nesting occurs in the predicate. We say that such `select` expressions are *flattened*. We flatten `select` expressions by introducing new variables to remove unneeded function references. For example, the two resolvents `BothIncomes` would be translated into the following flattened definitions:

```
create function BothIncomes(Person p) -> <Integer i, Integer n> as
   select _v1, _v2
   for each Integer _v1, Integer _v2, Person p
   where _v1=Income(p) and _v2=Netincome(p);
create function BothIncomes(Charstring nm) -> <Integer i, Integer n> as
    select i, n
    for each Integer i, Integer n, Person _v1
    where _v1 = PersonNamed(nm) and
          <i,n> = BothIncomes(_v1);
```

3.2 Type-adorned Queries

We need a way to identify each resolvent of an overloaded function with a unique name. We create a unique naming scheme for resolvent functions by annotating the name of the overloaded function with the name of its signature types. For example, the definitions of

Income will have the *type-adorned resolvents*
$\texttt{Income}_{Person->Integer}$ and $\texttt{Income}_{Charstring->Integer}$.

The system creates type-adorned function definitions by substituting overloaded functions in a flattened `select` expression with their type-adorned resolvents. We call such a substitution algorithm *overload resolution*,which is an amelioration we will discuss in more detail later. For example, after type annotation, the definition of `GrandSParentNetIncome` will be transformed into:

```
create function GrandSParentNetIncome_Person->Integer(c)
                -> Integer ni as
       select ni
       for each Person c, Integer ni, Student gp, Person p
       where ni = NetIncome_Person->Integer(gp) and
             gp = SParent_Person->Student(p) and
             p = Parent_Person->Person(c);
```

3.3 Datalog Queries

The OSQL compiler transforms the type-adorned resolvents into corresponding type-adorned Datalog rules and facts. In general, using Datalog terminology [14], we translate stored OSQL functions into extensional (EDB) predicates, derived functions into Datalog rules, and foreign functions into built-in Datalog predicates, which in our case are user-definable.

In our examples of generated Datalog programs, we use the conventional Datalog naming scheme where symbols spelled with capital letters denote variables and lowercase symbols denote constants. We type-adorn the Datalog rules and facts as before, with the difference that we do not differentiate between arguments and results in the Datalog rules.

For example the two resolvent functions for Income would generate the following two type-adorned EDB predicates:

$\texttt{income}_{Person,Integer}\texttt{(P,I)}$
$\texttt{income}_{Charstring,Integer}\texttt{(D,I)}$

The type-adorned definition of the derived function `GrandSParentNetIncome` would generate the following type-adorned Datalog rule:

$$\texttt{grandsparentnetincome}_{Person,Integer}\texttt{(C,NI) :-}$$
$$\texttt{netincome}_{Person,Integer}\texttt{(GP,NI) \&}$$
$$\texttt{sparent}_{Person,Student}\texttt{(P,GP) \&}$$
$$\texttt{parent}_{Person,Person}\texttt{(C,P).}$$

In general the type-adorned Datalog rules are direct mappings of the type-checked resolvents where the head of the rule is determined by the signature of the resolvent and the body of the rule its `select` expression.

3.4 Binding Pattern Adorned Rules

The optimizer will specialize further type-adorned Datalog rules into *binding pattern adorned* [15] Datalog rules. In binding pattern adorned Datalog rules, each literal is adorned with binding patterns, where superscripts ^{b} and ^{f} indicate whether the corresponding argument position is bound or free, respectively. The binding pattern adornments depend on which variables are arguments of the function as well as the order of the literals of the rule body.

The goal of the translator/optimizer is to translate each OSQL function definition into an optimized, typed-adorned, and binding pattern adorned Datalog rule. This Datalog rule constitutes a customized global optimization for the OSQL function. For example, $\texttt{GrandSParentNetIncome}_{Person->Integer}$ could get translated into:

$$\texttt{grandsparentnetincome}^{bf}_{Person,Integer}\texttt{(C,NI) :-}$$
$$\texttt{parent}^{bf}_{Person,Person}\texttt{(C,P) \&}$$
$$\texttt{sparent}^{bf}_{Person,Student}\texttt{(P,GP) \&}$$
$$\texttt{netincome}^{bf}_{Person,Integer}\texttt{(GP,NI).}$$

Notice that when the optimizer reorders literals in the body of a rule, different binding pattern adornments will be referenced by the literals. For example, an alternative suboptimal definition of the above rule would be:

$$\texttt{grandsparentnetincome}^{bf}_{Person,Integer}\texttt{(C,NI) :-}$$
$$\texttt{sparent}^{ff}_{Person,Student}\texttt{(P,GP) \&}$$
$$\texttt{parent}^{bb}_{Person,Person}\texttt{(C,P) \&}$$
$$\texttt{netincome}^{bf}_{Person,Integer}\texttt{(GP,NI).}$$

4 Ameliorations

Ameliorative techniques are ones that improve the execution in most (if not all) cases. For example, consider a query that has the following two conditions:

$$\texttt{income} > 50 \texttt{ and income} > 80$$

Obviously, it is unnecessary to apply both conditions and the first condition can be eliminated, resulting in improved performance. Therefore, as in this example to apply ameliorative techniques is generally useful. In this section we present a few of the ameliorative techniques that are particularly useful in the context of object-oriented queries.

4.1 Overload Resolution

Methods of objects are modeled in OSQL as functions, as was described in the introduction. The use of the same function (method) name for two object types introduces the need for overloaded functions. Overloaded functions carry the overhead of looking up which resolvent to use in each given situation. The system has an algorithm, called *overload resolution*, that for each function call finds out which resolvent to use in order to substitute the function call for a type-adorned function call.

An important amelioration is to analyze `select` expressions and perform overload resolution at compile time (early binding) rather than at run time (late binding). At compile time, the resolvents are completely determined by the types of the `for each` declared variables.[1] Compile time overload resolution has the following advantages:

[1]The early binding of overloaded functions can sometimes cause semantic problems, which are outside the scope of this paper.

1. Run time overload resolution (late binding) would require that the generated Datalog program contain variables bound to predicate names. Such higher-order predicates are inherently more difficult to optimize and interpret than ordinary Datalog rules.

2. With early binding we know at compile time exactly which resolvents are called from a given OSQL definition. This allows us to make global optimizations for each resolvent, by translating each resolvent to type-adorned and binding pattern adorned and optimized Datalog rules.

3. There is higher a cost to doing dynamic function resolvent lookup at run time. The overload resolution is a potentially expensive operation since the type hierarchy needs to be traversed to find matching resolvents.

4. By doing overload resolution at compile time we detect typing errors early.

In Iris the overload resolution algorithm is a function of the first argument only; thus given a function call,[2]

$\texttt{f}(\texttt{A}_1,\ldots,\texttt{A}_n)$ = <$\texttt{R}_1,\ldots,\texttt{R}_m$>

we get the resolvent by looking only at the name of the called function, `f`, and the type, $\texttt{T}_1$, of its first argument, $\texttt{A}_1$.[3] The overload resolution algorithm can therefore be expressed as a function with the signature

```
Resolve(Function F,Type T) -> Function R
```

returning a resolvent function, `R`, for a given function, `F`, and the type of its first argument, `T`. The current overload resolution algorithm traverses the type hierarchy bottom up from $\texttt{T}_1$ looking for resolvents, similar to method lookup in Smalltalk and other object-oriented programming languages. Overload resolution is *undefined* if the overload resolution algorithm is unable to find a resolvent. Since OSQL allows multiple inheritance, it is possible to get more than one resolvent for a given signature, in which case the overload resolution algorithm signals *ambiguous* resolvents, which is regarded as an error. We will not further

[2]In order to avoid backtracking during type checking the first type functionally determines the other types of the resolvents.

[3]It is conceivable to generalize the overload resolution algorithm to operate on other function arguments and results as well.

elaborate on the overload resolution algorithm here; we just assume that there is a function, `Resolve`, that either computes a unique resolvent, given that the first argument of a function call is known, or fails.

Given a flattened predicate, we can assume that every argument of a function referenced in the predicate is an unambiguously typed variable.

The general rule for overload resolution of `select` predicates is that if we have a function reference

$\texttt{f}(\texttt{A}_1,\ldots)$

and the type of the first argument $\texttt{A}_1$ is known to be $\texttt{T}_1$, and $\texttt{Resolve}(\texttt{f},\texttt{T}_1)=\texttt{f}_{T_1,\ldots,->\ldots}$, then we can transform the function reference into the type-adorned function reference

$\texttt{f}_{T_1,\ldots,->\ldots}(\texttt{A}_1,\ldots)$

For example, in `GrandSParentNetIncome` we use the following overload resolution transformations:

$\texttt{Resolve(NetIncome,Person)} \texttt{ -> NetIncome}_{Person->Integer}$
$\texttt{Resolve(Parent,Student)} \texttt{ -> Parent}_{Person->Person}$
$\texttt{Resolve(SParent,Person)} \texttt{ -> SParent}_{Person->Student}$

In summary, OSQL modeling of methods as functions introduces the need for overloaded functions. The method lookup is optimized by overload resolution at compile time. Global optimization needs early binding to be able to optimize the entire search expression called from an OSQL function.

4.2 Type Check Removal

The `for each` type declarations in `select` expressions restrict value types of each declared variable. In OSQL we can regard every object to have an associated set of types to which it belongs. The set of types is normally obtained by traversing the type graph upward from the declared type of the object.[4] For example, an object of type `TeachingAssistant` would have the associated type set

[4]OSQL also has a feature to allow dynamic modification of the list of types associated with an object[6].

```
{TeachingAssistant, Teacher, Student, Person, UserTypeObject}
```

For testing type membership of objects there is a built-in function, `TypesOf`, with signature

```
TypesOf(Object o) -> Type t
```

`TypesOf` returns the set of types to which a given object belongs.

The general rule for adding type checks is to add a call to `TypesOf` for each variable declared in a `select` statement in order to test that the objects bound to the variables are of the declared types.[5] For example:

```
create function GrandSParentNetIncome_Person->Integer(c)
                -> Integer ni as
       select ni
       for each Person c, Integer ni, Student gp, Person p
       where TypesOf_Object->Type(c)=typePerson and
             TypesOf_Object->Type(ni)=typeInteger and
             TypesOf_Object->Type(gp)=typeStudent and
             TypesOf_Object->Type(p)=typePerson and
             ni = NetIncome_Person->Integer(gp) and
             gp = SParent_Person->Student(p) and
             p = Parent_Person->Person(c);
```

The variables `typeInteger`, `typeStudent`, and `typePerson` refer to constants denoting type objects for types `Integer` and `Person`.

Notice here, that equality (`=`) for a multivalued function like `TypesOf` means that *there exists* a result value among the set of values returned by `TypesOf` which is equal to the type objects `typeInteger`, `typeStudent`, `typeStudent`, and `typePerson`, respectively.

We allow built-in functions to be used bi-directionally as a relationship between arguments and results. For example, `TypesOf` can be used either for testing if an object is of a given type or to get all objects of a given type. For example, the function `AllStudents` returns the set of all students:

[5] We will soon describe an amelioration to remove most of these type checks.

```
create function AllStudents() -> Student s as
    select s
    for each Student s;
```

After adding TypesOf tests to the definition of AllStudents we get:

```
create function AllStudents() -> Student s as
       select s
       for each Student s
       where TypesOf(s)=typeStudent);
```

In this case TypesOf will be run backward to get all the possible objects, s, that have type Student.

In summary, by adding calls to TypesOf we get dynamic type checks as well as typed object generators. We will now describe an amelioration to remove unnecessary calls to TypesOf.

None of the TypesOf tests in the definition of GrandSParentNetIncome are actually needed. The reason for this is that the system guarantees the integrity of OSQL functions so that whenever an argument or result of a function is of a certain type the system will make sure that actual arguments match this type. In GrandSParentNetIncome we know that the argument and result of $\texttt{Parent}_{Person->Person}$ **must** be of type Person, which obsoletes the type test $\texttt{TypesOf}_{Object->Type}$(p)=typePerson. Similarly, since we know that $\texttt{Income}_{Person->Integer}$ returns integers we may remove the test $\texttt{TypesOf}_{Object->Type}$(ni)=typeInteger. Equivalent arguments can be made to remove all other type checks in GrandSParentNetIncome.

In general, consider a type-adorned function call,

$\texttt{f}_{T_1,...,T_m->T_{m+1},...,T_{m+n}}(\texttt{A}_1,..,\texttt{A}_m) = <\texttt{A}_{m+1},...,\texttt{A}_{m+n}>$

where some variable, $\texttt{A}_j$ is declared to be of type $\texttt{D}_j$. Then we may remove the type check for $\texttt{A}_j$ if the type $\texttt{T}_j$ is equal to $\texttt{D}_j$ or a subtype of $\texttt{D}_j$, denoted as

$\texttt{T}_j \subseteq \texttt{D}_j$

We say that f is a *type container* for the variable A_j. When a stored function is updated, the integrity maintenance system makes sure that the update does not violate the type restrictions of the signature of the function. Similarly, our type checking mechanism guarantees that derived functions can never return objects that violate its type restrictions.

The function SParent is an example of a function where dynamic type checking is needed for testing if the variable p is of type Student. The type test for p cannot be removed since $\texttt{Parent}_{Person->Person}$ is not a type container for Student.

```
create function SParent_{Person->Student}(c)
                -> Person p as
       select p
       for each Person c, Student p
       where TypesOf_{Object->Type}(p)=typeStudent and
             Parent_{Person->Person}(c)=p
```

4.3 Equality Rewrite

The function ParentS returns the parents of children who are students:

```
create function ParentS(Person c) -> Person s as
    select p
    for each Person p, Student s
    where p = Parent(s) and
          s = c;              /* Parent if child is student */
```

As can be seen by the function ParentS, equality checks are convenient for testing if an object is of a given type. However, if no further optimizations are made, the calls to = will result in inefficient calls to a foreign predicate, =. We will the get the following type- and binding pattern adorned Datalog rule:[6]

$$\texttt{parents}^{bf}_{Person,Student}\texttt{(C,P)} \texttt{ :- } \texttt{parent}^{bf}_{Person,Person}\texttt{(C,P) \&}$$
$$\texttt{typesof}^{fb}_{Object,Type}\texttt{(S,typeStudent) \&}$$
$$=^{bb}_{Object,Object}\texttt{(C,S).}$$

[6]The type annotation $:=_{Object,Object}$ denotes a boolean function whose both arguments are of type Object.

It can easily be seen from the definition that the introduction of the variable s in the definition of ParentS is present only to generate a test that c is of type Student.

The general rule is that the system will eliminate calls to = by substituting the variables in the calls. The system handles equality transitivity so that if A = B and B = C then A may be substituted for C.

In the example we get the following definition after an equality substitution:

$$\texttt{parents}^{bf}_{Person,Person}\texttt{(C,P)} \texttt{ :- } \texttt{parent}^{bf}_{Person,Person}\texttt{(C,P) \&}$$
$$\texttt{typesof}^{bb}_{Object,Type}\texttt{(C,typeStudent).}$$

This is a considerable optimization in this case, since the original definition of ParentS would first get all objects, S, of type Student and then test if C is equal to S.[7]

In summary, in object-oriented queries, equality tests are often used for testing type memberships. Optimization of equality tests are therefore important.

5 Optimizations

Query optimization techniques need to be applied judiciously, because the resulting execution can be worse than before the application of the technique. Most (if not all) commercial optimizers use a cost-based optimizer that applies optimization techniques depending on the cost improvements estimated by a cost model. In this section, we describe some of the techniques that fall into this category. We describe each of these techniques by inferring the increased need in the context of OO queries and relate the applicability of the known relational/Datalog techniques.

[7] A cost-based model for reordering foreign predicates, such as the one proposed by [3], would in this case have avoided the pitfall as well by reordering typesof and =, but it would still not entirely have removed the = call.

5.1 Datalog Optimization

Choice of bindings (i.e., join ordering, selection pushing) and access methods (e.g., join method, index creation/use) are two of the useful techniques that were considered very important in relational/Datalog query optimization. In fact, these techniques can result in many orders of magnitude improvement to the query. We term this improvement Datalog optimization because the techniques can be viewed as optimizing the Datalog program by ordering the body of each rule (i.e., determine the binding pattern adornment) and choosing the appropriate access methods. Viewed in this manner, it is obvious that the Datalog program corresponding to the query can be optimized using traditional technology and can accrue the phenomenal improvement.

Intuitively, the binding pattern adornment chooses the necessary nesting and inverting of functions that are deemed useful by whatever criteria dictated by the cost model. The use of different access methods allows the efficient implementation of computation of the functions, thereby accruing the advantages of restricting the computation to a smaller relevant set of facts. The important observation here is that the correspondence between OSQL queries and Datalog programs allowsthe straightforward use of traditional optimization technology in the context of OO queries.

With the object-oriented programming style of modeling object attributes as functions, the most common usage of OSQL functions from application programs is to call OSQL functions in the *forward* direction, where the arguments are known but the results are unknown.[8] Therefore, when an OSQL function is defined, the system will always optimize its body for use in the forward direction, i.e., where the arguments of the resolvent are known but its results computed. In order to use Datalog optimization techniques, we need to know for each Datalog predicate which arguments are bound or free. For example, the typed-adorned OSQL function $\texttt{GrandSParentNetIncome}_{Person->Integer}$ will be translated into a type- and binding pattern adorned Datalog rule, $\texttt{GrandSParentNetIncome}^{bf}_{Person->Integer}$, and optimized according to that binding pattern. The unoptimized rule would look like

[8]OSQL also has primitives to modify values of stored functions, which are used when updating object properties. We do not further elaborate on OSQL function updates here, but the reader is advised to read about updates in Iris in [6].

$$\texttt{grandsparentnetincome}^{bf}_{Person,Integer}\texttt{(C,NI)} :-$$
$$\texttt{netincome}^{ff}_{Person,Integer}\texttt{(GP,NI)} \;\&$$
$$\texttt{sparent}^{ff}_{Person,Student}\texttt{(P,GP)} \;\&$$
$$\texttt{parent}^{bb}_{Person,Person}\texttt{(C,P)}.$$

The rule above is clearly extremely inefficient and needs to be optimized. In this section we describe some useful optimization techniques.

5.2 Rule Expansion

The object-oriented nature of OSQL encourages the usage of many small functions where each function corresponds to attributes of objects, and where attributes are very often defined in terms of other attributes. For example, the function `GrandSParentNetIncome` is defined in terms of the functions `Sparent`, `Parent`, and `Netincome`; `Sparent` is defined in terms of `Parent`; and `Netincome` is defined in terms of `Minus`, `Income`, and `Taxes`. To avoid the interpretation of many small Datalog rules and to be able to globally optimize expressions as large as possible, we first perform *rule expansion*[15] on the generated Datalog program, which substitutes Datalog literals with nonrecursive rule bodies.

For example, after rule expansion, the Datalog rule for `GrandSparentNetIncome` will have this (still very inefficient) definition:

$$\texttt{grandsparentnetincome}^{bf}_{Person,Integer}\texttt{(C,NI)} :-$$
$$\texttt{income}^{ff}_{Person,Integer}\texttt{(GP,\_V2)} \;\&$$
$$\texttt{taxes}^{bf}_{Person,Integer}\texttt{(GP,\_V1)} \;\&$$
$$\texttt{plus}^{bfb}_{Integer,Integer,Integer}\texttt{(\_V2,NI,\_V1)} \;\&$$
$$\texttt{typesof}^{bb}_{Object,Type}\texttt{(GP,typeStudent)} \;\&$$
$$\texttt{parent}^{fb}_{Person,Person}\texttt{(P,GP)} \;\&$$
$$\texttt{parent}^{bb}_{Person,Person}\texttt{(C,P)}.$$

First, it easy to argue that rule expansion is not always a useful transformation. This is because, in the presence of disjuncts,it would result in a large number of rules and effectively eliminate the common subexpressions. On the other hand, in the presence of foreign and derived functions with restricted allowable bindings, such transformations would make a

difference between safe and unsafe executions. This is because one ordering of the derived functions may be unsafe if imposed on the original rules whereas the different rules in the expanded program can be ordered differently. Needless to say, that lack of safety is an extreme case of unoptimized execution and examples can be constructed wherein the advantage of expanded rules ordered independently can easily outweigh the disadvantage of the lack of common subexpressions.

5.3 Binding Pattern Optimization

The rule-expanded definition of $\texttt{grandsparentnetincome}^{bf}_{Person,Integer}$ is clearly very sub-optimal. We need to reorder its literals in order to get an optimal execution plan. At this point we postpone the problem of optimizing the order in which the two foreign functions, `TypesOf` and `Plus`, are called by simply placing them at the end of the rule to guarantee safe execution. The important optimization here is to reorder the other literals.

The system uses a method that combines the bound-is-easier heuristic of NAIL! [15] with knowledge of index availability and cardinality of the EDB relations involved. By simply applying bound-is-easier in this case we get the following optimal order:

$$
\begin{aligned}
\texttt{grandsparentnetincome}^{bf}_{Person,Integer}\texttt{(C,NI)} \;\texttt{:-}\;& \\
&\texttt{parent}^{bf}_{Person,Person}\texttt{(C,P)} \;\&\\
&\texttt{parent}^{bf}_{Person,Person}\texttt{(P,GP)} \;\&\\
&\texttt{income}^{bf}_{Person,Integer}\texttt{(GP,\_V2)} \;\&\\
&\texttt{taxes}^{bf}_{Person,Integer}\texttt{(GP,\_V1)} \;\&\\
&\texttt{plus}^{bfb}_{Integer,Integer,Integer}\texttt{(\_V2,NI,\_V1)} \;\&\\
&\texttt{typesof}^{bb}_{Object,Type}\texttt{(GP,typeStudent)} \;\&
\end{aligned}
$$

5.4 Foreign Predicate Optimization

The optimization of foreign predicates can be succintly stated as the optimal adornment of the binding pattern for each foreign and derived function. Whereas a relation in a Datalog rule can be adorned with any binding, foreign and derived functions are restricted by the fact that only certain bindings are allowable by its definition. This means that the

suboptimal binding pattern adornment can result in unsafe execution, i.e., in invoking a function with bindings that cannot be supported by its definition. For example, finding all values of x that satisfy the inequality $x > 5$ is an unsafe execution that is an extreme case of suboptimal execution. Consider, for example, a foreign or derived function that is defined for two out of four bindings but both the allowable bindings are not equally efficient. For example, the function `typesOf(X, T)` computes the type of the object X. Naturally, computing the type of a given object is very efficient but finding all the objects of a given type is not.

The necessity of choosing the adornment is also evident in the ability to pose constraints in formulation of the query. Our optimization method is similar to the constraint compilation technique [10] used for efficient constraint propagation.

Consider the OSQL function `ftoc` to convert Farenheit degrees into Celsius:

```
create function ftoc(Real f) -> Real c as
  select Div(Times(Minus(f,32.),5.),9.);
```

Assume we have stored the Celsius temperature of a person and want to use the function above as a constraint to calculate the Farenheit temperature:[9]

```
create function ctemp(Person p)-> Real c;
create function ftemp(Person p)-> Real f
 as select f
 for each Real f
 where ftoc(f) = ctemp(p);
```

Our foreign predicate mechanism also allows `Minus`, and `Div` to be defined as constraints in terms of the two foreign functions, `Plus` and `Times`:

```
create function Minus(Real x, Real y) -> Real r as
  select r
  for each Real r
  where Plus(y,r) = x;
```

[9]This case is more general than Iris' foreign function implementation[5].

```
create function Div(Real x, Real y) -> Real r as
  select r
  for each Real r
  where Times(y,r) = x;
```

With the above definitions we get the following unoptimized definition of `ftemp`:

$$\begin{aligned}\texttt{ftemp}^{bf}_{Person,Real}\texttt{(P,F)} \;\texttt{:-}\; & \texttt{plus}^{bff}_{Real,Real,Real}\texttt{(32,\_V3,F)} \;\&\\ & \texttt{times}^{bbf}_{Real,Real,Real}\texttt{(\_V3,5,\_V2)} \;\&\\ & \texttt{times}^{bfb}_{Real,Real,Real}\texttt{(9,\_V1,\_V2)} \;\&\\ & \texttt{ctemp}^{bb}_{Person,Real}\texttt{(P,\_V1).}\end{aligned}$$

The problem here is that the foreign predicates `plus` and `times` can only be called if two of their arguments are known, so their calls have to be reordered by the Datalog optimizer to generate a safe evaluation. In the example above, $\texttt{plus}^{bff}_{Real,Real,Real}$ is undefined and the rule is unsafe.

The final, optimized and binding pattern adorned definition of $\texttt{ftemp}_{Person}$ will look like:

$$\begin{aligned}\texttt{ftemp}^{bf}_{Person,Real}\texttt{(P,F)} \;\texttt{:-}\; & \texttt{ctemp}^{bf}_{Person,Real}\texttt{(P,\_V1)} \;\&\\ & \texttt{times}^{bbf}_{Real,Real,Real}\texttt{(9,\_V1,\_V2)} \;\&\\ & \texttt{times}^{fbb}_{Real,Real,Real}\texttt{(\_V3,5,\_V2)} \;\&\\ & \texttt{plus}^{bbf}_{Real,Real,Real}\texttt{(32,\_V3,F).}\end{aligned}$$

Our optimization here uses the bound-is-easier heuristic combined with trying at each step to choose only legal binding patterns for foreign predicates.

The following steps generate the above literal order:

First we place $\texttt{ctemp}^{bf}_{Person,Real}\texttt{(P,\_V1)}$
Now we have `P` and `_V1` bound, and we must place $\texttt{times}^{bbf}_{Real,Real,Real}\texttt{(9,\_V1,\_V2)}$ next.
Then we know `P`, `_V1`, and `_V2` which forces us to place $\texttt{times}^{fbb}_{Real,Real,Real}\texttt{(\_V3,5,\_V2)}$ next.
Finally we place $\texttt{plus}^{bbf}_{Real,Real,Real}\texttt{(32,\_V3,F)}$ last.

In summary, the need to optimize the foreign and derived functions is deemed very important in the context of OO queries. The crux of the problem is to integrate the optimization

of these functions in a seamless fashion to the optimization of the rest of the functions. One such proposal was used in the context of $\mathcal{LDL}$ [3], wherein the optimization of these computed functions was achieved by modeling them as 'infinite' relations, with certain finiteness constraints and the cost modeled using schematic information.

Here we use a simplified mechanism by which the user can specify different definitions of the foreign functions depending on their binding patterns. The system also allows a simple cost-based heuristic by allowing the user to specify different priorities on the different binding pattern adorned foreign predicate definitions. We have found our simplified foreign predicate scheme to be very effective.

6 Performance Measurements

To give an indication of the effects of the various optimizations, we have measured the execution times of some of our OSQL functions with and without optimizations. Our database contains 1000 `Person` objects and we have measured the CPU time to execute each derived OSQL function in the forward direction.[10] The measurements have been made for a main memory implementation of OSQL written in HP CommonLisp IV and run on an HP9000/370. Compile time overloading and foreign function reordering must be always active in order to produce executable Datalog expressions.

Function	*Optimization levels*			
	F	TC	JO	EQ
GrandSParentIncome	3.2ms	25ms	514s	3.2ms
ParentS	1.8ms	2.4ms	1.8ms	12s

F = Full optimization
TC = No type checking
JO = No join order optimization
EQ = No equality optimization

As expected, our measurements show that join order optimization is potentially the most

[10] We also made the test for 10000 objects, but the worst case execution speed then got prohibitive, while the best case remained approximately the same.

important one when object-oriented queries are specified declaratively. Even though we optimized our query system for very fast dynamic type checking, the effect of removing dynamic type checks gave one order of magnitude performance improvement for the relatively complicated function `GrandSParentIncome`. Finally, the use of equality substitution to simplify `ParentS` also made a one order of magnitude performance improvement.

7 Summary and Conclusions

We have described how to translate an object-oriented query language, OSQL, into corresponding optimized Datalog rules and facts. We described some important optimizations and ameliorations that are applied during the translation process.

Some optimizations that make use of OSQL constructs, such as function definitions and type checking, are applied as part of the translation into Datalog. Other optimizations transform the generated Datalog programs to make them executable and efficient.

We described some of the optimizations that are particularly important because of the object-oriented nature and usage of OSQL. Optimizations of importance from this point of view are static overload resolution, type check removal, foreign function removal, equality substitutions, and function optimization in the forward direction. Conventional relational and Datalog optimizations are also extremely important. We illustrated each type of amelioration and optimization with examples of how object-oriented OSQL queries are translated.

All algorithms are completely implemented in a main memory OSQL implementation, ADB. By including a simple top-down (SLR) interpreter for Datalog programs we are able to represent and manipulate full OSQL databases in ADB. The system uses maximally efficient representations of system data, and leverages its foreign predicate optimization technique to make system information transparently available at run time. The implementation techniques have allowed us to make a very compact system implementation (less than 2000 lines of CommonLisp code).

Future work would include extending the system to handle a larger class of recursive queries efficiently, for example by using techniques devised by NAIL! [14, 15] and $\mathcal{LDL}$ [11, 3, 4]. Special care will have to be taken to handle the duplicate semantics of OSQL ([12] describes some methods).

Other research directions would include the handling of late binding of overloaded functions, leading to the use of uninterpreted function symbols in generated Datalog programs [8]. Such techniques can also be used when constructing heterogeneous query languages where all database relations are not known at compile time [9].

Another interesting research area is to combine our main memory OSQL implementation with a disk-based one with primitives, to check out parts of the database to *work areas* represented in ADB. During long sessions the user would work against efficient ADB databases and only occasionally check data back into the central database.

It should be investigated if a formal description of the translation process from OSQL to logic combined with a formal semantics for the generated class of logic programs could serve as a method to formally define the semantics of OSQL.

We are working on extending the current implementation to handle aggregation and negation.

ACKNOWLEDGEMENTS:

Ravi Krishnamurthy helped me considerably improve earlier versions of this paper.

References

[1] D.Beech: A Foundation for Evolution from Relational to Object Databases, *Advances in Database Technology - EDBT '88*, Lecture Notes in Computer Science, Springer-Verlag, 1988, pp 251-270.

[2] S.Ceri, G.Pelagatti: *Distributed Databases Principles & Systems*, McGraw-Hill computer science series, 1984.

[3] D.Chimenti, R.Gamboa, R.Krishnamurthy: Towards an Open Architecture for $\mathcal{LDL}$, *Proc. 15th Intl. Conf. on Very Large Databases*, Amsterdam, the Netherlands, 1989, pp 195-204.

[4] D.Chimenti, R.Gamboa, R.Krishnamurthy, S.Naqvi, S.Tsur, C.Zaniolo: The $\mathcal{LDL}$ System Prototype, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990.

[5] T.Connors, P.Lyngbaek: Providing Uniform Access to Heterogeneous Information Bases. In Lecture Notes in Computer Science 334, *Advances in Object-Oriented Database Systems*, K.R.Dittrich, Ed., Springer-Verlag, Sept. 1988.

[6] D.Fishman et al.: Overview of the Iris DBMS, in W.Kim, F.H.Lochovsky (eds.): *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, 1989.

[7] S.E. Keene: *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

[8] R.Krishnamurthy, S.Naqvi: Towards a Real Horn Clause Language, *Proc. 14th Intl. Conf. on Very Large Databases*, Los Angeles, CA, 1988, pp 252-263.

[9] R.Krishnamurthy, W.Litwin, B.Kent: *Language Features for Interoperability of Databases with Semantic Discrepancies*, Technical memo HPL-DTD-90-14, DTD, HP Laboratories, 1501 Page Mill Rd., Palo Alto, CA 94304, 1991.

[10] W.Leler: *Constraint Programming Languages: Their Specification and Generation*, Addison-Wesley, 1988

[11] S.Naqvi, S.Tsur: *A Logical Language for Data and Knowledge Bases*, Computer Science Press, 1989.

[12] I.Mumik, S.Finkelstein, H.Pirahesh, R.Ramakrishnan: Magic is relevant, *Proc. SIGMOD 1990*, Atlantic City, NJ, 1990, pp 247-258.

[13] D.Shipman: The Functional Data Model and the Data Language DAPLEX, *ACM TODS*, 6(1), pp 140-173, March 1981.

[14] J.D.Ullman: *Principles of Database and Knowledge-Base Systems*, Volume I, Computer Science Press, 1988.

[15] J.D.Ullman: *Principles of Database and Knowledge-Base Systems*, Volume II, Computer Science Press, 1989.