OFL: A Functional Execution Model[1] for Object Query Languages

Georges Gardarin[†‡] Fernando Machuca[†*] Philippe Pucheral[†]

([†]) PRISM Laboratory
University of Versailles/Saint-Quentin
45, Avenue des Etats-Unis
78035 VERSAILLES Cedex
France

([‡]) Projet Rodin
INRIA
Rocquencourt
France

([*]) EDS International (France) SA
Le Guillaumet
060 Avenue du président Wilson
92800 Puteaux

email: <firstname>.<lastname>@prism.uvsq.fr

Abstract

We present a functional paradigm for querying efficiently abstract collections of complex objects. Abstract collections are used to model class extents, multivalued attributes as well as indexes or hashing tables. Our paradigm includes a functional language called OFL (Object Functional Language) and a supporting execution model based on graph traversals. OFL is able to support any complex object algebra with recursion as macros. It is an appropriate target language for OQL-like query compilers. The execution model provides various strategies including set-oriented and pipelined traversals. OFL has been implemented on top of an object manager. Measures of a typical query extracted from a geographical benchmark show the value of hybrid strategies integrating pipelined and set-oriented evaluations. They also show the potential of function result memorization, a typical optimization approach known as "Memoization" [2] in functional languages.

Keywords

Object retrieval, functional language, query execution strategy, performance measurement.

1. Introduction

The techniques proposed for executing object-oriented queries often use a traditional algebraic approach extending relational operators with object-oriented concepts [Beeri90, Shaw90, Straube90, Cluet92, Steenhagen94]. Algebraic trees are the target of the query compiler, which annotates them with physical algorithms and determines the most appropriate tree using some heuristic [Valduriez91, Lanzelotte93, Finance94]. Unfortunately, algebraic approaches are not very appropriate for expressing path expressions which are the most expected mechanism used to query object-oriented databases [Carey88, Gardarin92a, Kim92, Frohn94]. Other query evaluation techniques have been proposed such as low level navigation through reference handling [Delobel91, Gardarin92b]. Indeed, special indexes called access support relations were tailored for evaluating path expressions in a rule-based object query optimizer [Kemper90]. Similarly, a more general approach is presented in [Orenstein92]. They show that join optimization in object-oriented query processing is a much simpler problem because i) joins between foreign and primary keys are replaced by paths or precomputed joins; and ii) indexes may be created on whole paths. But, in some cases determined by cluster policies [Valduriez86], path expressions should be evaluated by explicit join operations [Jenq90, Shekita90, Blakeley93]. Therefore, an execution model integrating set-oriented and pipelined (i.e., one object at a time) operations should clearly emerge.

This paper proposes an intermediate functional language called OFL (Object Functional Language) as a target language for object-oriented query compilers. OFL is independent of any data model and any storage model. In addition, the language comes with a flexible execution model supporting both set-oriented and pipelined strategies. The aim of OFL is to provide a unified and optimized execution support for the different object-oriented query languages currently developed. More precisely, in the context of the IMPRESS Esprit Project, OFL is used as the target language of an OQL compiler [Cattell93]. More generally, we believe that OFL could be the basis of an abstract machine built on top of different object servers or database systems. This abstract machine will provide the intermediate language for compilers of database languages addressing a large class of advanced applications, with new requirements in terms of computing power (e.g., recursion) and modeling (e.g., complex objects).

Database languages can be used as intermediate languages. FQL [Buneman79], Daplex [Shipman81] and GENESIS [Batory88] are data model and manipulation languages for database systems grounded in the functional data model. Like OFL, they permit efficient implementations because functional expressions can be easily mapped to a parallel, multiprocess or multi-threaded process environment [Batory88]. The goal of OFL is to benefit from the functional data model, but as a pure intermediate language satisfying the compiler principles of high quality intermediate languages indicated in [Aho88]. A program of an intermediate language is an intermediate representation in the compiler process that should be easy to generate, easy to optimize, and easy to translate into the target code. Generating FQL, DAPLEX or GENESIS programs from OQL-like query compilers seems to be inappropriate. The problems include: data model mapping, difficult translation from abstract intermediate representation, and the absence in these languages of an associated efficient execution model integrating set-oriented and pipelined operations. OFL could be seen as an FQL-like or DAPLEX-like or GENESIS-like target intermediate language for a query compiler. That is, it is also grounded in the functional model but

[1] This work is partially funded by Esprit project IMPRESS n° 6355.

[2] Memoization is a term introduced in [Michie68] to denote the memorization of Memo functions.

deals with abstract collections to model the execution world of any OQL-like query compiler.

OFL is founded on the same principles as FAD [Bancilhon87]. FAD uses low level operators and operator constructors generalizing the relational algebra and providing the opportunity of dataflow in a parallel architecture. OFL gives quantifier functions that are constructors which permit the building of FAD low level operators [Machuca94]. Like FAD, OFL is an untyped language offering an efficient support of iteration, conditional, and set operations. One of the strongest points of OFL is the explicit existence of abstract collections. That is: a more general concept than set in FAD, but which satisfies the same parallel principle (all elements in a set can be processed in parallel) [Bancilhon87]. In conclusion, the fundamental difference between OFL and FAD is that FAD is a complete database language and OFL is a pure intermediate language with an efficient execution model that can be more easily integrated in OQL-like query compiler processes.

OFL deals with abstract collections of objects that may model class extents, multivalued attributes as well as indexes or hashing tables. Functions constitute the unique interface of abstract collections whose structures are completely hidden. Thus, the language is general enough to deal with any kind of collection that is usual in object-oriented systems, and it is independent of the storage model selected for the collections. OFL provides basic functions to traverse abstract collections and to evaluate predicates and projections on them. These basic functions are powerful enough to express different execution strategies. In addition, they may be used as building blocks to define ad-hoc operators for dedicated query languages, which are always needed to fulfil the requirements of specific database applications [Cruz88, Ammann92]. We show in [Machuca94] the construction as OFL macros of the Encore complex object algebra [Shaw90], the general naive recursive operator [Bancilhon86], and the general transitive closure recursive operator [Dar91].

OFL lends itself to a functional execution model. The major contribution of the functional execution model is to model the execution of functional programs as graph traversal algorithms. Set-oriented, pipelined or hybrid execution strategies can be generated for the same query, depending on the selected graph traversal strategy. We show how OFL and its supporting execution model may be integrated in OQL-like query optimizers. Queries of a typical geographical benchmark have been implemented in OFL. Measures show that set-oriented or pipelined strategies are appropriate for different types of queries and databases. Moreover, the results illustrate the importance of memorizing the intermediate results, mainly for pipelined strategies.

The remainder of this paper is organized as follows. Section 2 introduces abstract collections as a way to unify the management of various forms of complex object collections. Section 3 presents the OFL object functional language proposed to manipulate abstract collections. Section 4 describes the execution model supporting OFL. We show that this execution model supports both set-oriented and pipelined evaluations of complex queries. Section 5 focuses on the implementation of the execution model. It also summarizes our experiences and gives real measures comparing graph traversal strategies. Section 6 addresses the integration of OFL and its execution model in OQL-like query compiler processes. Finally, Section 7 concludes the paper by indicating directions for further research.

2. Abstract Collections

A *collection* is an object that groups other objects. A collection acts as a container of objects of the same type. In relational systems, relations are the unique collections a user can query. In object-oriented systems, a collection can model a class extent (i.e., a container holding all the class instances) as well as a multivalued attribute like a set, a list or an array. Thus, collection instances can themselves contain collections, thereby making the query execution process tricky. Object-oriented query compilers must face the diversity of collection definitions. While the user query involves only collections containing user objects, system collections like indexes and temporary results can also participate in the query evaluation. The intermediate language proposed in this paper deals with both user collections and system collections.

We introduce the notion of *abstract collection* as a unified way for manipulating any kind of collection of objects, independently of the type of the collection instances and of the storage organization of the collection. In OFL, an abstract collection is seen as a container of objects encapsulated by two specific sets of functions. The *behavioural functions* model the external collection definition. The behavioural functions correspond to all methods attached to the collection type. In particular, functions to read and update the object attributes and functions to create or remove objects are considered as behavioural functions. It is possible to iterate over the elements of a collection. In OQL, iteration is done by defining an iterator that maintains a current position within the collection to be traversed. With OFL, which manipulates only abstract collections, iteration functions and iterators are attached to abstract collections as *traversal functions*. The traversal functions encapsulate the storage organization of the collection.

Definition 1: Abstract Collection
A container of objects encapsulated by a finite set of behavioural and traversal functions.

The generality of OFL relies on a standardized interface for the traversal functions of all collections. Three traversal functions are mandatory to manipulate a collection in the query execution process, namely *First*, *Next* and *Get*. As shown in Section 5, these three functions are invoked by the functional language evaluator to iterate on any abstract collection. First and Next permit to scan a collection and deliver a reference to the requested collection instance. Get delivers the value of the current collection instance. The distinction between the value of a collection instance and a reference to it is useful to materialize temporary results in different ways. For example, it has been demonstrated in [Bitton86, Lehman86] that main memory space and CPU time can be saved by materializing temporary results by means of references pointing to the selected objects instead of copying the values of these objects.

Indexes are essential in all optimization strategies [Selinger87, Kim89, Kemper90, Orenstein92, Lanzelotte93]. Indexes are integrated in our functional language as common abstract collections. Indeed, conventional indexes may be seen as collections where an object or a set of objects is related to an atomic value [Kemper90]. In object-oriented databases, path indexes may be seen as collections providing a direct backward link [Cluet92]; or collections supporting reference chains from one object instance to another instance object or to a collection of instances objects [Kemper90], in a way similar to multivalued attributes in user collections. Thus, these collections may be scanned using the aforementioned traversal functions. Specific behavioural functions are defined on indexes like Find(Index, keyValue), which delivers the object identifiers referencing all objects of the indexed collection sharing the same key value; or Getasr(ASR, REST, SELPRED, PROJ) which retrieves objects (projected onto the attributes in the PROJ list) from an access

support relation ASR, for which REST and SELPRED predicates are satisfied [Kemper90].

By encapsulating abstract collections with behavioural and traversal functions, class extents, multivalued attributes, temporary results, and indexes can be treated in the same way by our intermediate functional language. Behavioural and traversal functions may be *monovalued* or *multivalued*. The *monovalued* functions map one or many abstract objects to a single abstract object. In contrast, the *multivalued* functions return a collection of abstract objects. Functions may apply to a single object within a collection or to the whole collection (e.g., aggregate). We introduce now some notations that will be used in the sequel of the paper. We denote *C* the universal set of collections and *F* the universal set of functions. For an abstract collection *Ci*, we denote its associated finite set of functions *FCi*. We denote *Ci.fj* the jth function of FCi. Figure 2.1 summarizes these notations.

FCi = { Ci.f1, Ci.f2, ... ,Ci.fn }
monovalued Ci.fj : Ci X ... ===> Ck
multivalued Ci.fn : Ci X ... ===>> Ck

$C = \bigcup_{i=1}^{\infty} Ci$ $\quad F = \bigcup_{i=1}^{\infty} FCi$

Figure 2.1: Summary of notations

3. OFL: A Functional Language for Abstract Collections

OFL is based on a functional approach. Functions map collections of objects to collections of objects. Functions can be used as building blocks to define algebraic operators (e.g., a join operator) as well as to express a complete query execution plan. Functions are general enough to express collection traversals, to evaluate unary, binary and n-ary predicates on the current instances of all collections involved in a query, and to apply simple or complex projections on the selected instances. To ease the mapping between the external object query language and our intermediate language, the proposed language-constructs permit the mixing of imperative and functional programming features. We first introduce the elementary function constructors, then we propose quantifier functions to express complex iterations on collections.

3.1 Elementary Function Constructors

As usual with functional programming [Bakus81], functions on abstract collections can be composed together. They can also be conditionally applied. Successive compositions yield functional expressions generally referred to as path expressions in object-oriented databases.

Definition 2: Composition
A function forming operation that maps two functions f and g into a single one as follows: f.g (x) = f(g(x)).

The definition of conditionals requires that of predicates. A *predicate* is simply a function that returns a Boolean.

Definition 3: Conditional
A function forming operation that maps a predicate p and two functions f and g into a single one as follows:

If_Then_Else (p, f, g) (x) $= f(x)$ *if p is True*
$= g(x)$ *if p is False.*

Definition 4: Path Expression
A finite sequence of function compositions of the form $f_0.f_1 \cdots f_n$.

The function f_0 is the beginning of the path and f_n its end. If f_i, f_{i+1} are in the path, the abstract collection defining the first argument of f_i must be the abstract collection describing the result of f_{i+1}. A problem is the specification of path expressions mixing monovalued and multivalued functions. We use intermediate functions to apply monovalued functions to multivalued results. These intermediate functions are proposed in the functional language as second order functions called quantifier functions.

3.2 Quantifier Functions

The goal of quantifier functions is to provide tools to apply a function to all objects of a collection or to any object of a collection. It gives capabilities to construct complex collection path expressions. Quantifier functions are devoted to the traversal of the instances of one collection. They have three arguments. The first argument is the collection to be traversed. The second argument is a predicate (Boolean function) indicating if the current instance has to be considered in the operation. The third argument is the function to apply when the evaluation of the predicate argument is true. OFL offers two quantifier functions ForAll and ForAny defined as follows.

Definition 5: Apply to all
A second order function of signature ForAll(C, p, f) that applies a function f to all objects of a collection C satisfying a predicate p.

Definition 6: Apply to any
A second order function of signature ForAny(C, p, f) that applies a function f to any object of a collection C satisfying a predicate p.

Quantifier functions apply functions to collections. Within a collection, objects are processed one at a time. To qualify the current instance of a collection Ci that is processed internally to a ForAll or ForAny function, we use the function **Current**(Ci).

To illustrate, let us assume the functional ODL-like schema presented in Figure 3.1. Methods and attributes become monovalued or multivalued functions. This ODL schema describes persons owning vehicles composed of parts; lastname, color and partlabel are string valued attributes; salary, birth year, model and price are integer valued attributes; owner and composed are set-valued attributes represented by double arrows in the figure. Figure 3.2 illustrates a possible functional translation of two different OQL queries. Throughout this paper, we will note functions in abstract collections with capital letters to differentiate with methods and attributes in the ODL-like schema. We adopted an easy functional notation [Field88] commonly used in programming languages. The first query selects the names of all persons whose first name is Bill while the second query selects the name of people having at least one red vehicle. We named **Person'** the abstract collection representing the Person class extents. For clarity, all the query examples given in this paper are expressed with an OQL syntax. However, our intermediate functional language is independent of any object query language.

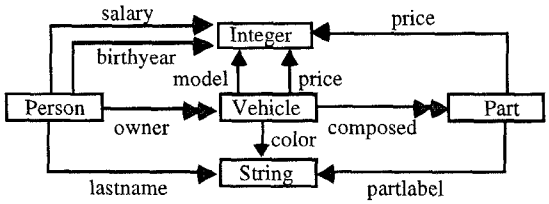

Figure 3.1: An example of a functional ODL-like schema

```
SELECT      p.lastname
FROM        p in Person
WHERE       p.firstname = "Bill"

gives:

ForAll(Person',
   StringEqual(FirstName(Current(Person')),"Bill"),
   Display(LastName(Current(Person'))))
SELECT      p.lastname
FROM        p in Person
WHERE       exists v in p.owner : v.color = "Red"

gives:
ForAll(Person',null,
  ForAny (Owner(Current(Person')),
   StringEqual(
     Color(Current(Owner(Current(Person')))),"Red"),
   Display(LastName(Current(Person')))))
```

Figure 3.2: Example of the ForAll and ForAny operations

3.3 Iteration and Sequence Functions

To obtain the power of a complete functional language, we add to OFL capabilities to iterate on a function using a while construct. As the language is purely functional, the iteration is also defined as a function as follows.

Definition 7: Iteration
A second order function of signature While (p,f) that applies a function f repeatedly until p becomes false.

Furthermore, as usual with functional programming, we add a sequence construct to sequentially apply functions.

Definition 8: Sequence
A function forming operation of the form Sequence(f1, f2, ..., fn) whose result is the list of results obtained by applying fi's in the given order.

3.4 Traversing Multiple Collections

The major contribution of quantifier functions is to provide functionalities to traverse multiple collections. Thus, quantifier functions can be nested to process path expressions and complex algebraic operations. The problem is then the retrieval of instances in each collection to assemble the objects instancing a path. Predicates and projections are applied to the Cartesian product of the collection instances to generate operation results. A ForAll function having as third argument a ForAll function gives an implementation of the binary algebraic join operation. The predicate and projection functions may apply to the current instances of the two collections. More generally, when traversing multiple collections, the functional language evaluator keeps track of a current instance for each collection. The predicate and projection functions may apply to all the current instances. Figure 3.3 gives a representation of an instanced path in a multiple collection traversal.

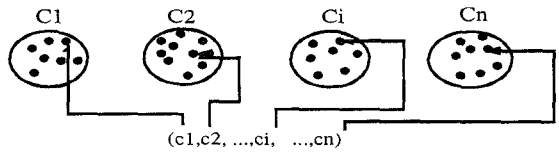

Figure 3.3: An instanced path in a multiple collection traversal

To make multiple collection traversals easier to express, we use a **DefVar** function to explicitely name abstract collection variables. For example, DefVar(JohnVehicle, Owner(John)) defines the abstract collection JohnVehicle which can be used anywhere in a functional program to reference the collection formed by the set-valued attribute John.owner. In some sense, the concept of collection variable makes explicit the concept of iterator in a functional paradigm.

Figure 3.4 illustrates a traversal of multiple collections using quantifier functions and collection variable definitions. The functional program given in this figure is one possible translation of the given complex OQL query expressed on the ODL schema presented in Figure 3.1. For this query, **Person'**, **Vehicle'** and **Part'** abstract collection variables are used. Person' represents the Person class extents. Vehicle' refers to the collection p.owner traversed for each current instance of Person'. In addition, a collection variable Part' is defined as the collection v.composed traversed for each current instance of Vehicle'.

```
SELECT      p.firstname, p.lastname, v.model, v.color, c.price
FROM        p in Person, v in p.Owner
WHERE       exists c in v composed : c.price > 20000
            and v.model > p.birthyear and
            p.salary < c.price

gives:

ForAll(Person',null,
  ForAll(DefVar(Vehicle',Owner(Current(Person'))),
    GreaterInteger(Model(Current(Vehicle')),
                   BirthYear(Current(Person'))),
    ForAny(DefVar(Part',Composed(Current(Vehicle'))),
      And(GreaterInteger(Price(Current(Part')),20000),
          LessInteger(Salary(Current(Person')),
                      Price(Current(Part')))),
      Sequence(Display(FirstName(Current(Person'))),
        Display(LastName(Current(Person'))),
        Display(Model(Current(Vehicle'))),
        Display(Color(Current(Vehicle'))),
        Display(Price(Current(Part'))))))
```

Figure 3.4: Example of a functional expression evaluating a complex query

4. Functional Expression Execution Model

The execution model supports execution of functional expressions. OFL programs are functional expressions describing abstract collection traversals. In the traversal, predicate and projection functional expressions are applied. In this section, we propose a general graph-based formalism to ease the representation of OFL programs. In particular, set-oriented and pipelined

programs can be obtained based on two different graph traversal strategies. We describe the integration of this graph-based formalism in OQL-like query optimization processes in Section 6.

Functions in the execution model have non typed arguments. They have as parameters abstract values referred to by pointers. They return an abstract value as a pointer. Correct typing of functional expressions is checked at compile time, when generating the functional expression. The execution model uses execution units to describe the non-typed functions.

4.1 Execution Units

Introducing execution units permits the description of non-typed function executions. An execution unit is a memory structure specified by a pointer containing execution information.

Definition 9: Execution unit
A structure describing either a function or a value to support execution of functional expressions.

The execution unit contains all information required to run a function. It includes the function name giving the code address and a pointer to the parameter list. The input parameters are described in turn by execution units. Execution units describe output parameters useful for memorizing result.

The description of an operation is memorized in a direct acyclic graph as shown in Figure 4.1. The figure represents the DAG for the functional expression f1(f3(c1,c3),f2(c2)). Note that an execution unit may refer directly two others: its first parameter and the unit representing the expression next to it. Abstract value lengths are included in value execution units.

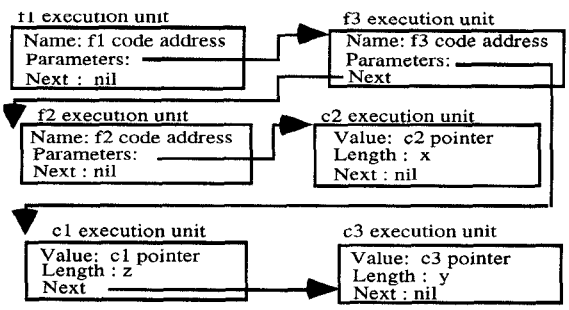

Figure 4.1: An abstract operation as a graph of execution units.

4.2 The Collection Traversal Graph

The collection traversal graph represents the traversal of abstract collections performed when executing a functional expression. We integrate in a single graph all the handy information allowing different optimization strategies. As an index is also an abstract collection (system collection), the index traversals are integrated in the graph.

Definition 10: Collection Traversal Graph
A directed labelled graph $G=(V, E, L_V, L_E)$ where each vertex $v \in V$ represents an abstract collection, each arc $(vi,vj) \in E$ represents a function (execution unit) from vi to vj and where $lv(vi) \in L_V$ is the name of the corresponding collection and $le(vi,vj) \in L_E$ is the name of the corresponding function.

The vertices are labelled by names representing collections accessed during execution. The arcs are labelled by execution units representing the functions at execution. For example, if Ci and Cj are abstract collections and fci.k is a function of FCi mapping Ci into Cj, then (Ci,Cj) is an arc of the graph, labelled by the function execution unit fci.k. For simplicity, we note the function execution unit as the function name.

Labelling the arcs by execution units shows in the graph only the relevant information for query evaluation. Execution units are abstractions of functions at execution storing all the necessary information for their executions. When a query refers n times to the same collection, the collection is represented by n different vertices in the graph. A function appearing in n different function execution units appears in n different labelled arcs. For convenience, we distinguish more precisely monovalued and multivalued function on the graph by marking with a double arrow the target of a multivalued function.

For completeness, the graph representation should also support conditional, apply to all, apply to any, and iteration functions. To represent these second order constructs, we simply introduce composed labels on the graph. A composed label includes two function execution units. The first one is chosen among if <predicate>, while <predicate>, ForAll, and ForAny; it simply gives the control function name. The predicates, and the projections for quantifier functions are name parameters that can refer to other collection traversal graphs for complete evaluation. The second function is the output function of the control structure, i.e., the *then* part for an *if* (*if then else* are not fully integrated, but represented by two *if then* constructs), the function to execute for a while and the applied function for quantifier functions.

4.3 An Application Example

This complete example integrates an OQL user query expressed on the schema given in Figure 3.1, a functional expression evaluating the query, and the collection traversal graph describing the query execution. The OQL user query and the functional evaluation expressions are shown in Figure 4.2. The query retrieves the last names of sixteen year old persons and the colors and part labels of the vehicles they own.

```
SELECT   p.lastname, v.color, c.partlabel
FROM     p in Person, v in p.owner, c in v.composed
WHERE    p.age = 16 and v.price=c.price

ForAll(DefVar(Person16,Find(AgeIndex,16)),null,
  ForAll(DefVar(Vehicle16,Owner(Current(Person16))),
         null,
  ForAll(DefVar(Part16,Composed(Current(Vehicle16))),
    IntegerEqual(Price(Current(Vehicle16)),
    Price(Current(Part16))),
    Sequence(Display(LastName(Current(Person16))),
             Display(Color(Current(Vehicle16))),
             Display(PartLabel(Current(Part16)))))))
```

Figure 4.2: A query example with functional expression.

Figure 4.3 shows a possible adorned collection traversal graph indicating the evaluation of the functional expression given in Figure 4.2. The collection traversal graph is adorned with the predicate and the projection collection traversal graphs of the quantifier function ForAll3. Functions with multiple arguments are represented by a list of ordered input arcs. They are labelled by the same function execution unit based on the notation presented in the graphic representation of the typed algebra [Cluet92]. A target node (black filled) in the collection traversal graph is used to indicate the end of the traversal. It happens when the second label

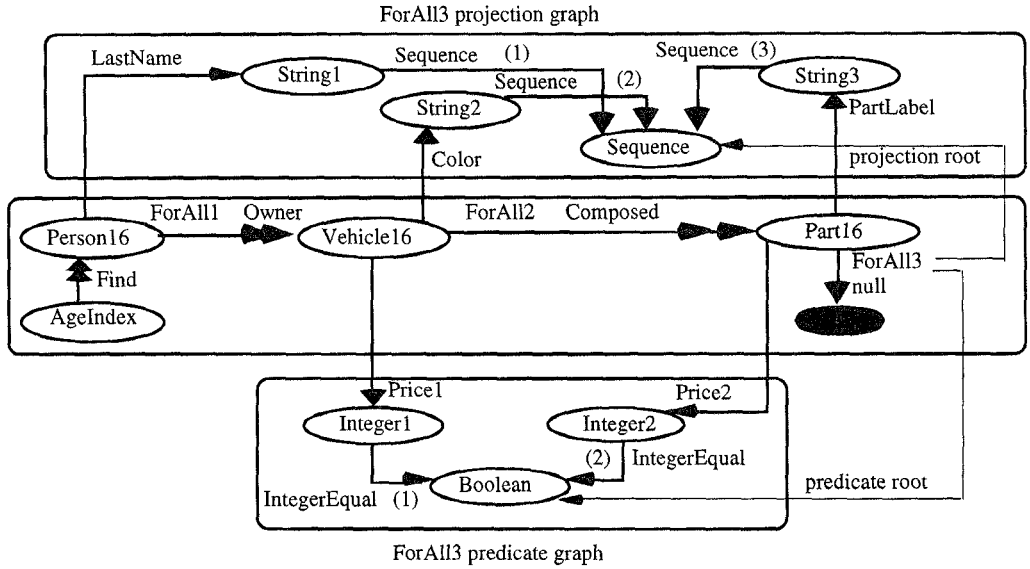

Figure 4.3: An adorned collection traversal graph for the query of Fig. 4.2.

of a quantifier composed label is null. AgeIndex, Person16,Vehicle16 and Part16 are the abstract collections of the collection traversal graph. Sequence, String1, String2, String3, Integer1, Integer2 and Boolean are abstract collections in the predicate and projection collection traversal graphs of ForAll3 execution unit. The AgeIndex collection is used to model an index traversal giving fast access to instances of the Person class of a given age. The function execution unit Find represents the search index function that gives as result a set of person having a specific age. Further, Find, Owner and Composed are execution units representing multivalued functions. ForAll1, ForAll2, and ForAll3 are execution units describing quantifier functions. ForAll1 and ForAll2 execution units have neither predicate nor projection graphs. The graph describing the projection of the ForAll3 function has as root node the Sequence abstract collection. In addition, the predicate collection graph of ForAll3 has as root node the Boolean abstract collection. Finally, Price1 and Price2 execution units describe the v.price function and the p.price functions respectively.

4.4 Paths in Collection Traversal Graphs

A path expression or simply a path in the collection traversal graph is a finite sequence of function execution units, $EU_0.EU_1. \ldots .EU_n$ such that EU_i is connected to EU_{i+1}. The execution unit EU_0 is the *beginning* of the path and EU_n its *end*. If EU_i and EU_{i+1} are in the path, the abstract collection defining the first argument of the function represented by the EU_{i+1} unit must be the same as the abstract collection describing the result of the function represented by the EU_i unit.

A path **p** has its corresponding collection path. The collection path **cp** is an ordered list $C_0.C_1. \ldots .C_m$ of abstract collection execution units. If EU_i is an execution unit in p andthe function describing by this execution unit, applies on collection C_i to collection C_k; C_i and C_k are elements of **cp**. The abstract collection execution unit C_0 is the *collection beginning* of the collection path and C_m its *collection end*. Thus, if p has n elements, **cp** has n+1 elements. A path is *complex* if it contains an EU_i representing a multivalued function. Otherwise, it is *simple*. The complexity of a path is characterized by the number of multivalued functions in the path. It is called the *degree* of complexity of the path.

4.5 Graph Traversal Strategies

The collection traversal graph represents a functional expression, which derives from a query. Several strategies can be applied to traverse complex paths. They correspond to different query processing approaches. The strategy chosen may drastically change the time to process a query. First, we discuss three traversal strategies for evaluating a complex path on the collection traversal graph. The strategies are illustrated on the graph given in Figure 4.6 representing a tree query for the generic OQL query given in Figure 4.4 on the ODL schema of Figure 3.1. Cz', Cr', Ck', and Ch' are the abstract collections refering to the collections (e.g., Cz, c1.fi, c2.fj, c3.fm) described in the FROM clause of the generic OQL query in Figure 4.5. Remark that the pi predicate and projection functions are hidden in the graph because they are described in ForAll function execution units.

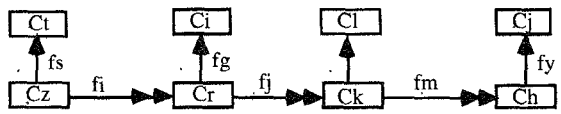

Figure 4.4: A generic functional ODL schema.

SELECT	c3.fp, c4.fy
FROM	c1 in Cz, c2 in c1.fi, c3 in c2.fj, c4 in c3.fm
WHERE	p1(c1) and p2(c1,c2) and p3(c1,c2,c3) and p4(c1,c2,c3,c4)

Figure 4.5: A generic OQL user query

Figure 4.6: A generic collection traversal graph.

Set-oriented Strategy

The goal of this strategy is the decomposition of a long path p in shorter paths denoted p1,p2, ...,pn. The evaluation of the long path p is replaced by the evaluation of the p1,p2, ...,pn adjacent paths. Each pi evaluation produces an intermediate collection that is used as input of next pj paths. When all the pi path lengths are less or equal to 2, each pi can be replaced by an algebraic collection operation. Then, the evaluation method is identical to the classic binary algebraic query decomposition strategy. Such a decomposition is not always very efficient even though it is the only approach implemented in several systems. The application of this strategy gives annotated OFL programs as shown in figure 4.7. In the OFL program, OFL quantifier functions are annotated with the result collections. Intermediate collections (e.g., T1, T2, T3) are tuple collections as in ENCORE algebra [Shaw90]. Insert functions have a standard functional signature Insert(collection,function). This facilitates the preprocess algorithm expanding annotations into OFL statements [Machuca94]. For tuple collections, function is always the tuple constructor function. A generic GetAtt function — GetAtt(tuple object, attribute position) — is used in the example.

```
The long path p:      Cz'.fi.Cr'.fj.Ck'.fm
The collection path cp:   Cz'.Cr'.Ck'.Ch'

The set-oriented strategy:

a classic preliminary selection on Cz' generates T1

p1=Cz'.fi
cp1=T1.Cr'      the evaluation of p1 generates T2

p2=Cz'.fi.Cr'.fj
cp2=T2.Ck'      the evaluation of p2 generates T3

p3=Cz'.fi.Cr'.fj.Ck'.fm
cp3=T3.Ch'
the evaluation of p3 generates the query result

The annoted OFL program:

ForAll(
  ForAll(
   ForAll(ForAll(Cz',p1,Tuple(Current(Cz'))) [T1],
     null,
     ForAll(DefVar(Cr',Cz'.fi(GetAtt(Current(T1),1))),
       p2,
       Tuple(GetAtt(Current(T1),1),Current(Cr')))) [T2],
   null,
   ForAll(DefVar(Ck',Cr'.fj(GetAtt(Current(T2),2)))),
     p3,
     Tuple(GetAtt(Current(T2),1),
          GetAtt(Current(T2),2),Current(Ck')))) [T3],
   null,
   ForAll(DefVar(Ch',Ck'.fm(GetAtt(Current(T3),3))),
     p4,
     Sequence(Display(Ck'.fp(GetAtt(Current(T3),3)),
                Display(Ch'.fy(Current(Ch')))))))
```

Figure 4.7: Set-oriented evaluation of the generic query.

Pipelined Strategy

The objective of the pipelined strategy is the evaluation of the long path p directly, from one object of the source collection up to generate an object of the output collection. The advantages of this method is twofold [Pucheral92]. First, temporary collections are no longer needed. This saves main memory space and CPU time to build them. Second, the first objects in the result can be produced and delivered to the user while the query is still being processed. This greatly reduces elapsed time. Due to this, pipelined processing is considered as the most efficient strategy for main memory database systems [Amman85]. A possible program for pipelined evaluation of the generic query (see figure 4.5 and 4.6) is sketched in figure 4.8.

```
The long path p:      Cz'.fi.Cr'.fj.Ck'.fm
The collection path cp:   Cz'.Cr'.Ck'.Ch'

The pipelined strategy:

p = Cz'.fi.Cr'.fj.Ck'.fm
cp = Cz'.Cr'.Ck'.Ch'

The OFL program:

ForAll(Cz',p1,
   ForAll(DefVar(Cr',Cz'.fi(Current(Cz'))),p2,
     ForAll(DefVar(Ck',Cr'.fj(Current(Cr'))),p3,
       ForAll(DefVar(Ch',Ck'.fm(Current(Ck'))),p4,
         Sequence(Display(Ck'.fp(Current(Ck'))),
                  Display(Ch'.fy(Current(Ch')))))))
```

Figure 4.8: Pipelined evaluation of the generic query.

Hybrid Strategy

One more general strategy is a hybrid of set-oriented and pipelined strategies. Set-oriented strategies are efficient when intermediate results are small while pipelined strategies are best for large collections with indexes. When a function applies to all the instances of a collection (e.g., aggregate functions), pipelined strategies are not possible. Thus, hybrid strategies mixing set-oriented and pipelined strategies are required. As another example, it is important to integrate fast hashed join algorithms to process joining functions. Hashed joins generate temporary collections and require a set-oriented strategy. In contrast, an algebraic binary decomposition may be expensive if the cost of the generation and traversal of intermediate results is high. Often, an optimized strategy must merge set-oriented and pipelined strategies for different part of a long path. It is important to determine the best hybrid strategy to optimize a given query. To describe more precisely an execution plan, we propose in our implementation a simple language representing the programs generated by set-oriented, pipelined or hybrid strategies.

5. Implementation and Performance Measurement

The evaluator of OFL has been implemented in the IMPRESS Esprit project on top of the IMPRESS object manager. We first give an overview of the language grammar and the language evaluator and show that this evaluator is independent of any data model and any storage model. Then we concentrate on real experiments and performance measures.

5.1 The Functional Language Evaluator

The major goal of the language is to describe the target OFL programs generated by set-oriented, pipelined and hybrid strategies. OFL programs are programs for a virtual machine implemented on an object manager.

The grammar of the language G = (Vn, Vt, P, S) is simple. The starting non terminal symbol (S) is Program. The set of non terminal (Vn) and terminal (Vt) symbols are presented in Figure 5.1. The set of production rules (P) is given in Figure 5.2.

```
Vn (non terminal symbols) =
    {Program,FunctionExecutionUnit,
     ParameterList,Parameter,FunctionName,
     AbstractValueExecutionUnit}

Vt (terminal symbols) =
    {ForAll, ForAny,While,If-Then-Else,Sequence,pointer}
```

Figure 5.1: Terminal and nonterminal grammar sets

```
<Program> ::=  <FunctionExecutionUnit>
<FunctionExecutionUnit> := <FunctionName> (<Parameter List>)
<ParameterList> := <Parameter> / <Parameter><Parameter List>
<Parameter> :=
<FunctionExecutionUnit>/ <AbstractValueExecutionUnit>
<FunctionName> := ForAll / ForAny / While
                  / If-Then-Else / Sequence / pointer
<AbstractValueExecutionUnit> :=  pointer
```

Figure 5.2: The set of production rules

The Semantics of the Language

This language describes function calls. Functions are non-typed; they represent operations on abstract collections. Indeed, a function parameter may be a value or a function. Values are called abstract values because their types are unknown from the language evaluator. An abstract value may be an instance of an abstract collection, an attribute of this instance or a function result. This makes the language evaluator independent of any data model. Control functions — ForAll, ForAny, While, If-Then-Else, and Sequence — have special evaluations.

The Execution Algorithm

A preprocessor expands annotations in annotated OFL programs into OFL statements [Machuca94]. The OFL execution algorithm is a hybrid of functional and imperative execution algorithms. The OFL algorithm can not use a pure functional execution because function calls can correspond to functions with side effects. In fact, functions are written in any programming language. However, the algorithm permits the two classical ***Eager*** and ***Lazy*** functional evaluation modes [Field88]. When the functional evaluation mode is eager, the evaluation of the function call is performed in two steps. First, we evaluate its parameters and then we invoke the function with the evaluated parameters. When lazy execution mode, the function call is invoked with the not yet evaluated parameters. The first version of the evaluator is naturally recursive since function parameters may be themselves functions; but for performance purposes a stack-based implementation like FAD [Bancilhon87] should be adopted. Quantifier functions (e.g., ForAll, ForAny) have a particular evaluation. The evaluation of the quantifier functions in turn invokes the traversal functions attached to the abstract collections to iterate on them. This makes the language evaluator independent of any storage model.

```
Executor (Program)
 switch ( execution unit type)
  AbstractValue : return ( Program )
  Function :
   switch ( function type )
    ForAll : return ForAllEvaluation( parameter list )
    ForAny : return ForAnyEvaluation( parameter list )
    Function :
        return FunctionEvaluation( parameter list )
        ...

ForAllEvaluation (parameter list)
  abstract collection = Executor(first parameter)
  Executor(traversal function First of the abstract collection)
  while (Current(abstract collection) ≠ end of collection)
   if (Executor(second parameter) is true)
      Executor(third parameter)
   Executor(traversal function Next of the abstract collection)

ForAnyEvaluation (parameter list)
  abstract collection = Executor(first parameter)
  Executor(traversal function First of the abstract collection)
  while (Current(abstract collection)
        ≠ end of collection and
       Executor(second parameter) is false)
     Executor(traversal function Next of the abstract collection)
  if (Current(abstract collection) ≠ end of collection)
      Executor(third parameter)

FunctionEvaluation (function execution unit )
 if function result has been memorized
   return function execution unit memorized result value
 else
   if function mode evaluation is EagerEvaluation
     function invocation (
       Executor(first parameter),
       Executor(second parameter),...)
   else ( Lazy Evaluation )
     function invocation (
       first parameter,second parameter,...)
```

Figure 5.3: The evaluator algorithm.

5.2 Implementation Issues

Due to space limitation, we focus on two implementation topics: memorization of function results and complex object algebra implementation.

"Memoization" is an important optimization strategy in functional languages [Michie68, Field88]. This strategy is a generalization of constant expressions and common subexpressions in relational databases [Cluet92]. As the execution of complex functions (e.g. collection functions or complex object functions) is expensive, functions having the same arguments should be executed only once. A function call may be a complex functional expression, if its arguments are also function calls. FAD and GENESIS point out the importance of the memoization strategy, but they do not describe an exact implementation. With OFL the implementation is simple. Functional expressions are described by an execution unit directed acyclic graph (DAG). To avoid multiple evaluations, a function already stored in the DAG with identical parameters is marked with a flag specifying it has not to be re-evaluated. A pointer to the function execution unit having the same result is memorized. When evaluating a DAG, we keep each function result value in its execution unit, so it can be retrieved easily from the pointer. The cache memory needed for keeping results is managed as a global stack. After each collection processing phase (e.g., after a predicate or projection evaluation), the program can deallocate from the stack all the pointers referencing values that are no longer used in the evaluation.

The execution model is being used to implement a complex object algebra [Machuca94]. The algebra implemented is a

Road1 and Road2 Objects Number	Set-oriented Strategy			Pipelined Strategy			T1/T2 Ratio
	Geographic Function calls Number	Geographic Function execution time	Total Execution Time (T1)	Geographic Function calls Number	Geographic Function execution time	Total Execution Time (T2)	
10	20	0.004	0.03	30	0.006	0.03	1 00
20	40	0.008	0.08	100	0.020	0.07	1.14
50	100	0.020	0.15	450	0.090	0.12	1.25
100	200	0.040	0.71	1100	0.220	0.51	1.39
200	400	0.080	0.99	1400	0.286	0.64	1.54
500	1000	0.100	1.33	1700	0.352	0.79	1.68
1000	2000	0.400	1.58	1900	0.384	0.87	1.81
2000	4000	0.800	1.91	2500	0.510	0.98	1.94

Figure 5.5: Results of measures with memoization

Road1 and Road2 Objects Number	Set-oriented Strategy			Pipelined Strategy			T1/T2 Ratio
	Geographic Function calls Number	Geographic Function execution time	Total Execution Time (T1)	Geographic Function calls Number	Geographic Function execution time	Total Execution Time (T2)	
10	40	0.008	0.04	110	0.02	0.03	1.33
20	80	0.016	0.09	420	0.08	0.09	1.00
50	200	0.040	0 27	2050	0.41	0.61	0 44
100	400	0.080	0.98	5100	1.02	1.14	0.85
200	800	0.160	1.25	6200	1.24	1.42	0.88
500	2000	0 400	1.58	8450	1.69	1 82	0.87
1000	4000	0.802	1.97	12200	2.44	2.61	0.75
2000	8000	1.604	2.58	18400	3.68	3.94	0.65

Figure 5.6: Results of measures without memoization

generalized version of the Encore algebra [Shaw90]. The operations are implemented using quantifier functions having as parameters (e.g., collections, predicate and projection expressions) functions and abstract values. These operations are called from two application environments, the GEOS geographical object server and the SPOKE persistent object-oriented programming language. The GEOS geographical object server maps queries on collections of arcs, polygons or points to the algebra. The GEOS optimized execution plan is generated by using an algebraic approach. It is then converted to a Sequence operation applied to a list of quantifier functions. In contrast, the SPOKE optimizer uses a pipelined strategy for queries involving path expressions.

5.3 Performance Measurement

Our performance study focuses on the comparison between set-oriented and pipelined functional program executions. The measures illustrate also the importance of function result memorization. In this section, we extract a sample of a complete benchmark that has been developed for Geographical Information Systems in the Pythagoras Esprit project. In this benchmark, functions applied to complex objects are usually expensive. We select here the geographic length function applied to geographic objects representing roads. The user is interested in comparing road lengths of two regions described in Road1 and Road2 class extents. Figure 5.4 shows the OQL user query and the corresponding annotated set-oriented, and pipelined OFL programs.

```
SELECT   r1.GeoRoadLength, r2.GeoRoadLength
FROM     r1 in Road1, r2 in Road2
WHERE    r1.GeoRoadLength > 1000 and
         r2 GeoRoadLength > 1500 and
         r1.GeoRoadLength> r2.GeoRoadLength
```

The set-oriented strategy gives:

The annotated OFL program

```
ForAll(
  ForAll(Road1,
    DoubleGt(GeoRoadLength(Current(Road1)),1000),
    Tuple(GeoRoadLength(Current(Road1))) [T1],
  null,
  ForAll(ForAll(Road2,
    DoubleGt(GeoRoadLength(Current(Road2)),1500),
    Tuple(
      GeoRoadLength(Current(Road1))))) [T2],
      DoubleGt(GetAtt(Current(T1),1),
               GetAtt(Current(T2),1)),
    Sequence(Display(GetAtt(Current(T1),1)),
             Display(GetAtt(Current(T2),1)))))))
```

```
The pipelined strategy gives:

The OFL program

ForAll(Road1,
   DoubleGt(GeoRoadLength(Current(Road1)),1000),
   ForAll(Road2,
      And(DoubleGt(GeoRoadLength(Current(Road2)),1500),
          DoubleGt(GeoRoadLength(Current(Road1)),
                   GeoRoadLength(Current(Road2)))),
      Sequence(Display(GeoRoadLength(Current(Road1)),
               Display(GeoRoadLength(Current(Road2)))))
```

Figure 5.4: Set-oriented and pipelined translation of a geographic user query

The functional programs were evaluated on collections containing from 100 to 2000 objects. The size of the collections has been kept small to guarantee that no memory page fault occurs at execution time. This highlights the difference between the function evaluation strategies. We measure the number of geographic function calls, the function execution time and the total execution time for the set-oriented and for the pipelined strategies. The number of function calls is determinated by the number of objects in the collections and the predicate selectivity. Figure 5.5 and 5.6 give the results of these measures, respectively with and without memorization. The results of Figure 5.5 show that the pipelined strategy outperforms the set-oriented one in all cases. The gap between the two strategies increases as the size of the temporary collections grows. The comparison between Figure 5.5 and Figure 5.6 clearly shows the importance of memorization. The complex geographic function participates several times in predicate and projection expressions. Thus, it is important to evaluate it only once. Memorization is even more important with a pipelined strategy. This can be explained by the fact that the pipelined program calls the geographic function within the internal loop while the set-oriented one calls it only for the initial filtering. Without memorization, the number of function calls performed by the pipelined strategy highly increases and the set-oriented strategy becomes rapidly the best strategy. As a conclusion, an optimized system should implement both strategies and choose at each step between them.

6. Compiling OQL-like Queries in OFL

OFL is a pure intermediate language satisfying the most important qualities of such a language [Aho88]. We have already shown in section 4, that OFL programs are easily generated by applying traversal strategies in the collection traversal graph. We have also shown in section 5, that OFL programs are easily executed. In this section we address the integration of OFL in OQL-like query compiler processes. We briefly describe i) a syntax collection traversal graph, a straightforward representation of an OQL-like query; ii) a mapping from the syntax collection traversal graph to the collection traversal graph; and iii) a simple integration in OQL-like query compiler processes of the syntax collection traversal graph, the collection traversal graph, and the OFL programs.

The Syntax Collection Traversal Graph

An OQL-like query describes a traversal of collections. A syntax collection traversal graph is a graphic representation of an OQL-like query. The graph vertices represent the collections involved in the query. To be more exact, each collection and its corresponding variable defined in the FROM clause of the query is represented by a vertex. Each vertex is labelled by a tuple of three values: the collection name, the variable name, and the dictionary information entry [Gardarin94].

Arcs represent traversal precedences. Each arc is labelled by a tuple of three values: a quantifier execution unit, a path expression if any, and a traversal precedence type. Quantifier execution units store n-ary predicates and projection expressions. Predicates and projections are deduced from the clause WHERE and SELECT, respectively. Predicates and projections are graphs similar to those in the collection traversal graph. In fact, variables in the syntax collection traversal graph are equivalent to current instances in the collection traversal graph. Traversal precedence types include: functional join, explicit join, and subquery. Path expressions and traversal precedence types are directly obtained from the FROM clause. Let us illustrate the syntax collection traversal graph for the OQL query in Figure 4.2. A target node (black filled) in the syntax collection traversal graph is used to indicate the end of a traversal. It is similar to the target node in the collection traversal graph. To simplify the figure, we omit additional node and arc information. In this graph there are only functional join traversal precedence arcs. We illustrate in the nodes how the collection is deduced from path expressions described in the arcs.

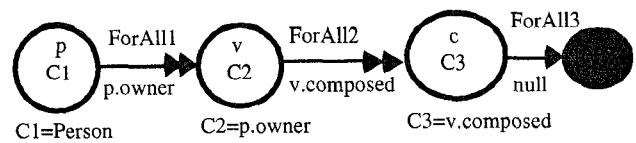

Figure 6.1: A Syntax Collection Traversal Graph of the OQL query in Fig. 4.2.

A formal definition of the syntax collection traversal graph is proposed in [Gardarin94]. If the query is expressed as union, intersection, or difference of sub-queries, each sub-query corresponds to a single syntax collection traversal graph. In general, each SFW (select from where) is represented by a syntax collection traversal graph; or union, or difference, or intersection of syntax collection traversal graphs.

The Mapping from the Syntax Collection Traversal Graph

We have seen that the collection traversal graph can sometimes be a straightforward representation of an OQL-like query. That is the case when one collection of the syntax collection traversal graph can be mapped with a unique abstract collection in the collection traversal graph. The methods of the collection become the functions of the abstract collection. However, a collection in the syntax graph, by inheritance principles, can correspond to n abstract collections in the collection traversal graph. This problem is completely analyzed in [Gardarin94] since special indexes can be used for faster traversal in multiple collections [Kilger94, Sreenath94]. Another important point in the mapping is that an OQL-like query may have overlapping path expressions [Kemper90]. These give overlapping functional expressions in the collection traversal graph [Gardarin94].

A Basic Chain Integrating OFL

The syntax collection traversal graph, the collection traversal graph, and the OFL programs are the basic elements in the functional graph traversal optimization strategy proposed in

[Gardarin94]. These elements are logically connected as shown in Figure 6.2. In our strategy we extensivily use the collection traversal graph as a global representation, which permits classical optimization strategies and new simple heuristics. The importance of a global representation (i.e., the typed algebra) was illustrated in [Cluet92]. For example, it allows exhaustive global factorization of common subexpressions. In addition, the collection traversal graph models dataflow dependencies, which are suitable for parallel executions [Bancilhon87].

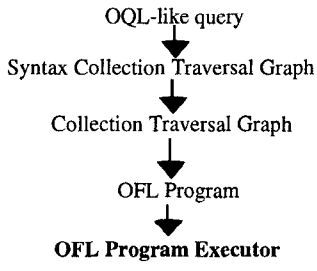

Figure 6.2: The Basic Chain Integrating OFL.

7. Summary and Future Research

We have proposed an intermediate functional language as a target language for object-oriented OQL-like query compilers. The intermediate language supports any complex object algebra with recursion. This functional language deals with abstract collections of objects. By unifying the management of class extents, multivalued attributes, indexes and temporary results; abstract collections make the functional language independent of any data model and any underlying storage model. The proposed functional constructs can be used as building blocks to define ad-hoc operators as well as to express different execution strategies. The functional language lends itself to a functional execution model. A graph based formalism has been introduced to ease the translation of user queries into a functional program. Either set-oriented and pipelined strategies can be generated for a user query depending on the way the collection traversal graph is traversed. This functional language and its execution model have been implemented in the IMPRESS Esprit project on top of an object manager. Performance measures have showed the value of supporting set-oriented and pipelined execution strategies.

We are working on the optimizer for compiling OQL queries, and on the extensions of the evaluator to a parallel architecture. Optimization appears to be a classic problem for graph traversals. The nature of the problem is determined by the kind of traversal precedences. For functional traversal precedences it seems to be a topological sorting problem: the best functional program is one that performs the least costly translation of the traversal collection graph. A cost model has to be developed to valuate the edges of this graph with traversal costs and traversal selectivities and to estimate the size of the visited collections. Then, search strategies could be used to select the best traversal. Parallelism could be considered in the evaluator at a fine object granularity. Quantifier functions can be evaluated in parallel by different processors, each having its own memory of objects. Further work remains to be done on these topics.

Acknowledgements

We are indebted to the anonymous reviewers for their constructive comments which have enhanced this paper. Special thanks to Carl Gewirtz who helped edit the paper.

References

[Aho88] Aho A. V., Sethi R., Ullman J. D., "Compilers : Principles, Techniques, and Tools," Book, chapter 1, 8 to 12, Addison-Wesley, 1988.

[Amman85] Amman A., Hanrahan M., and Krishnamurthy R , "Design of a Memory Resident DBMS", IEEE COMPCON, San Francisco, California, February 1985.

[Amman92] Amman B., Schol M., "Gram: A Graph Data Model and Query Language", ACM ECHT'92, Milan, December 1992.

[Bakus81] Bakus J., "Function Level Program as Mathematical Objects," ACM Transactions on Database Systems, 6(1), October 1981.

[Bancilhon86] Bancilhon F., "An amateur's Introduction to Recursive Query processing Strategies," Intl. ACM SIGMOD Conf. on Management of Data, Washington D.C., May 1986.

[Bancilhon87] Bancilhon F., Briggs T., Khoshafian S., Valduriez P., "FAD, a Powerful and Simple Database Language," Intl. Conf. on VLDB, Brighton, England, 1987.

[Batory88] Batory D., Leung T., and Wise T., "Implementation Concepts for an Extensible Data Model and Data Language," ACM Transactions on Database Systems, 13(3), September 1988.

[Beeri90] Beeri C, Kornatzky Y., "Algebraic Optimisation of Object-Oriented Query Languages," in Proc. ICDT, Paris, France, 1990.

[Bitton86] Bitton D., Turbyfill C., "Performance Evaluation of Main Memory Database Systems", Cornell University, TR 86-731.

[Blakeley93] Blakeley J., Graefe G., "Experiences Building the Open OODB Query Optimizer," Intl. ACM SIGMOD Conf. on Management of Data, 1993.

[Buneman79] Buneman P., Frankel R.E., "FQL—A functioal query language—," Intl. ACM SIGMOD Conf. on Management of Data, 1979.

[Carey88] Carey, M. J., Dewitt D., Vandenberg S. L., "A Data Model and Query Language for EXODUS," Intl. ACM SIGMOD Conf. on Management of Data, 1988.

[Cattell93] Cattell R.G.G. Ed., "Object Databases: The ODMG-93 Standard", Book, Morgan & Kaufman, 1993.

[Cluet92] Cluet, S., Delobel C., "A General Framework for the Optimization of Object-Oriented Queries," Intl. ACM SIGMOD Conf. on Management of Data, 1992.

[Cruz88] Cruz, I.F., "Domains of Application for the G^+ Query Language", Office and Database Systems Research, ed. F.H. Lochovsky, CSRI, Univ. of Toronto, 1988.

[Dar91] Dar S., Agrawal R., Jagadish H.V., "Optimization of Generalized Transitive Closure Queries," Intl. Conf. on Data Engineering, Kobe, Japan, April 1991.

[Delobel91] Delobel, C., Lécluse C., Richard P., "Bases de Données : des Systèmes Relationels aux Systèmes à Objects," Book, InterEditions, Paris.

[Field88] Field, A., Harrison P., "Functional Programming," chapter 11 and 12, Book, Addison Wesley 1988.

[Finance94] Finance, B., Gardarin G., "A rule-based optimizer with multiple search strategies," Data & Knowledge Engineering 13, 1994.

[Frohn94] Frohn J., Lausen G., Uphoff H., "Access to Objects by Path Expressions and Rules", Intl. Conf. on VLDB, Santiago, Chile, August 1994.

[Gardarin92a] Gardarin G., Valduriez P., "ESQL: An Object-Oriented SQL with F-Logic Semantics," Intl. Conf. on Data Engineering, Phoenix, February 1992.

[Gardarin92b] Gardarin G., Lanzelotte R., "Optimizing Object-Oriented Database Queries using Cost-Controlled Rewriting" 3rd Intl. Conf. conference on Extended Data Base Technology. (EDBT), Springer-Verlag Ed.,Vienna, Austria, March 1992.

[Gardarin94] Gardarin G., Machuca F., Pucheral P., "A Functional Traversal Graph Optimization Strategy for OQL-like Query Optimizers", Technical Report, PRISM Laboratory, (submitted for publication), 1994.

[Jenq90] Jenq, B. P., Woelk, D., Kim, W., Lee, W., "Query Processing in Distributed ORION," Intl. Conf. EDBT, Venice, Italy, March 1990.

[Kemper90] Kemper A., G. Moerkotte, "Advanced Query Processing in Object Bases Using Access Support Relations," Intl. Conf. on VLDB, Brisbane, Australia, 1990.

[Kilger94] Kilger C., Moerkotte G., "Indexing Multiple Sets," Intl. Conf on VLDB, Santiago, Chile, August 1994.

[Kim89] Kim, W., Kim K., Dale A., "Indexing Techniques for Object-Oriented Databases," Book, chapter 15, Addison-Wesley, 1988.

[Kim92] Kim, W., Sagiv Y., Kifer M., "Querying Object-Oriented Databases," ACM SIGMOD, USA, June 1992.

[Lanzelotte93] Lanzelotte R., Cheiney J.P., "Vers une nouvelle génération d'optimiseurs pour les SGBD orientés objet," Tecnique et Science Informatiques, Volume 12(4), 1993.

[Lehman86] Lehman T., Carey M., "Query Processing in Main Memory Database Management Systems", ACM SIGMOD Int. Conf., Washington, D.C., May 1986.

[Machuca94] Machuca F., Gardarin G., Pucheral P., "A Functional Execution Model for Complex Object Algebras," Technical Report, PRISM Laboratory, University of Versailles Saint-Quentin, Versailles, France, February, 1994.

[Michie68] Michie D., "Memo Functions and Machine Learning," Nature, (218), 1968. pp. 19-22.

[Orenstein92] Orenstein J., Haradhvala S., Margulies B., Sakahara D., "Query Processing in the ObjectStore Database System," Intl. ACM SIGMOD Conf. on Management of Data, 1992.

[Pucheral92] Pucheral P., Thévenin J.M., "Pipelined Query Processing in the DBGraph Storage Model," Intl. Conf. on EDBT, Vienna, Austria, March 1992.

[Selinger87] Selinger P. et al., "Acces Path Selection in a Relational Database Managment System," in Proc. of the ACM SIGMOD Conference, New York, 1987.

[Shaw90] Shaw G., Zdonik S., "A Query Algebra for Object-Oriented Databases," in IEEE, 1990.

[Shipman81] Shipman, D., "The functional Data Model and the Data Language DAPLEX," ACM Transactions on Database Systems, 6(1), March 1981.

[Shekita90] Shekita E. J., Carey M. J., "A Perfomance Evaluation of Pointer-Based Joins," in Proc. of the ACM SIGMOD Conference, 1990.

[Sreenath94] Sreenath B., Seshadri S., "The hcC-tree: An Efficient Index Structure For Object Oriented Databases," Intl. Conf. on VLDB, Santiago, Chile, August 1994.

[Steenhagen94] Steenhagen H. J., Apers P. M.G., Blanken H., de By R. A., "From Nested-Loop to Join Queries in OODB," Intl. Conf. on VLDB, Santiago de Chile, Chile, September 1994.

[Straube90] Straube D., Ozsu T , "Queries and Query Processing in Object-Oriented Database Systems,"Technical Report, Departement of computing science, university of Alberta, Edmonton, Alberta, Canada, 1990.

[Valduriez86] Valduriez P., Khoshafian S., Copeland G., "Implementation Techniques of Complex Objects," Intl. Conf. on VLDB, Kyoto, August 1986.

[Valduriez91] Valduriez P., Lanzelotte R., Ziane M , and Cheiney J.P., "Optimization of non Recursive Queries in OODB," In Proc. DOOD, Munich, Germany, 1991.