Efficient Management of Object-Oriented Queries with Late Binding

by

Staffan Flodin

February1996
ISBN 91-7871-667-5
Linköping Studies in Science and Technology
ISSN 0280-7971
Thesis 538
LiU-Tek-Lic 1996:03

ABSTRACT

To support new application areas for database systems such as mechanical engineering applications or office automation applications a powerful data model is required that supports the modelling of complex data, e.g. the *object-oriented* model.

The object-oriented model supports *subtyping*, *inheritance*, *operator overloading* and *overriding*. These are features to assist the programmer in managing the complexity of the data being modelled.

Another desirable feature of a powerful data model is the ability to use inverted functions in the query language, i.e. for an arbitrary function call *fn(x)=y*, retrieve the arguments *x* for a given result *y*.

Optimization of database queries is important in a large database system since query optimization can reduce the execution cost dramatically. The optimization considered here is a cost-based global optimization where all operations are assigned a cost and a way of a priori estimating the number of objects in the result. To utilize available indexes the optimizer has full access to all operations used by the query, i.e. its *implementation*.

The object-oriented data modelling features lead to the requirement of having late bound functions in queries which require special query processing strategies to achieve good performance. This is so because late bound functions obstruct global optimization since the implementation of a late bound function cannot be accessed by the optimizer and available indexes remain hidden within the function body.

In this thesis the area of query processing is described and an approach to the management of late bound functions is presented which allows optimization of invertible late bound functions where available indexes are utilized even though the function is late bound. This ability provides a system with support for the modelling of complex relations and efficient execution of queries over such complex relations.

This work has been supported by The Swedish Board for Industrial and Technical Development (NUTEK) and The Swedish Research Council (TFR).

Department of Computer and Information Science
Linköping University
S-581 83 Linköping
Sweden

1 Introduction

This thesis addresses the problem of how to *efficiently* manage *late bound* function calls in the *execution plan* in an *object-oriented database management system* (*OODBMS*) [4].

An object-oriented database management system is a system based on the object-oriented model with database facilities [4][26]. An execution plan is a strategy of how to efficiently access and combine data in the database to produce the answer to a query submitted to the database [52][53][54]. A late bound function is a function whose implementation cannot be selected in advance of its invocation since function names may denote several implementations and the selection of implementation is made based on the types of the arguments the function is applied on.

This is an important issue to address in the overall area of *query processing* in a *database management system* (*DBMS*).

To support new application areas such as mechanical engineering applications or office automation applications, a powerful data model is required that supports the modelling of complex data, e.g. the object-oriented model [8][10].

The object-oriented model supports *subtyping*, *inheritance*, *operator overloading* and *overriding*. These are features to assist the programmer in managing the complexity of the data being modelled.

Optimization of database queries is particularly important in order to achieve satisfactory system performance since query optimization can reduce execution cost dramatically. State of the art query optimization is cost-based where all operations are assigned a cost and a way of a priori estimating the number of result objects [47]. It is important that the optimizer has access to all referenced operations, i.e. *global optimization*. To achieve global optimization, all function calls in the execution plan are substituted by their implementations.

Unfortunately, the object-oriented features lead to the need for late bound functions in queries, which is contradictory to good performance. The reason for this contradiction is that late bound functions obstruct global optimization since the function calls cannot be substituted by their implementations and available indexes and other important information remain hidden within the late bound function bodies.

Another desirable feature of a powerful data model is the ability to use inverted functions in the query language, i.e. for an arbitrary function call $fn(x)=y$, to retrieve the arguments x for a given result y.

Invertibility and late binding is a combination that requires special treatment to be executable and optimizable. We propose such novel query processing strategies for managing inverted late bound functions in the execution plan.

Our strategy is to substitute each late bound function call in the execution plan by an algebra operator, DTR, which is invertible and optimizable.

The ability to use inverted late bound functions and to optimize them using a cost-based optimizer means that the advantages of the modelling capabilities can be fully utilized with little or no performance degradation.

The main contribution of this thesis are:

- An efficient method of executing late bound functions, both inverted and non-inverted.
- A method to make late bound functions optimizable using a regular cost-based optimizer.
- A performance study which demonstrates a dramatic performance improvement achieved by our approach.
- A mechanism for resolving when late binding must be used and a description of an incremental query compiler.

1.1 Outline of the thesis

The next section, section 1.2, presents an introduction to query processing which will be followed by an example using the relational data model. To illustrate the object-oriented model the example will then be remodelled using this model.

In section 1.5 the text conventions and denotations used throughout this thesis will presented. In chapter 2 a basic review of the object-oriented and functional data models is given and is followed by a survey of some object-oriented and extended relational database systems in chapter 3.

This provides the background for type resolution and invertibility in an object-oriented model which is presented in chapter 4. In chapter 5 the solution to the management of late bound functions is presented. The solution is to be placed in the context of the data model described in the proceeding chapters.

To improve the object-oriented model, the message passing style of function invocation will be replaced by a strategy using *multi-functions*. A multi-function is a function where all argument types are used to resolve which implementation to employ. In chapter 6 the model described chapter 4 is extended with multi-functions and type resolution in such a model is described.

Finally, a discussion and outline of future work are presented in chapter 7.

1.2 Query processing

The term query processing refers to all actions that need to be performed in order to execute a request for information over some data maintained by a DBMS.

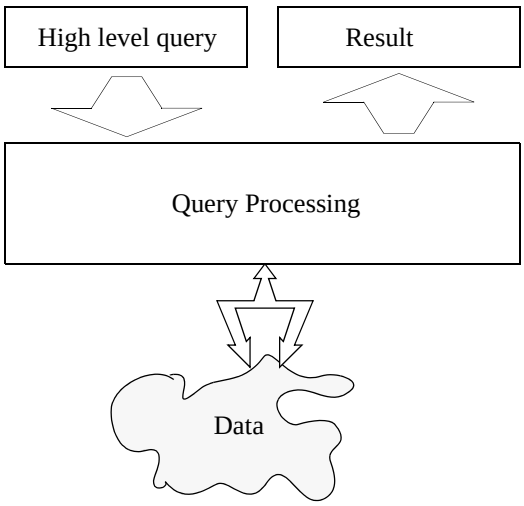


Figure 1.1: Query processing overview

Processing a query involves interpretation and execution of the query expressed in some database language, for example SQL. In figure 1.1 a schematic overview of query processing is given. A statement expressed in some database language is fed into the query processor. It is then the responsibility of the query processor to perform whatever the statement expresses and return an answer to the caller.

The area of query processing covers a wide range of issues. The problems addressed in this thesis relate to the translation of a high level query into an intermediate representation of the query, i.e. an algebraic representation (the box coloured black in figure 1.2), and the transformation (optimization) of the generated algebra expression.

The translation of a high-level declarative representation to a low-level procedural representation is a major step which involves several sub-issues; For object-oriented systems one such a problem is providing an algebraic representation of late bound functions.

A refined view of query processing is pictured below (fig. 1.2) where the overall task of query processing has been broken up into several sub-tasks.

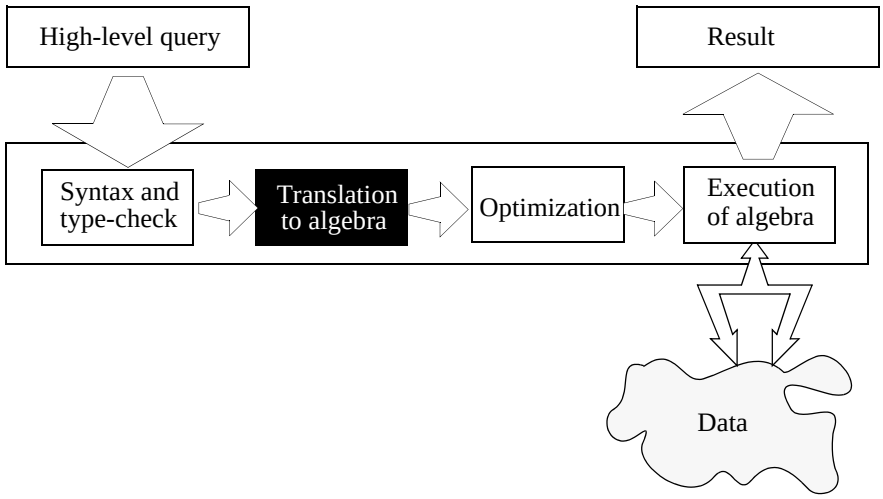


Figure 1.2: Query processing steps

- The query is expressed in a high-level *declarative* language which is translated into some intermediate representation in order to make it easier for the system to handle. In a declarative query language the user specifies what data is to be retrieved rather than how to retrieve it; the latter is the task of the query processor.
- The declarative representation of the query is *syntax* and *type-check*ed to make sure the query is correctly expressed and that type related runtime error will not occur. A syntactically correct and type-checked query is further processed by the query processor. An erroneous query will cause an exception to be signalled and no further processing will be carried out.
- The type-checked query is translated into an intermediate representation. This intermediate representation is constructed in such a way that it can be processed efficiently by a system, i.e. an *algebra*, a procedural language. An algebra typically consists of low-level operations to access the data and to perform certain operations on the data. In some cases the query is first translated into another non-procedural representation, a *calculus*, on which certain optimizations are performed before the algebraic representation of the query is constructed.
- Optimization of a query is the task of finding the best plan or at least a good plan among all possible execution plans. The execution plan is derived from the algebraic representation. In order to be able to optimize a query, the system must have the ability to generate all equivalent execution plans and the ability to judge whether a certain execution plan, ***A***, is cheaper to execute than another plan, ***B***, according to some cost measure.

- The optimized representation is then executed and the result is returned to the caller, user or process. Execution of a query involves the traversal of the execution plan and execution of each statement in the plan. It is in this phase of the query processing the data in the database is accessed.

Query processing in an object-oriented database management system is more demanding than query processing in a *relational database management system* (*RDBMS*) [26] since the discrepancy between the conceptual data model and the physical storage is much larger [54]. This is so because the expressiveness of the object-oriented (OO) model allows the application programmer to build complex models which then require very powerful query processing capabilities to provide efficiency.

1.3 The relational model

An important milestone in the history of database management was the introduction of the Relational model [13] in the 70's. Along with the relational model came systems with query languages and *data independence*. Data independence is an important feature which isolates the user or the application from the physical representation of the data. In a system with a high level of data independence it is possible to change the representation of the data and add new data without having to change any applications accessing the data. If data is removed, then only applications accessing the removed data should be affected.

The relational model is a simple and comprehensive model that has gained popularity and commercial success. Many commercial systems use the *SQL* query language. SQL is an abbreviation for *Structured Query Language* and includes a declarative query language. SQL was developed at IBM to interface their prototype SYSTEM-R [5] and was originally called SEQUEL. In addition to the declarative query component, SQL contains procedural constructs for schema definitions and table population, thus SQL is both a data definition language (DDL) and data manipulation language (DML).

The fundamental concept of the relational model is the *table* into which the data being modelled is entered. Each table consists of one or more *attributes* which can only contain atomic values[1]. Data from different tables can be combined by connecting the rows from the different tables that agree upon the value of some attributes, a *join*[2]. Other operations on the table are *selection* and *projection*. A selection retrieves the rows in a table that have a certain value for a certain attribute. A projection of a table is a view of the table where some attributes have been removed.

1. An atomic value is a value which cannot be divided into sub-components, e.g. integers
2. In this case an equi-join. For theta-joins other operators than = are allowed; e.g. <, >, etc.

Table:Employees

Name	Ssn	Age
Bill	21	39
John	5	21
Steve	43	28
Ron	2	56

Table: Works_At

Department	Empl
Construction	21
Management	2
Sales	5

Figure 1.3: Relation tables

In figure 1.3 there are two relation tables: One table, *Employees*, lists the name, social security number and the age of the employees of a company and the other table, *Works_At*, lists the department name and the social security number where each employee works. By joining the two tables on the attributes *Ssn* and *Empl*, respectively, we can also work out the names of the persons that work at each department.

Although the relational model was a major step forward, it has its shortcomings when it comes to modelling complex data. As will be shown, modelling complex data in the relational model can be difficult since the data has to be adapted to the flat structure of a table where each attribute must be atomic valued. The data has to be adapted to tables in such a way that certain properties hold for the data. These properties are called *Normal forms* [26].

1.3.1 An example

To illustrate data modelling in the relational model consider the example where a company wants to keep track of which sub-components a given component consists of and of who delivers these components. This example will be reused in section 1.4.2 to illustrate data modelling using the object-oriented model.

To model this example in the relational model, all data have to fit into tables consisting only of atomic-valued attributes.

Table: Component

Id	manuf	name
1	m6	PenA
2	m3	Tube
3	m6	Tube
4	m7	Cartridge
5	m2	Inktube
6	m6	Ballpoint
7	m7	Spring
8	m2	Ball
9	m3	Cartridge

Table: Consists_of

Id1	Id2
1	2
1	3
1	4
4	5
4	6
6	7
6	8
6	9
...	...

Table: Manuf

Name	No	Street	Id
ICInc	1	A Rd	m2
H&V	3	D Rd	m3
Jans	4	B Rd	m6
Sears	9	C Rd	m7
...	...		...

Figure 1.4: Example Relational database

In this example there are three tables. The *Component* table lists all the parts the company handles. The *Consists_of* table lists the components, *Id2*, a certain component, *Id1*, consists of. For example the component named “PenA” with *Id* 1 consists of the components with *Id* 2, 3, and 4. The last table, *Manuf*, lists the manufactures the company purchases parts from.

Consider the scenario where one of the companies which delivers some parts, for example *ICInc*, raises its prices significantly. A relevant database query is the retrieval of all parts that consist of some parts from the company named *ICInc*. This query can be expressed in SQL as:

```
SELECT c2.name
FROM component c, component c2, consists_of co, manuf m
WHERE m.name='ICInc' AND c.manuf=m.id AND
      co.id2=c.id AND co.id1=c2.id
```

Example 1: SQL query

One possible corresponding relational algebra *query tree* to the query is illustrated in figure 1.5. The initial query tree mirrors the structure of the original query. On the initial tree rewrite rules are applied to transform the tree into an equivalent tree that is more efficient to execute. Finally, the execution plan is derived from the rewritten query tree by traversing the tree in some order. Thus, the following tree is a result of several transformations on the initial tree generated from the declarative query.

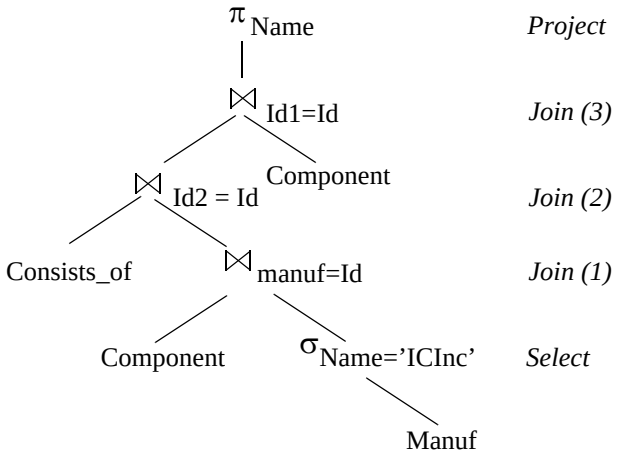


Figure 1.5: Relational algebra query tree

The execution plan is described by the query tree by defining a traversal order of the tree. One such traversal of the query tree above is to first access the *Manuf* table and all rows which satisfy the condition *name='ICInc'* are selected. The selected rows are then joined with the *Component* table. The result of the first join is joined with the *Consists_of* table using the join condition that the *Id2* attribute from the *Consists_of* table must be equal to the *Id* attribute from the *Component* table in the result of the first join.

The result of the second join is then joined with the *Component* table and the result of this join is projected onto the *name* attribute from the *Component* table which will be the result of the query.

Using the data in the example (fig. 1.4) the result is the set of names {*Cartridge*, *Ballpoint*}[3]. Note that the components manufactured by *ICInk* do not appear in the result set, i.e. the components named *Inktube* and *Ball*.

Perhaps the easiest way to attend to this problem is to extend the *Consists_of* table with rows that contain the same value for both attributes, i.e. *Id1* and *Id2*. Another solution is to define an entirely new query which retrieves the union of the results of the first and third join before the projection onto the attribute *name*. This fairly simple query on the data results in a complicated query tree containing three joins.

To overcome the shortcomings of the relational model, much attention database research has focused on the object-oriented model within the database community. The object-oriented model is a much richer model oriented towards modelling the behaviour of objects instead of oriented towards fitting the data into a specific data structure, e.g. the relation table.

3. This example can be further complicated by saying that any part that consists of a part in the result set should also be a member of the result, i.e. transitive closure.

1.4 Object-oriented concepts

In this section a brief overview is given regarding the terminology of the object-oriented model as defined in [4] and the example from section 1.3.1 is modelled in the object-oriented model.

The object-oriented model is an attempt to support the modelling of complex applications such as computer aided design (CAD), office information systems and finite element modelling (FEM) [8][10]. The object-oriented model contains constructs that enable the programmer to manage data involving complex relationships relatively easy compared to modelling within the relational model.

The object-oriented model is a general concept describing the features of object-orientation. Specific variants of the object-oriented model are called *object-oriented data models*. For example the data model of C++ [57] is an object-oriented data model

1.4.1 Object-oriented terminology

The object-oriented model is built on the notion of an *abstract data type* (*ADT*) where the ADT is modelled as a *class* and the functions and procedures that operate on the abstract data type are *methods* of the class. All o*bjects* are instances of some class i.e. a data item of some data type. In this thesis the term *type* is used in favour of class.

Objects that share the same behaviour are grouped into types. The types are organized in a *subtype - supertype* hierarchy where subtypes inherit properties from their supertypes. Inheritance means that if type ***A*** is a subtype of type ***B*** and if ***B*** has a certain property, ***p***, then the type ***A*** also possesses the property ***p***. Properties include functions, procedures and variables.

The subtyping can be viewed as a specialization of general behaviour or concept into more specific behaviour or a more specific concept. For example a pit bull terrier is a specialization of the more general concept of dog.

To describe the behaviour of objects *functions* (*methods*[4]*) are used*, which are invoked by *message passing*. A message is sent to an object and the object responds to the message by invoking the procedural specification that corresponds to the message. Functions can return *atomic values*, *composite values* or other objects. Atomic values include integers, reals or characters. Composite values include lists, sets and arrays. The support for composite (*complex*) values constitutes a major difference from the relational model where attribute values have to be atomic.

Each object is unique, i.e. two objects can exhibit exactly the same behaviour and yet be different objects. The identity is maintained by the system which assigns each object a unique object identifier, *OID*. This is illustrated in figure 1.6.

4. In standard object-oriented terminology the notion of method is often used, in this thesis the notion of function is preferred.

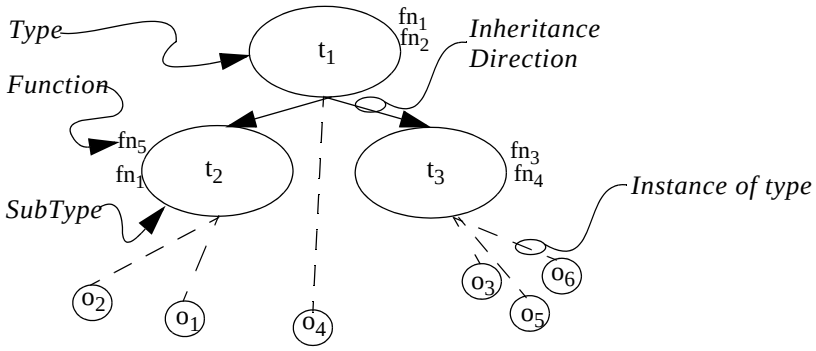


Figure 1.6: Object diagram

In the Object diagram (fig. 1.6) there are three types t_1, t_2 and t_3 with five functions $fn_1 \ldots fn_5$. The functions fn_1, fn_2 are defined in type t_1 and are inherited by types t_2 and t_3.

There are six objects, $o_1 \ldots o_6$. The objects o_1, o_2 are instances of type t_2, o_4 is an instance of type t_1 and the other objects are instances of type t_3.

The functions fn_1, fn_2 are applicable to object o_4 and functions fn_1, fn_2 and fn_5 are applicable to objects o_1, o_2. Functions $fn_1 \ldots fn_4$ are applicable to the instances of type t_3, i.e. objects o_3, o_5 and o_6.

Function names may be overloaded, i.e. given several implementations. Each implementation is called a *resolvent*. Functions inherited by a type can be given a new implementation for that type, i.e. *overriding*. In figure 1.6 the function fn_1 is overridden in the type t_2. Late binding of functions to resolvents is a consequence of overriding.

The object-oriented model is a powerful modelling tool that permits the user to model real-world complex behaviour and relations in a fairly easy manner. The object-oriented model has gained popularity both as a model for building computer programs and is supported by programming languages such as C++ [57], SmallTalk and Eiffel [42] and as a model for managing complex data [18][28][35][36].

1.4.2 The example revisited

Consider again the example presented in section 1.3.1. In this section the example will be modelled using object-oriented concepts.

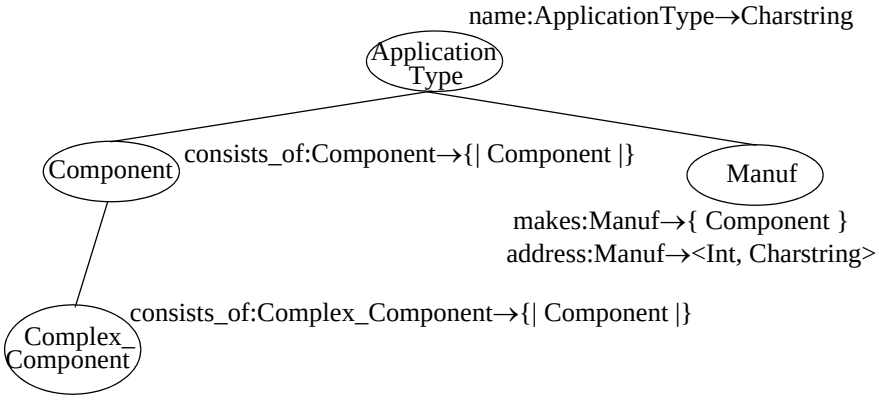


Figure 1.7: Example object-oriented database schema

The above figure shows the types and functions required to model the example in section 1.3.1. The root of the type tree is a type which summarizes all types required in the example application. The function *name* is defined for this type and is inherited by the subtypes.

A function named *consists_o*f is defined for the *Complex_Component* type, which given a *Complex_Component* object returns a multiset[5] of *Component* objects. This function overrides the *consists_of* function defined for the *Component* type which returns the multiset containing only the object itself, i.e. a *singleton* multiset.

Two functions are defined to the *Manuf* type: *makes* and *address*. The function named *makes* returns a set of components for a given object of the *Manuf* type. The function named *address* returns a tuple consisting of an integer value and a charstring value which correspond to the street number and street name respectively.

The schema (fig. 1.7) must be populated with objects that correspond to the data in the tables in the original example (fig. 1.4). The populated database schema is illustrated in an object diagram as follows.

5. A multiset is a set which can contain duplicates, often called bag, denoted {| |}

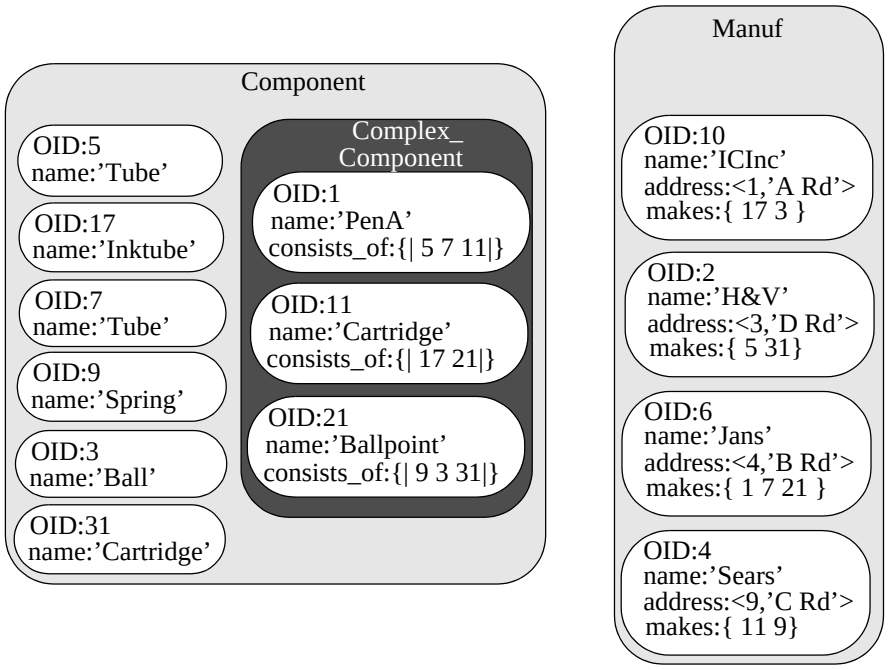


Figure 1.8: Object diagram

In the object diagram (fig. 1.8) all objects which are instances of a particular type are enclosed in a box named after the type. Note that the instances of the *Complex_Component* type are contained within the box of the *Component* type. The reason is that the *Complex_Component* type is a subtype of the *Component* type whose instances are in fact also instances to the *Component* type. Disjoint from the *Component* objects are the objects which are instances of the *Manuf* type.

Once again we are interested in the retrieval of all parts that consist of subparts from the company named *ICInc*. In this example the OSQL [28] language is used.

```
SELECT name(c) FOR EACH Component c, Manuf m
WHERE name(m)='ICInc' AND consists_of(c) = makes(m);
```

Example 2: OSQL query

The =-operator in OSQL is overloaded and in above query the =-operator is equivalent to an intersection operation.

For illustration purpose only, algebra operations similar to the relational algebra operations are used with the extension that complex data can be used in conditions and that *select* and *project* can use functions in their conditions. In

this example algebra, a project using a function is analogous to a function invocation. The translation from the declarative query to the algebra tree is made in several steps. A possible query tree for this query[6] is the following tree:

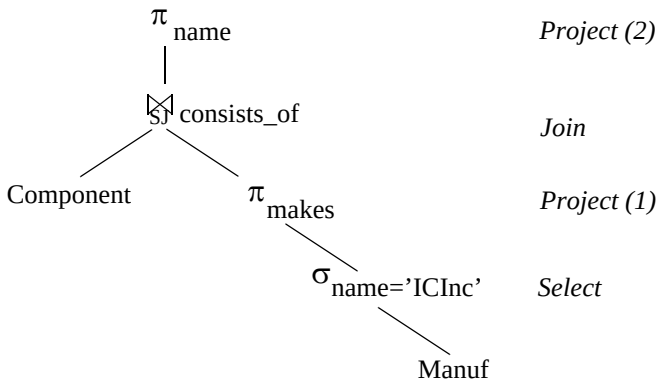


Figure 1.9: Object-oriented Query tree

In the object-oriented query tree (fig. 1.9) of the example two of the three original joins (fig. 1.5) are eliminated and the tree has a simpler structure. The join that remains joins the result of applying the *consists_of* function on all objects of *Component* type, which also includes all objects of the *Complex_Component* type, with the products that the object named '*ICInk*' makes.

The join[7] condition is a set condition which evaluates to a true value if any object in the result of applying the function *consists_of* on the objects in the left branch is a member of the right branch. The results of the join are the objects from the left branch which fulfilled the join condition. The result of the join is projected onto the *name* function and the result is the set of object names: {'*Cartridge*', '*Ballpoint*', '*Inktube*', '*Ball*'}

Recall the query tree from the corresponding relational example (fig. 1.5) which contained three joins. The object-oriented query tree (fig. 1.9) contains only one join. The join that remains has a condition which is fairly complex in nature. To provide the optimizer with more options to find a good plan, an object-oriented algebra has to be constructed out of operations on a lower level, thus the task of query processing will become more complex.

6. The translation from the OSQL query to the query tree is outside the scope of this thesis.
7. This kind of join is called semi-join, denoted ⋈SJ [26].

1.4.3 Late binding

The requirement of having *late binding* in the query language is a consequence of function redefinition in the type hierarchy with inheritance. One example of function redefinition (*overriding*) is given in section 1.4.2 (fig. 1.7) where the function named *consists_of* is defined for the *Component* type and the definition is overridden in *Complex_Component* type.

As shown in the object diagram (fig. 1.8) all objects that are instances of the *Complex_Component* type are also instances of *Component* type. Thus querying the objects of the *Component* type means querying all instances of the *Complex_Component* type as well.

If there exists only one definition of a particular function for the types whose extents are being queried, this definition can be selected before execution. This a priori selection is possible since the definition is applicable to all objects that are instances of the types being queried. This is called *early binding* or *compile-time type resolution*.

If, on the other hand, there exist more than one definition of a particular function defined for the queried types, the definition to be used is dependent on the type of the object which the function is applied to, thus the function definition cannot be selected until the time of application. This is called *late binding* or *run time type resolution*.

Consider the algebra operator *join* in the query tree (fig. 1.9). Here the function *consists_of* must be late bound to select the correct function definition depending on the type of the object. In *project*(*1*) (fig. 1.9) the function *makes* can be early bound.

As the example shows, an object-oriented model with late binding gives the user a very flexible system when modelling complex data.

1.4.4 Invertibility

In the normal case a function is applied to some arguments to produce some result. The values bound to the arguments are known and the result is sought. If, on the other hand, the result is known and the arguments are sought, the inverse of the function can be applied to produce a result. Thus, a desirable property of an declarative object-oriented query language is the ability to invert functions, i.e. for an arbitrary function call in the query language *fn(x)=y*, retrieve the set of arguments *x* for a given result *y*.

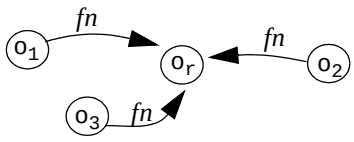
```
square(Number)->Number
sqrt(Number)->{Number}

SELECT x FOR EACH Number x
WHERE square(x)=9;
```

Example 3: Function inverses and application of function inverse

In example 3 two functions are defined, each function being the inverse of the other. The function *square* has the property that it maps two different values to the same result, i.e. the result of its inverse is a set consisting of several values. The inverse of the function *square* is then used in a query. During optimization of the query, a cost-based optimizer will choose the inverse of function *square*, i.e. function *sqrt*, if this makes the query cheaper to execute.

In the context of a database, a function *fn*, may map several different objects to the same result. Thus the inverse, fn^{-1}, must find all objects that return this particular result when the function *fn* is applied to them.



$$fn^{-1}(o_r) = \{o_1, o_2, o_3\}$$

Figure 1.10: Function mappings and inverse

Function inverses should be system inferred and it is the optimizer that selects which function to use, its definition or its inverse, to achieve the best performance.

A special problem is the combination of invertible functions and late binding. The problem is special because the objects in the result of an inverted late bound function are used to select which definition of the late bound function to apply on them.

1.4.5 Problems with late binding

As shown in the example in section 1.4.2, the benefits of late binding are enhanced modelling capabilities and greater flexibility in the query language. Unfortunately, these useful properties have their price. Late binding introduces complexity into the query processing in the following ways:

- Late bound functions obstruct global optimization at compile time since it is not until runtime that the definition can be selected.
- The ability to use inverted functions, i.e. for an arbitrary function call *fn(x)=y* retrieve the argument *x* for a given result *y*, when the function *fn* is late bound constitutes a problem.
- The binding time, i.e. early or late, must be resolved by the system to provide a flexible and expressive query language.
- The query compiler must function incrementally to manage an evolving database schema given that the system resolves whether to bind functions late or early.

These problems will be addressed in the remaining chapters of this thesis.

1.5 Text conventions

This section describes and explains the text conventions and denotations used in the thesis.

1.5.1 Denotations

The letter *t* is used to denote an arbitrary type, t_{sub} denotes any subtype of *t* and t_{sup} denotes any supertype of *t*.

1.5.2 Figures

Figures are used throughout the text to exemplify and enhance the readability. In the figures the following graphical conventions are adopted:

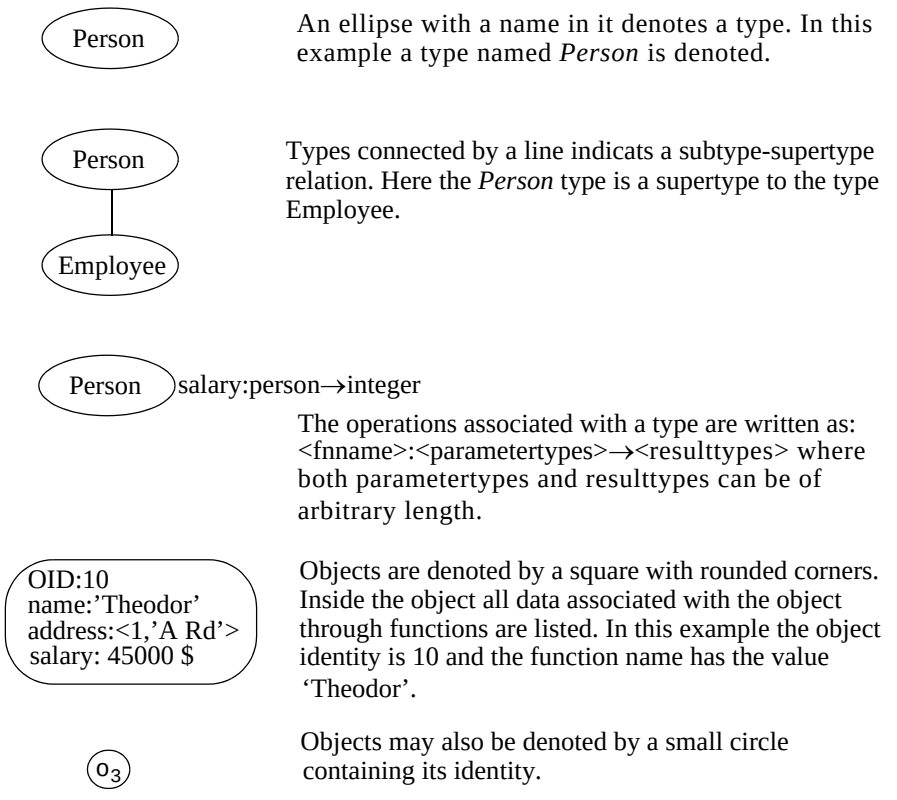


Figure 1.11: Graphical conventions

1.5.3 Fonts

Emphasized text is used to denote a new concept and to distinguish certain words given in normal text in order to enhance readability. For example, type or function names that have appeared in an example are be put in *emphasized* text in order to make the reference to the example more clear.

Examples of code expressions in examples and figures are written in `courier font` to distinguish them from the text. In the text this font is never used.

1.5.4 Naming

In the examples of code reserved words are always CAPITALIZED, names of types are always written with a leading Capital, variable and function names are always written in lower case.

```
FOR EACH Person p SELECT name(p);
```

Example 4: Code example

In the example above an expression that selects the names of all persons in the database is given. The expression is initiated with two reserved words: *FOR EACH*. A variable *p* is declared to be of *Person* type. The functions named *name* is then applied to the variable *p*. Bearing these conventions in mind, little or no knowledge about the syntax of the languages that are used is required.

Intermediate representations of function resolvents and queries are written as:

```
Employee.reports_to(Employee e)->Supervisor   Interface
SELECT _G2
WHERE  _G1=Employee.department(e)and           Implementation
       _G2=Department.mgr(_G1);
```

Example 5: Intermediate query representation

The interface is the name of the function and the argument list and result type. The implementation is the type-checked and rewritten body of the original function.

2 Object-oriented and extended relational models

In this section the object-oriented model and the functional model are described.

2.1 Object-orientation

The object-oriented data model has received a considerable amount of attention during the last decade [4][10][22][23][27][28][41][43][48][53][54]. The object-oriented (OO) model is an attempt to support the modelling of complex applications such as computer-aided design (CAD), office information systems and finite element modelling (FEM) [8][10]. The OO data model contains constructs that enable the programmer to manage data exhibiting complex relationships with relative ease compared to modelling within the relational model.

Modelling data within the relational model forces the programmer to fit all data into tables. The object-oriented model is more of a conceptual model where the programmer has more modelling ‘tools’ and the system is responsible for the mapping between the conceptual OO model and physical storage.

2.2 Object-oriented concepts

In “The Object-Oriented Database System Manifesto” [4] the OO model for database systems is defined. The OO concepts are grouped into *mandatory*, *optional* and *open characteristics* where the mandatory characteristics are those that a system must satisfy in order to be viewed as an object-oriented system. The optional characteristics can be added to enhance the performance of the system. The open characteristics are those where the designer has a number of choices. In this section the mandatory features are reviewed.

The mandatory OO features are:

- Support for complex objects
- Object identity
- Encapsulation
- Types and classes
- Type (Class) Hierarchy
- Overriding, overloading and late binding
- Computational completeness
- Extensibility

2.2.1 Complex objects

Complex objects are built from simpler ones such as *Integer, Character* or *Boolean* as well as user-defined object types by applying a constructor to them. There are various complex object constructors such as set, bag, list, array or tuple. The constructors must be applicable to any object type. For example, the list constructor must be applicable to any object type in the database schema, both user-defined and system-defined object types.

2.2.2 Object identity

Object identity must be supported in order to distinguish between different objects with identical behaviour. Object identity is maintained by a system-generated and immutable *object identifier* (*OID*). Thus objects can be equal, i.e. have the same behaviour and objects can be identical, i.e. have the same object identity.

In an identity-based system objects can be shared and updated easily. Consider the following figure:

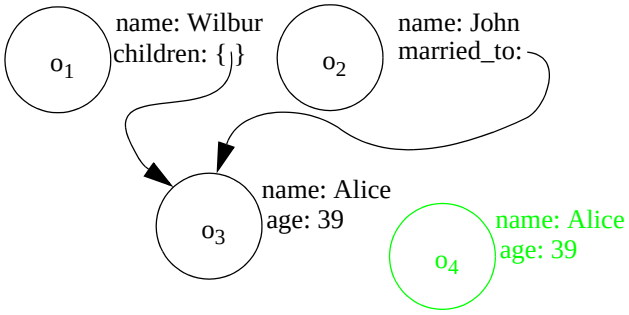


Figure 2.1: Identity-based system

The objects o_1 and o_2 share object o_3 but via different relations. Finding the objects that are related to the same object through the *children* and *married_to* relations involve only object identity tests. In a value-based model the issue is a bit different, consider the following:

```
(<name Wilbur> <children (<name Alice><age 39>))
(<name John> <married_to (<name Alice><age 39>))
(<name Alice> <age 39>)
(<name Alice> <age 39>)
```

Example 6: Value based system

In a value based model, such as the relational model, the above mentioned search means checking that all values are equal. In the example (ex. 6) two 39

year old Alice exist and the question as to whether John is married to the daughter of Wilbur cannot be decided. To overcome such problems in value-based models the concept of a *primary key* and *normalization* [14] are introduced. A primary key is a set of attributes whose values can be used to uniquely identify an object.

Another issue involves managing updates in a value-based system. Whenever the age of Alice changes then the references maintained by Wilbur and John need to be updated as well[8].

This problem does not exist when objects are shared as in an identity based system. In this thesis object identity is considered as a mandatory object-oriented feature (see [23] for an alternative approach).

2.2.3 Encapsulation

Encapsulation is the distinction between behaviour and implementation. The implementation is hidden from other objects and applications, thus the only way to manipulate objects is through the methods applicable to the object, i.e. its *interface*. Encapsulation provides some data independence: the implementation can be changed without any changes in the application programs since these did not have any access to the previous implementation, nor do they have access to the new. This claim is valid as long as the existing interface methods model the same behaviour.

A strict definition of encapsulation allows only methods, not data, to be in the interface and methods can only see and manipulate data within each object. Encapsulation in OO terminology corresponds to data independence in relational terminology.

The strict definition of encapsulation may be too restrictive in some applications and in some implementations of the OO model the encapsulation is relaxed, e.g. *friends* in C++ [57], a declaration that allows object types to access private data from each other.

2.2.4 Types and classes

Support for types or classes is also a mandatory characteristic of an OO database system. Types are traditionally used for correctness control of programs. Types in OO terminology describe the structure of the class where a class is the container for objects and methods. In this thesis we are primarily interested in types and will use the notion of type in the remaining. A type is used to summarize the common features of a set of objects with the same characteristics.

A type consists of an interface and an implementation. The interface consists of a set of operations together with their *signatures*. A signature is the name of the operation annotated with the names of the argument and result types. The implementation part of the type consists of a data part and an operation part. The data part describes the internal structure of the type and the operation part

8. This is the case even for tables with primary keys if the value of an attribute in the primary key is changed.

consists of the procedures that implement the interface operations.

Whether data is allowed in the interface of a type is optional. In [43] a type is specified to contain a set of hidden *instance variables* and in C++ data is allowed in the interface.

All objects are instances of some type and thus it is possible to apply all methods defined for the type to the object. The set of objects that are instances of a particular type is called the *extent* of that type.

2.2.5 Type hierarchy

The types are organized in an inheritance hierarchy. Inheritance is a powerful modelling tool which helps in factoring out shared specifications and implementations.

A type that inherits properties from another type is called a *subtype* of that type, the *supertype*. A subtype specializes the behaviour of its supertype by additional behaviour and, in some models, changes inherited behaviour, thus the subtype-supertype relation can be viewed as a *specialisation - generalisation* relation.

The inheritance mechanism is central to the object-oriented model: creating an OO model of some problem is very much the creation of a type hierarchy with inheritance that mirrors the structure of the problem. Consider the following *database schema* that models a company database:

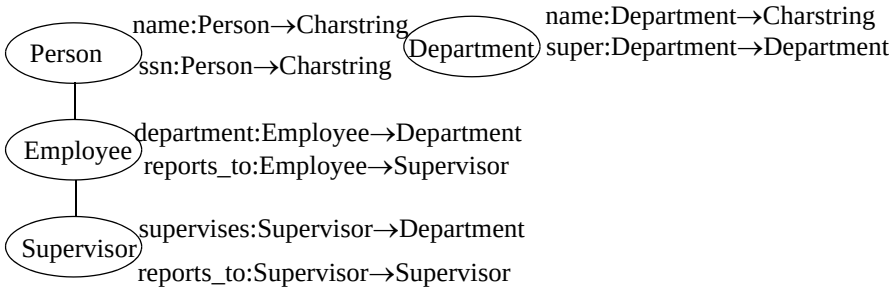


Figure 2.2: Database schema

In above figure there are four types, *Person, Employee, Supervisor* and *Department*. The *Supervisor* type inherits from *Employee* type which in turns inherits from the *Person* type. This means that the methods *name, ssn, Department, reports_to* and *supervises* are applicable to objects of the *Supervisor* type.

A type may inherit properties from more than one other type. This is called *multiple inheritance*. Multiple inheritance is not a mandatory characteristics of the OO model. Allowing multiple inheritance adds complexity to the model and problems arise when a type inherits operations with the same name from more than one of its supertypes. Multiple inheritance is, despite its drawbacks, a useful feature since it allows combination of types.

Closely related to inheritance is *polymorphism*, the ability to take several

forms. *Inclusion polymorphism* [15] describes subtypes and inheritance where an object can be viewed as belonging to many different types that need not be disjoint. This means that for any specification of an object of type *t* an object of type t_{sub} can be used. The polymorphisms experienced in an OO model are examined more closely in section 2.3.

2.2.6 Overriding, overloading and late binding

Another important characteristic of an OO database system is support for overriding, overloading and late binding. These notions all stem from the desire to have the same name to denote different operations where the type of the object is used to select which operation to apply when there are several operations with the same name.

The term overloading denotes when operation names have several implementations. Each implementation of an overloaded operation is called a *resolvent*. In the database schema (fig. 2.2) there are two overloaded names: *name* and *reports_to*. The function *name* is defined for arguments of *Person* type and is also defined for arguments of *Department* type. These two definitions share the same name but have different implementations.

In method name overloading a particular polymorphism is introduced into the model, *parametric polymorphism* [15]. Parametric polymorphism means that a method works uniformly over a range of types. These types normally exhibit some kind of common structure. In the OO model this common structure is found among types related through the subtype-supertype relation.

To exemplify overriding consider again the example (fig. 2.2) where the name *reports_to* is overloaded with definitions for arguments of *Employee* type and arguments of the *Supervisor* type. Note that the *reports_to* function definition of *Employee* type is inherited by the *Supervisor* type and that the *reports_to* function is also defined for the *Supervisor* type. In this case the name *reports_to* defined for employees is *overridden* by the definition of *reports_to* for supervisors. All subtypes of the *Supervisor* type inherits the definition of *reports_to* defined for supervisors.

Overriding occurs when an inherited name is given a new implementation in the type it is inherited by. This is why the overloaded method name *name* (fig. 2.2) is not overridden: there is no inherited definition that is given a new implementation for *name*.

Binding method names to implementations can either be done early or late. Early binding is when the binding can be done at *compile time*, whereas late binding is when the binding of the method name to an implementation of the name is done at *run time*. Late binding of methods is a consequence of inheritance in the type hierarchy and overriding of method names.

2.2.7 Computational completeness

Yet another desirable feature of an OO database is *computational completeness*. This means that any computable function should be expressible in the *data manipulation language* (*DML*) of the system. A good example of a DML

is SQL, The current SQL standard, SQL-92 [40], however, is not computationally complete.

Computational completeness means that the DML does not need any invocations of programs written in any language other than the DML. A DML is normally used for querying and updating data in a declarative fashion and extending it to become computationally complete will result in a large language with both declarative and procedural characteristics. Often systems have the ability to define complex computations in an auxiliary language, e.g. C++, and introduce these definitions into the query languages.

Computational completeness is outside the scope of this thesis and will not be addressed further in what follows.

2.2.8 Extensibility

The final mandatory OO characteristic mentioned in [4] is *extensibility*. Extensibility means that it must be possible to add user-defined types to the set of predefined types in the database. There must not be any distinction in usage between system-defined types and user-defined types. It is in [4] not a requirement that the set of collection types such as tuples, sets, lists, or bags are extensible.

2.3 Polymorphism

In an object-oriented system several different polymorphisms are experienced. A consequence of the inheritance in the type hierarchy is that a reference declared to denote objects of type t can denote objects of type t_{sub}. This is legal since all properties of type t are inherited by type t_{sub} so any operation on the reference declared as type t will be applicable to any object of type t_{sub}. This property is an instance of inclusion polymorphism [15].

This so-called coercion of the reference to denote objects of types other than the declared type must ensure *substitutability*. Substitutability means that any object of type t can be used in any context specifying an object of type t_{sup}. Consider the following figure:

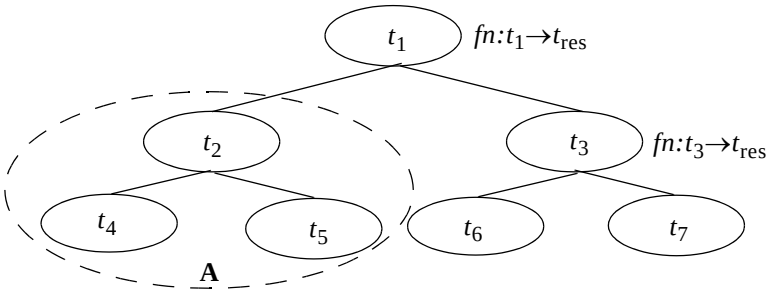


Figure 2.3: Example type hierarchy

Any reference declared to denote objects of type t_2 can, without violating the substitutability, denote objects of any type in the subtree marked **A** (fig. 2.3). Any other object of a type other than the types in **A** would violate the substitutability if denoted by the reference. This is so because t_2 would not be a supertype of the type of the object denoted by the reference.

In the example above (fig. 2.3) the method name *fn* is overloaded. The method *fn* is applicable to all types in the type hierarchy. Determining which of the two resolvents of the method name to use is based on the type of the object that the method is applied to.

In a polymorphic language the selection of which implementation of a method name to use at a particular application, *type resolution*, is a more complex task than in non polymorphic languages, e.g. C, Pascal.

For example: Let *ref* be a reference declared to denote objects of type t_2. Under the substitutability criterion the reference *ref* can denote objects of type t_2, t_4 or t_5.

A method invocation *fn*(*ref*) when *ref* is declared as type t_2, t_4 or t_5 will be resolved as an invocation of the resolvent $fn{:}t_1 \rightarrow t_{res}$ on *ref*.

If on the other hand the reference *ref* were declared to denote objects of type t_1, then *ref* can denote objects of type t_1 .. t_7. A method invocation *fn*(*ref*) will be resolved to $fn{:}t_1 \rightarrow t_{res}$ if *ref* denotes an object of type t_1, t_2, t_4 or t_5. and to $fn{:}t_3 \rightarrow t_{res}$ if *ref* denotes an object of type t_3, t_6 or t_7. This is an example of when type resolution cannot be carried out until runtime, i.e. late binding must be used.

2.4 Limitations of some object-oriented models

Recall from section 2.2.4 that types consist of an interface and an implementation. The interface consists of a set of operations together with their signatures and the implementation consists of the procedures that implement the operations of the interface. Now consider the following function with two arguments:

```
distance(Polygon p,Line l)->Length_unit
```

Example 7: Two argument function

In this example a function named *distance* that takes two arguments of type *Polygon* and *Line* is defined. The function returns the distance between its arguments.

In which type interface and type implementation should the function be defined? Can it be defined in the *Polygon* type, the *Line* type, both types or neither of them? Relations like this do not fit well into the OO model. This problem is dealt with in [24] where the argument is in favour of having this type of function outside any type, thus abandoning encapsulation.

Another limitation is message passing as a model for method invocation. Using message passing, there might be additional arguments supplied with the method. This corresponds to a function application where a function is applied to the object that receives the message plus any additional arguments. With message passing only the type of the object the message is sent to is used to resolve which message to invoke, hence overloading is limited to the first argument only, e.g. [10][18][28][53][57].

To exemplify the weakness of overloading on the first argument only, consider the two resolvents of the operation *distance* below (ex. 8) where *Segment*[9] is a subtype of *Line*.

```
distance(Polygon p,Line l)->Length_unit
distance(Polygon p,Segment s)->Length_unit
```

Example 8: Overloaded multi-methods

The message *distance* sent to a *polygon* with a *line* as argument cannot be resolved to a method at compile time since it is legal to coerce a reference to a *Line* into a *Segment* and different implementations of the method will be applied depending on the runtime binding.

In, for example, C++ [57] based systems this cannot be achieved since the first argument, i.e. the object which the message is sent to, is treated as special [3]. Consider the following C++ example.

9. Here, a segment is a sequence of n lines where the endpoint of the i:th line is equal to the startpoint of the i+1:th line for all i in [1..n-1].

```
class Line {
// Definition of the class
}

class Segment : public Line {
// Definition of the class
}

class Polygon {
public: int distance(Line li);
        int distance(Segment se);
}

int main(void){
Polygon *pl = new Polygon; //Create an inst. of Polygon
Line *li = new Line;
Segment *se = new Segment;
pl->distance(*li);
li=se;
pl->distance(*li); //li now denotes a segment but the
                   //method invoked is distance(line)
}
```

Example 9: Example of multi-methods in C++

In the example above, the reference *li* which is declared to denote objects of the *Line* type is set to denote an object of the *Segment* type. Since the *Segment* type is a subtype of the *Line* type this coercion is legal to perform. This type-casting is not reflected in the method that is invoked. The only way to achieve the desired behaviour is by programming the selection of the method manually.

2.5 The Functional data model

Another data model that has received some attention is the functional data model. This data model was presented by D.W. Shipman along with the DAPLEX language [50]. Here I will use the name DAPLEX to denote both the data model and the language. DAPLEX is an attempt to provide a database system interface which allows the user to model more conceptually.

The examples in this section use the syntax of the DAPLEX language. The examples are, however, kept simple so the examples should be understandable without any explanation of the DAPLEX language.

2.5.1 An overview of DAPLEX

In DAPLEX there are two major constructs, *entity* and *function*. An entity is meant to bear a one-to-one correspondence to real world objects. A DAPLEX function is a mapping from entities to a set of target entities. Functions can be either *single-valued* or *multi-valued*. Single-valued functions always returns a

singleton set of entities whereas a multi-valued function returns a set[10] of arbitrary cardinality. Multi-valued functions are initialized to return the empty set and single-valued functions must be initialized by the user.

Functions in DAPLEX can have zero, one or more arguments. Functions with zero arguments define entity types.

```
DECLARE Person() =>> ENTITY
```

Example 10: DAPLEX entity type definition

In the above example the entity type and function named *person* is defined. Only the entities returned by the function named *person* are of type *Person*, thus a call to the function returns the entire extent of the entity type *Person*. The function named *person* is a multi-valued function as indicated by a double arrow: =>>.

```
DECLARE name(Person) => STRING
```

Example 11: DAPLEX function definition

In this example a single-valued function is defined as indicated by a single arrow: =>. The defined function returns a singleton set containing an entity of the type STRING when applied to an entity of the *Person* type.

The entity types in DAPLEX can be ordered in subtype - supertype relationships.

```
DECLARE Student() =>> Person
```

Example 12: Subtype declaration

In above example the *Student* entity type is declared as a subtype of the *Person* entity type. The set of entities of the *Student* type will then be a subset of the set of entities of the *Person* type, thus the *person* function (ex. 10) will also return entities of the *Student* type. This means that the subtype - supertype relation of DAPLEX introduces inclusion polymorphism [15].

All functions defined to apply to entities of the *Person* type are inherited by entities of the *Student* type. Multiple inheritance is allowed in DAPLEX but with the restriction that if a type A inherits function *fn* from both type B and type C, then it must be the same *fn* that is inherited, thus an arbitrary selection of which one to choose can be made. To illustrate, consider the figure below.

10. A set in DAPLEX is a set in the mathematical sense, i.e. an unordered collection of elements without duplicates

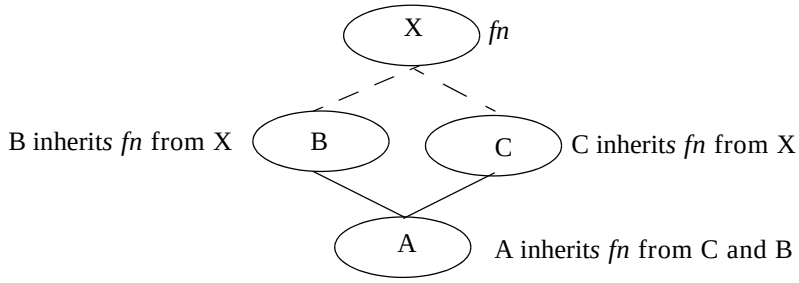


Figure 2.4: Multiple inheritance

In figure 2.4 a legal variant of multiple inheritance in DAPLEX is illustrated. The function *fn* which is inherited by the type A from both type B and type C is indeed exactly the same *fn* since it is inherited from type X by the types B and C.

Functions with more than one argument are legal in the DAPLEX data model and are defined in a similar way to functions with one argument.

```
DECLARE grade(Student,Course) => INTEGER
```

Example 13: Multiple argument function

The definition in example 13 defines a function named *grade* which takes two arguments and returns a singleton set with an entity of the *Integer* type.

Function inversions are declared explicitly as illustrated below.

```
DECLARE instructor(Course) => Instructor

DEFINE course(Instructor) =>> Course
 INVERSE OF instructor(Course)
```

Example 14: Function inversion

The above definitions define a function *instructor* that returns the instructor of a course. In addition, a function *course* is defined as an inverse of the function *instructor*. The *course* function returns the courses that a particular instructor is an instructor of. The keyword INVERSE OF is a syntactic construct to simplify the definition of inverses and the course function can be defined without using the INVERSE OF construct as:

```
DEFINE course(Instructor) =>> Course SUCH THAT
  Instructor = instructor(Course)
```

Example 15: DAPLEX function definition

2.5.2 Derived data

The *course* function (ex. 15) is defined in terms of another function, in this case the function named *instructor*. Hence, the function *course* is a *derived function*.

A derived function derives new properties from existing properties. Functions that are not derived, e.g. *instructor* (ex. 14), define properties of entity types. These properties can then be combined in various ways in derived functions to express new properties.

The use of derived data in DAPLEX is made *transparent* to the user, i.e. the user does not have to know whether the data is a stored property or derived data. Using derived data enhances the logical data independence, naturalness and usability of the database system.

2.5.3 Overloading

Function name overloading is permitted in DAPLEX. The resolution of the function resolvent is made based on the *role* of the entities the function is applied to. The role is the entity type that the entities are to be viewed as. Roles can always be determined by a static analysis of the data description, i.e. early binding.

Recall that when entity types are defined, a function with zero parameters is also defined. This function returns the set of all entities of a given type and is used when querying sets of data. By using the *person()* function all entities of the *Person* type are returned. Also recall that the entities of the *Student* type is a subset of the entities of the *Person* type whereby all entities of the *Student* type are contained in the set returned by the function *person()*.

By disallowing redefinition of functions in subtypes the problem of having late bound functions is eliminated. Hence, to override function definitions as described in section 2.2.6 is not meaningful in DAPLEX.

2.5.4 Type resolution

In DAPLEX overloaded functions can take more than one argument. Recall that message passing in the object-oriented model only uses the type of the first argument of the function to resolve which implementation to use. In DAPLEX all argument types participate in type resolution.

To illustrate type resolution consider the following hierarchy.

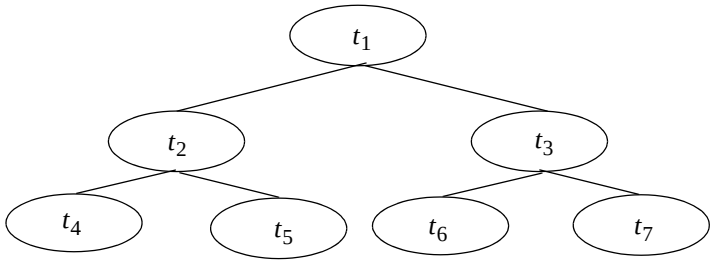


Figure 2.5: Type hierarchy

Assume there are two resolvents to the function named *fn*, namely $fn(t_4, t_1) \rightarrow t_{res}$ and $fn(t_2, t_3) \rightarrow t_{res}$, and an application of the function name *fn* as *fn*(*d*, *f*) where *d* is declared as being of type t_4 and *f* is declared as being of type t_6.

Which implementation of *fn* is the correct one to select? The resolvent $fn(t_4, t_1) \rightarrow t_{res}$ matches the first argument best and the resolvent $fn(t_2, t_3) \rightarrow t_{res}$ is the best match for the second argument. Since both arguments are equally important in DAPLEX, type resolution of the function application *fn*(*d*, *f*) will result in two equally applicable functions and such function application is considered ambiguous and thus illegal. Using message passing in an OO system the resolvent $fn(t_4, t_1) \rightarrow t_{res}$ will be selected.

2.5.5 Aggregate functions

An aggregate function is a function that applied on a collection of objects calculates some kind of property over the collection. Examples of aggregate functions are average, min, max and sum. Aggregate expressions are somewhat difficult to express in DAPLEX since it is set oriented, i.e. the result of any invocation is always filtered to only contain unique objects / values.

For example, calculating the average age of a set of persons cannot be accomplished simply by applying the aggregate function *average* to the resulting set of applying the function *age* to all persons since this set would not contain any duplicates.

This problem is solved by using a special operator, OVER, and iterating over the set of persons rather than over the set of the age of the persons.

2.5.6 Special operators in the DAPLEX language

One special operator in the language already encountered is the INVERSE OF which is a syntactic construct to simplify the definition of function inverses. Another special operator is the TRANSITIVE OF operator that allows the *transitive closure* of a function to be calculated.

A transitive closure is computed as follows. Assume the DAPLEX function

fn applied to entity *x* yields the entity set $\{e_1\ e_2\ e_3\}$. The transitive closure of *fn*(*x*) is then the union of $\{e_1\ e_2\ e_3\}$ and the transitive closure of $fn(e_1)$, $fn(e_2)$ and $fn(e_3)$.

To manipulate entity sets the operators INTERSECTION OF, UNION OF and DIFFERENCE OF are used. These operators can be used when creating new entity types:

```
DEFINE Surfing_students() =>>
  INTERSECTION OF Surfer, Student
```

Example 16: Intersection type

In this example a new entity type is the intersection of two existing entity types, *Surfer* and *Student* so that both are subtypes of the entity type *Person*. A new entity type is created which is a subtype of each of the entity types in the expression, thus the functions applicable are the ones defined or inherited by the new type.

The operator COMPOUND OF that can be used to specify pairs of related entities.

```
DEFINE Enrolment() =>>
  COMPOUND OF Student, course(Student)
```

Example 17: Pairs in DAPLEX

In example 17 a new entity type is defined which is the cartesian *product* of the operands, *Student* and the set of courses each student is associated with.

The DAPLEX model also contains *constraints* and *triggers* that add active behaviour to the data model. This functionality is, however, beyond the scope of this thesis and the reader is directed to [51] for further discussion about active database systems.

2.5.7 DAPLEX and object-orientation

The set of features of the DAPLEX data model and the object-oriented model are not completely disjoint, but there are features that are mandatory if a system is to be viewed as object-oriented which are not included in DAPLEX.

DAPLEX supports object identity[11] and includes the notion of type. Types are organized in a type hierarchy with inheritance, all in accordance with OO.

11. The DAPLEX paper [50] is rather vague as to whether entities have distinct identities as required according to the object-oriented database system manifesto [4]. The interpretation of this author is that entities in DAPLEX have distinct identities as required.

However, DAPLEX does not permit late binding, it lacks support for complex objects, no encapsulation is allowed, and the DAPLEX language is not computationally complete. The closest DAPLEX comes to complex object support is the COMPOUND-statement that creates a tuple-valued entity type.

In [24] a proposal of how the DAPLEX model can be extended to become object-oriented is presented. The proposal is mainly oriented towards encapsulation, better support for complex objects and extending the language to include recursion[12]. The proposal includes a language, OODAPLEX, where OODAPLEX denotes the entire proposal.

In OODAPLEX it is possible to override function definitions in subtypes, and thus late binding is required in the model.

OODAPLEX supports multi-argument functions and the problem of fitting them into the OO model is recognized, see section 2.4 above.

The rather strange requirement in DAPLEX that if functions with the same name are inherited several times via multiple inheritance then, it must be exactly the same function, is relaxed. Several different functions with the same name can be inherited but any conflicts that arise must be manually resolved by explicitly stating which resolvent to use.

The notion of *mutable* and *immutable types* is introduced in OODAPLEX. An immutable type is a type whose instances cannot be created, deleted or modified by users as opposed to the case for mutable types. Immutable types are the *primitive* types such as *Integer*, *Real*, *Boolean* and *String*.

Better support for complex objects is introduced: OODAPLEX supports, for example, the aggregate types set, tuple and multiset which are also treated as immutables, i.e. an instance of an aggregate type is identified by its members. Multisets are important since they often form the input to aggregate functions such as count, average, max or min., thus the awkward way of calculating aggregates in DAPLEX is removed.

12. How to extend the algebra to include recursion is not specified.

3 Overview of research systems and standards

In this section an overview of a few well-known object-oriented and extended relational database systems are presented with an emphasis on their data models and any special features they incorporate.

3.1 The ORION database system

The ORION [7][8][9][35] OO prototype system was built (using the Common Lisp language) as a support for an expert system, PROTEUS, in the late 1980s. The importance of supporting schema evolution and including a query language were recognized in the ORION project. Instead of having the applications navigate through the data by following references, a query language was to be provided. ORION also supports objects with multiple versions and has a framework for the management of different versions of objects.

3.1.1 Schema evolution in ORION

ORION includes a framework for managing an evolving schema. The schema changes allowed are as follows:

- Add, remove or change name of a type (ORION uses notion of Class).
- Insert or remove a superclass of an existing class.
- Add, remove or change a method of a class.
- Add, remove or change an attribute of a class.

In order to maintain the schema in a consistent state with respect to the types of changes listed above, a set of properties, *invariants of the class lattice*, are identified. These invariants must be preserved despite changes to the schema.

For some changes there may be more than one way to preserve the invariants and a set of rules have been developed which preserves these invariants [9]. These rules guide the preservation of the invariants when there is more than one way to preserve an invariant.

3.1.2 The ORION data model

In ORION classes are organised in a subtype-supertype hierarchy with inheritance. The hierarchy is rooted in the *Object* class from which all other classes are reachable. A subclass inherits both instance variables and methods from its

supertypes. For each class the following must hold.

- All instance variables and all methods defined in or inherited by a class must have distinct names.
- All instance variables and methods of a class must have a distinct identity.
- If several distinct instance variables or methods with the same name can be inherited, at least one must be inherited.

The first requirement listed above means that methods cannot be overloaded within a class[13], thus the *distance* example as described in section 2.4 cannot be realized in the ORION data model.

The second requirement is exactly the same as DAPLEX [50] also requires, as in the example of multiple inheritance in DAPLEX (fig. 2.4). A function can be inherited by a type from several supertypes only if it is exactly the same function inherited.

The last requirement says that two distinct instance variables or methods with the same name can not be inherited by a single class. This is achieved by inheriting one of them or inheriting both by renaming as required. This last requirement also means that names can be overridden in subclasses; rather than inheriting a method or variable, the one defined in the class is used.

Objects in ORION have distinct identity where each object can exist in several versions. Each version is identified by a version number, thus each distinct object and version of the object can be identified.

In order to be able to manipulate a set of objects as a single logical entity ORION provides *composite objects*. A composite object is an object with a hierarchy of *exclusive component objects*. The hierarchy of classes the objects belong to is a *composite object hierarchy*.

All objects, except the root, of a composite object are called *dependent objects*. A dependent object is referenced by another object through a *composite link*. A dependent object cannot exist on its own. If the object it depends on is deleted, the dependent object itself is deleted. A dependent object can only be referenced once within any composite object. The dependent object can however be referenced by another object through a non-composite reference.

The advantage of having composite objects in ORION is the performance improvement achieved by clustering composite objects on secondary storage. The reason for this is the likelihood of accessing some of the dependent objects whenever a root object is accessed. It is also argued that using the composite object as the unit for locking is advantageous for system performance.

One disadvantage of having composite objects is that deeply nested objects are costly to traverse since no fast access methods can be utilized. This problem is recognized in OODAPLEX [24] with regards to aggregate structures where it is advised not to build such structures.

The query model and query processing in ORION are relatively simple. Queries are expressed in a Lisp-like syntax and the execution of the query is mainly a traversal of the expression.

13. Which is the same as overloading on the first argument only.

Method invocation in ORION is based on the message-passing metaphor and the binding of method implementation to messages is always late. ORION is an pioneering system where query optimization and efficient query processing was seen as a future problem; instead the importance of providing a query language, support for schema evolution, composite objects and versions was recognized.

3.2 O_2

O_2 [25][33][36] is an object-oriented system developed by the *Altair* group to support business applications such as office automation and multimedia applications. The project began in 1986 and two prototypes were implemented; the first one was demonstrated in late 1987 and the second in mid-1989.

3.2.1 Objects and values

O_2 supports both objects and values, which is a side-step from pure OO where everything is viewed as an object. The structure of values are known by the user whereas the structure of the instances of a class are encapsulated within the class.

The reason to have both objects and values is that pure OO has the drawback that every time a complex value is needed, a new class has to be defined of which the complex value can be an instance. This leads to an undesirable growth of the class hierarchy.

In O_2 the objects are viewed as a pair consisting of an object identifier and a value. Besides simple atomic values O_2 supports set values, lists and tuples. Set and list values are defined as a finite collections of values. A tuple value is a collection of pairs of attributes and values defined by a partial function *fn*. $<a_1{:}id_1\ a_2{:}id_2\ \ldots\ a_n{:}id_n>$ is a tuple value where $fn(a_i)=id_i$ for all a_i in the tuple.

3.2.2 Types and classes

In O_2 a class is an association of a class name and a type. The type associated with a class describes the structure of the objects that are instances of the class. A type is either a *constructed type* or a *basic type* and all types have a type structure. A constructed type can be set or tuple structured.

The classes are organized in a subclass - superclass hierarchy with inheritance. An object which is an instance of a type *t* is also an instance of all supertypes to type *t*, thus inclusion polymorphism exists in the model. Multiple inheritance is allowed but any ambiguities that may arise when the same name is inherited from different supertypes must be resolved manually.

3.2.3 Methods

Every class has a set of attached methods that are used to manipulate the instances of the class. Only the method signatures are visible to others, not the implementation, i.e. O_2 employs encapsulation. Methods are inherited from

superclass to subclass. If a method is defined in several superclasses of a particular class, then the method that is defined in the most specific superclass is the one that is inherited.

It is possible to override inherited methods in subclasses so late binding is required. Multi-argument methods are supported but the type of the first argument is used to determine to which type the method is attached and it is only possible to overload on the first argument, which is equivalent to OO message passing.

In O_2 messages are replaced by function calls whenever possible. In unresolved cases, method name resolution is accomplished by the executing code in an ad hoc manner [61]. This means that whenever possible early binding is used.

3.3 The EXODUS project

The database project, EXODUS [16][44][60], which started in 1986 at the University of Wisconsin was mainly oriented towards problems of data structuring and processing requirements of emerging application areas. In the EXODUS system a versatile storage manager and a general purpose type system have been implemented as well as a set of tools that help the database programmer to develop new database system software. For example, tools are provided that generate a query optimizer from a rule-based description of the possible algebra transformations.

As the EXODUS project progressed the problem of designing a data model and a query language started to emerge. The data model of EXODUS is known as the EXTRA data model and the query language is known as EXCESS where EXTRA is an abbreviation for *EXtensible Types for Relations and Attributes* and EXCESS stands for *EXtensible Calculus for Entity and Set Support*. Today the EXODUS project has terminated and the SHORE (Scalable heterogeneous object repository)[17] project has replaced it.

3.3.1 Type hierarchy

In the EXTRA data model the types are organized in a hierarchy with inheritance. Multiple inheritance is allowed, but in the case of conflict either the same attribute or the function is inherited from several supertypes, see section 2.5.1 (fig. 2.4), or the conflict is resolved by the programmer by explicitly stating which implementation of the name in conflict to use. A subtype inherits all of its supertype's properties. In subtypes it is possible to redefine inherited properties, thus late binding is required.

The types are either *base types* or *tuple types*. There are a number of *predefined base types* such as *Integer, Real* or *Character*. and new ones can be defined. Base types are types whose instances are identified by their values and the value of an instance of a base type cannot be deleted or changed. Base types correspond to the immutable types of OODAPLEX [24]. In EXTRA it is possible for the user to define new base types by adding abstract data types (*ADT*s).

ADTs are written in the E programming language[14] [44] whereas the schema types (tuple types) are written in the EXCESS language.

The tuple types are created solely by the programmer to capture the behaviour of the modelled entity.

```
DEFINE TYPE Person:
(
    ssn : Int4,
   name : Char[]
)
```

Example 18: Tuple type definition

In this example, a tuple type named *Person* is defined. The type contains two attributes, *ssn* and *name*. The tuple types of EXTRA correspond to the tuple structured types in O_2 [36].

3.3.2 Complex objects

The EXTRA data model supports the complex objects tuple, array and set. Arrays can be of both fixed size and variable size where in variable sized arrays elements can be inserted and deleted anywhere in the array. Arrays can be declared to hold data of any type. The set data type can also be declared to hold data of any type. If a set or array has been declared to hold data of type t then any data of type t_{sub} can be stored in the set or array.

In order to further support complex objects, values that are references to other object are allowed, thus object graph structures can be built.

In EXTRA three different kinds of value semantics exist. It is up to the database programmer which value semantics to use by declaring the value of an attribute as: `OWN`, `REF` or `OWN REF`. To illustrate, consider the following examples.

```
DEFINE TYPE Parent:
(
children : {OWN Person}
)
INHERITS Person
```

Example 19: Declaration of value as OWN

In above example the attribute *children* is declared to denote a set of *Person* owned by the *Parent* object. The elements in the set are values rather than objects, thus the elements do not have any OID and cannot be referenced elsewhere in the database. If the owner of an object is deleted, the object itself is deleted, and owned objects cannot exist on their own. If a *Parent* is deleted from the database so are the children.

14. The E language is an extension to C++[57]

```
DEFINE TYPE Parent:
(
children : {OWN REF Person}
)
INHERITS Person
```

Example 20: Declaration of value as OWN REF

If the declaration of the *children* attribute is `OWN REF` (ex. 20) the value is a set of references to objects of the *Person* data type. Each *Person* referenced can now be referenced from elsewhere in the database. The objects are still owned, thus the deletion semantics remains. A *Person* instance referenced in a *children* set of a *Parent* instance cannot be referenced in another *children* set simultaneously. This corresponds to the complex objects in the ORION data model [35].

```
DEFINE TYPE Parent:
(
   children : {REF Person}
)
INHERITS Person
```

Example 21: Declaration of value as REF

By declaring the *children* attribute as a set of references to *Person* instances (ex. 21), referenced objects are not automatically deleted whenever a referencing object is deleted.

3.3.3 Functions

The value of an attribute may be derived from the database whenever required. Such an attribute is called a *derived attribute*. A derived attribute is defined by an EXCESS query. A derived attribute in EXTRA is similar to a derived function in DAPLEX [50]. Functions may have several arguments and are associated with the declared type of the first argument; thus the invocation of a function is analogous to message passing in object-oriented languages. Late binding is supported but only for the first argument of the function in a similar way to functions declared as virtual in C++ [57]. This means that in an EXCESS/EXTRA application implementing the *distance* example, described in section 2.4, the type dispatch have to be explicitly programmed as in any C++ implementation of that example.

The EXCESS data model only supports set and not multiset and the syntax of the EXTRA language when aggregate functions are present is somewhat awkward, analogously to DAPLEX.

When querying the database using the EXCESS language, it is possible to range over any named persistent set of objects, an array, or over a sub-range of

an array which, for example, enables any user to query the set of *children* of a given *parent* in a simple way.

In [60] an approach to managing late bound function calls in the execution plan is presented which deals with how to perform a runtime dispatch and the difficulty of optimizing late bound functions. This approach will be evaluated in chapter 5.

3.4 The object database standard: ODMG-93

The ODMG standard proposal [18] is a product of the *Object Database Management Group* (ODMG) whose members come from a number of commercial companies. ODMG is a portability standard in that it guarantees that a compliant application can easily be ported from one system to another [6]. ODMG defines an ODBMS [19] as a DBMS that integrates database capabilities with object-oriented programming language capabilities. In an ODBMS database objects appear as programming language objects.

The ODMG-93 standard proposal targets systems where the application and data definitions are compiled and linked together into one application where the application program and the database manager share the same type system and execution environment. Systems that support applications with this type of architecture are called *Database programming languages* in [18]. The concepts are illustrated in the figure below:

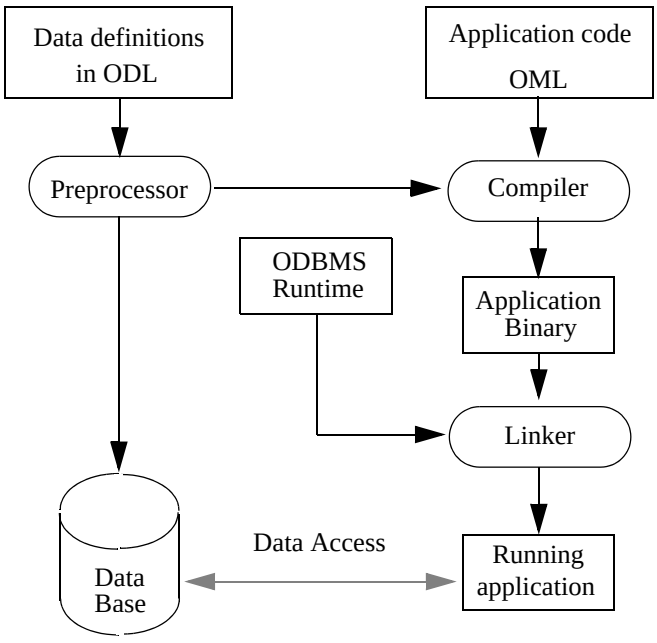


Figure 3.1: ODMG target architecture

3.4.1 The ODMG target architecture

To clarify the ODMG standard proposal this section will provide an overview of the architecture that the ODMG standard proposal is targeted at (fig. 3.1).

All data definitions are expressed using an *Object Definition Language* (ODL). The ODL is used only to define types and their properties, i.e. signatures for operations, relationships and attributes. It is not used for defining the implementation of the operations. The ODL is independent of the syntax of the programming language used. A data definition in ODL is *portable*, i.e. it can be supported by any ODMG-compliant ODBMS. The application implementation is written in an *Object Manipulation Language* (OML). A typical OML is C++ extended to provide transactions and query invocation capabilities. Queries are expressed in an Object Query Language (OQL) and are invoked from the OML.

The disadvantage of this architecture appears in the restrictions placed on the system by the implementation language[15]. The systems use the method dispatch that the implementation language provides, they use the same argument-passing and compile time rules. Thus the optimizations that can be performed is limited, the incorporation multi-functions to provide a system capable of expressing the distance example, section 2.4, is a challenge and efficient management of late bound functions is also a challenge.

3.4.2 The ODMG-93 data model

Types are organized in an inheritance hierarchy where subtypes inherit all properties of their supertypes. In a subtype it is possible to add new properties or to redefine inherited properties.

Objects have distinct identity and are instances of types. The model supports substitutability, i.e. any object of type t can be used wherever an object of type t or t_{sup} is expected. The extent of type t is a subset of the extent of type t_{sup}, thus the model supports inclusion polymorphism. Extents are not maintained by the system, thus the application must explicitly maintain the extents itself.

A type consists of an interface and one or more implementations where the interface is public and the implementation is hidden. The interface consists of *operation signatures*, *attribute signatures* and *relationship signatures*.

The operation signature is the name of the operation annotated with the name and type of any arguments and return value and the names of any *exceptions*[16] that can be raised. Operation names may be overloaded, but only the type of the first argument is used in type resolution although several arguments are allowed. Each operation name must be unique within the type definition. This means that the distance example described in section 2.4 cannot be modelled in an elegant way.

An attribute signature is the name and type of any attributes. An attribute can only be declared to denote literal values, e.g. *Integer, Real* or *Character*.

15. C++
16. An exception signals an error condition. An exception is raised when an error is encountered.

Finally, a relationship signature specifies any binary relationships that the instances of the type can participate in. The signature defines the type of the other object or set of objects involved and the *traversal function*. A traversal function is used to refer to the related object or set of objects. Traversal functions are defined for traversing the related objects in one direction. Traversal in the opposite direction is performed by the *inverse traversal function*.

```
interface Student
{
   takes: Set<course> INVERSE Course::is_taken_by
}

interface Course
{
   is_taken_by: Set<Student> INVERSE Student::takes
}
```

Example 22: Inverse traversal function in ODMG-93

In example 22, interfaces to the types *Student* and *Course* are defined. In the interface to the *Student* data type a relationship to a set of *Courses* is defined. The relationship has a traversal function named *takes* that has an inverse traversal function named *is_taken_by* defined in the interface to the *Course* type. Note that relationships are defined between mutable types.

Four collection types are supported: *set*, *bag*, *list* and *array* where sets are unordered duplicate-free collections. Bags are unordered collections that may contain duplicates. Lists are ordered collections that may contain duplicates. Finally, the model allows arrays, whose length must be specified at creation. The main difference between lists and arrays is that arrays cannot be traversed by fetching the next element. An array must be traversed by retrieving the element at a given position. Lists can be traversed in either way.

Collections are typed, i.e. they can only contain elements of a given type. Since substitutability is supported, any element of type t_{sub} can occur in a collection of type t. Each of the collection types has an immutable variant where only creation of instances is allowed. Immutable collections are identified by their members, whereas mutable collections retain their identity after insertion or deletion of elements. The elements in collections are retrieved by using an iterator that traverses the collection element by element.

Extents are predicate-defined collections where the predicate checks that the members are of a given type.

One important use of collection types is to maintain the type extents.

3.4.3 OQL

OQL [6][18] is a declarative high-level query language defined in the ODMG standard to facilitate interactive ad hoc queries. OQL uses the same type system as the application programming language, i.e. C++. Thus C++ elements can be passed as parameters and the result can be used directly in C++. Hence, there is

no *impedance mismatch*[17] between OQL and C++.

The ODMG standard defines the syntax and deals with how to bind a query language to a programming language. The ODMG approach is to support a loose coupling where query functions are introduced that take strings containing queries as their argument. These queries are optimized at runtime. Invocation of an OQL query is carried out by calling the *oql-function*[18] defined to take a variable and a string. The result of the OQL query is bound to the variable. The OQL query is contained in the string argument to the oql function.

Since methods called from OQL are allowed to have side effects and the cost of invocation is unknown, thus optimization of OQL queries is limited. For example reordering is not possible.

3.5 SQL-3 Standard proposal draft

A standard proposal under construction is SQL-3. The SQL-3 standard proposal incorporates object-oriented features into the language. Since the standard is under construction, this section is based on a tutorial given at the 1995 ACM SIGMOD conference by Nelson M. Mattos, IBM. The following extensions are being proposed at present:

- Type system extended with ADTs
- ADTs organized in a subtype - supertype hierarchy with inheritance
- Substitutability among instances of ADTs
- Function overloading permitted
- Support for multi-functions where type dispatch is based on all argument types
- Support for late binding
- Support for roles, i.e. an instance of type *t* can be treated as an instance of any supertype of *t*.
- OID support of instances of ADTs
- Parameterized types

It is interesting to note that the need for multi-methods is recognized as a desirable feature in a database query language. Recall the distance example (ex. 9) which cannot be given a simple implementation unless multi-methods (multi-functions) are supported by the language. Also note that support for late binding is required; thus the approach presented in this thesis of how to manage late binding in a system is highly relevant to any system that strives to comply with the SQL-3 standard.

17. Impedance mismatch occurs when two different type systems with different expressiveness in different environments must exchange data.
18. This is a function in the host language that takes as argument a string containing an OQL-query.

3.6 POSTGRES

POSTGRES [45][56] is an extended relational system where the basic relational model has been extended with inheritance and support for complex objects.

Tables can be declared to inherit attributes from other tables as:

```
CREATE Person (name = Charstring[25],
               birthdate = date)

CREATE Student (programme = Charstring[20])
INHERITS (Person)
```

Example 23: Table declarations in POSTGRES

There are two ways to query a particular table in POSTGRES. Either the query spans a certain table and all tables are declared to inherit from the queried table or the query only spans over the queried table only.

```
RETRIEVE (p.name)                RETRIEVE (p.name)
FROM p IN Person*                FROM p IN Person
WHERE birthdate = 080594         WHERE birthdate = 080594
```

Example 24: Queries in POSTGRES

In the left query above the *-operator is used to specify that the result of the query should be the result of executing the query over the *Person* table and, transitively, union all tables declared to inherit from the *Perso*n table. In the query to the right the *-operator is omitted, which means that the query must only be executed on the *Person* table.

New atomic data types can be introduced by defining ADTs and a set of operations associated with the new type as

```
DEFINE TYPE box IS (Internal Length = 16,
                    InputProc = ChartoBox,
                    OutputProc = BoxtoChar)


DEFINE OPERATOR equal(box, box) RETURNS boolean IS
      (proc = box_equal, Precedence = 2,
       Associativity = “Left” )
```

Example 25: POSTGRES ADT definition

An ADT is defined in POSTGRES by the DEFINE TYPE statement (ex. 25) where the type name, size, input and output methods are declared. The opera-

tions on the ADT are created by the DEFINE OPERATOR statement where input and output types are declared and the procedural implementation of the type is specified. The procedural specification is implemented in a regular programming language, e.g. C, and is then associated with a name. In example 25 the ADT function named equal is associated with the procedural specification named box_equal.

Table attributes can be typed with any ADT and the operations on the ADT can be used in queries. Procedures operating on tuples from tables can be defined and their names can be overridden in subtypes. A type-resolving scheme that handles multiple inheritance by selecting the first applicable procedure has been suggested.

It has not been explicitly stated what happens if the *-operator is used in combination with overriding, e.g., if procedures will be bound early or late. one can assume that late binding is not supported by POSTGRES.

In POSTGRES, a few useful features have been incorporated which resembles OO features. However the relational style of modelling data remains, i.e. accommodating data in tables.

3.7 Summary

All systems except POSTGRES reviewed in this chapter allow for late binding but few of them address the problem of how to provide efficient management of late bound function calls.

Both standard proposals reviewed in this chapter, ODMG-93 and SQL-3, realize that it is desirable to have late binding, so the work presented in this thesis is relevant to both the standards and to any system which incorporates late binding. Multi-functions is part of the current SQL-3 proposal. Our solution to the management of late bound functions is presented in chapter 5 and the incorporation of multi-functions into an object-oriented system is presented in section 6.

4 Type resolution and invertibility

In this chapter type resolution and invertibility in a basic object-oriented data model is presented. A basic object-oriented model uses message passing as function invocation. The data model presented is strongly influenced by the functional data model DAPLEX [50] and by the Iris [28] data model with the OSQL [38] query language. OSQL (Object SQL developed at HP Lab) is an object-oriented extension of SQL.

The data model includes a query language AMOSQL.v0[19] [34] which is an extension of OSQL with rules [46][51]. This basic model will later in this thesis be used as platform for describing our approach to efficient management of late bound functions (chapter 5). In chapter 6 it will be extended with *multi-functions* to become an extended object-oriented model.

4.1 Functions, types and objects

The relations between functions, types and objects can be described as in the following figure:

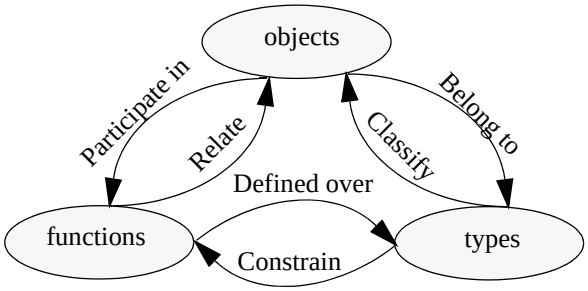


Figure 4.1: Functions, types and objects

All objects are instances to some type and types are used to classify objects. Side-effect-free functions that denote mappings from objects to objects relate objects. Functions are constrained to accept only objects that are instances of the declared argument types or any subtype of the declared argument type.

19. In this section it will be referred to AMOSQL.

Types inherit all properties from their supertypes and any conflict due to multiple inheritance must be resolved by the user. Inclusion polymorphism is included in the model and it is possible to override properties in subtypes, thus late binding is required in the model. Types are created along with their properties as:

```
CREATE TYPE Person SUBTYPE OF Mammal
PROPERTIES (name Charstring, age Integer);
```

Example 26: Create type statement in AMOSQL

If the *SUBTYPE OF* part is omitted in the type definition, the defined type will become a subtype of the system-defined type *Usertypeobject* in the type hierarchy. In the *PROPERTIES* part of the statement a shorthand way of creating *stored functions*[20] is provided. In example 26 two stored functions are created, the function named *name* and the function named *age*. Additional stored functions can be added later.

The data model contains both mutable objects and immutable objects. Mutable objects are identified by their OIDs and immutable objects are identified by their values, i.e. *literal objects*. A literal is always assumed to exist and its value cannot be changed. Examples of literals are the types *Integer*, *Real* and *Character*. A literal object can be either simple or complex. The complex literals *Bag*, *Vector*, *List* and *Tuple* are supported. New objects are created as follows:

```
CREATE Person (name, age) INSTANCES
('Ralph',12), ('Rodney', 7), ('Rick', 11);
```

Example 27: Create objects in AMOSQL

In this example three objects of the *Person* type are created and given values for the attributes *name* and *age*.

The notion of a function is almost consistent with DAPLEX [50] (chapter 2.5) which supports both stored and derived functions. A difference is that the zero argument extent function in DAPLEX does not exist in this model[21]. Stored functions can be both literal and object valued. Both simple and complex literals are supported. The notion of a stored function replaces the concepts of *relation* and *attribute* in the ODMG-93 model [18].

```
CREATE FUNCTION married_to (Person p) -> Person AS STORED;
```

Example 28: Create stored function in AMOSQL

20. A stored function stores properties of objects.
21. A finite extent of type *t* can be generated by a function *typesof*(*type t*).

In example 28 an additional stored function, *married_to*, is added to the set of stored functions applicable to objects of the *Person* type. The function *married_to* is declared to denote an object of the *Person* type, which corresponds to a relationship in ODMG-93.

Functions can be overridden in subtypes and late binding is required. Function invocation is based on the OO message passing style where the type of the first argument is used to resolve which implementation to use, although several arguments are allowed.

Resolvent names are created by annotating the function name with the name of the first argument type. For example, the resolvent name of the function defined above (ex. 28) is *Person.married_to*. The *signature* of the same function resolvent is *married_to*:*Person*→*Person*. The signature is the function name annotated with the types of all its arguments and result types.

4.2 Queries

Queries are expressed in AMOSQL as

```
SELECT name(p) FOR EACH Person p
WHERE age(p)>3;
```

Example 29: Query in AMOSQL

In above query the name of all objects of the *Person* type is selected if the age is greater than three. The query contains one *range variable*, *p*, which states that all objects of the *Person* type and subtypes of the *Person* type are considered, since inclusion polymorphism is part of the model.

A query can select several properties as:

```
SELECT name(p), age(p) FOR EACH Person p, Person q
WHERE age(q)>30 and married_to(p)=q;
```

Example 30: Multi query in AMOSQL

In this query the name and age of all persons who are married to a person older than 30 are selected.

Queries can be used to define derived functions

```
CREATE FUNCTION adultp(Person p)-> Boolean AS
SELECT true FOR EACH Person q
WHERE age(q)>30 and married_to(p)=q;
```

Example 31: Derived function in AMOSQL

In above example the derived function *adultp* is created which, given an object of the *Person* type, returns *true* if the argument object is defined as being married to a person older than 30.

At time of creation the derived function is compiled and optimized, thus invocation of the function as an ad hoc query will be substantially much faster than invocation of an entirely new query that has to be parsed, type-checked and optimized.

4.3 Static and dynamic types

This section presents a more precise approach to subtypes and inheritance. Subtyping will be given a definition based on type conformance [11] and the notion of static and dynamic type [52][53] will be introduced. This framework will be used in chapter 5 where type resolution and resolution of late binding will be further investigated.

Definition 1: *Type conformance*

A type t_i conforms to type t_j if

i t_i provides at least the operations of t_j

ii The result types of all operations of t_i conform to the result types of the corresponding operations of t_j.

iii The argument types other than the first argument type of the operations of type t_j conform to the argument types of the corresponding operations of type t_i.

The definition of type conformance can then be used to constrain what is meant by a subtype.

Definition 2: *Subtype*

A type t_i is a subtype of type t_j if t_i conforms with t_j.
$t_i < t_j$ denotes 't_i is subtype of type t_j'.
$t_i \leq t_j$ denotes $t_i < t_j$ or $t_i = t_j$
t_{sub} denotes t or an arbitrary subtype of type t, t_{sup} denotes t or an arbitrary supertype to type t.

In a subtype it is possible change the implementation of an inherited operation and to add behaviour. Adding behaviour can be done freely but when changing the implementation of an inherited method, the new implementation must not violate type safety which ensures that type errors will not occur when a method is late bound.

An instance of a type is also an instance of any supertype of that type. This property results in the notion of *substitutability* [49].

Definition 3: *Substitutability*

An object, *o*, of type *t* can be used in any context specifying an object of type t_{sup}.

Substitutability means that whenever an object of a certain type is expected, objects that are instances of any subtype can occur. Under substitutability the set of all objects, *extent*, of a certain type is defined as:

Definition 4: *Extent*

The extent of a type t is the set of objects $\boldsymbol{O}=\{o_1 \ldots o_n\}$ where for each $o \in \boldsymbol{O}$ one of the following must hold
i $typeof(o)=t$ or
ii $typeof(o)=t_{sub}$
Where *typeof* is returning the most specific type of the object. The extent of a type t is denoted $ext(t)$.

This definition implies that the extent of a type contains all objects that are instances of any subtype of the type. To exemplify, consider the following example database schema.

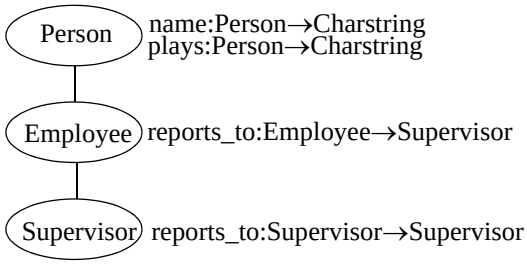


Figure 4.2: Example database schema

Assume that the database schema (fig. 4.2) is populated with objects of the types *Person*, *Employee*, and of the *Supervisor* type:

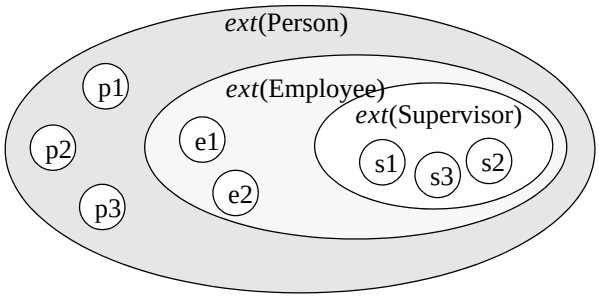


Figure 4.3: Database extents

The names of all persons are sought in the populated database. This is expressed in AMOSQL as:

```
SELECT name(p) FOR EACH Person p;
```

Example 32: Simple query over persona

The range variable, *p*, in above query ranges over the entire extent of the *Person* type which is equal to the set {p1, p2, p3, e1, e2, s1, s2, s3}. Thus the result of the query is the set obtained by applying *Person.name* to every element of the set.

Consider another query over the same database (fig. 4.2)

```
SELECT reports_to(e) FOR EACH Employee e;
```

Example 33: Simple range query

This query will range over the extent of the *Employee* type which is equal to the set {e1, e2, s1, s2, s3}. The result of this query is the set obtained by applying the resolvent *Employee.reports_to* to the objects in {e1, e2} union the result of applying the resolvent *Supervisor.reports_to* to the objects in {s1, s2, s3}.

In the previous example the result was the union result of applying two different resolvents on disjoint subsets of the set the query ranged over. In order to decide which resolvent to apply to a particular object, the notions of *static type* and *dynamic type* [53] are required.

Definition 5: *Static type*

The static type of a reference, *ref*, is the declared type of the reference. ***S***(*ref*) denotes the static type of a reference *ref*.

For example, the range variable *p* (ex. 29) has as its static type the *Person* type and for the resolvent *Person.name* (fig. 4.2) the static type is *Person*. In a strongly typed language it is always possible to determine the static type of a reference at compile time.

Definition 6: *Dynamic type*

The dynamic type of a reference, *ref*, is the type of the object referenced at runtime. ***D***(*ref*) denotes the dynamic type of a reference *ref*.

Constrained under substitutability the dynamic type of a reference can be the static type of the reference or any subtype of this type.

In the previous example (ex. 33), different resolvents were applied depending on which object was referenced, i.e. the dynamic type of the range variable

was used to select which resolvent to apply.

Closely related to the static and dynamic type is the *dynamic type set*.

Definition 7: *Dynamic type set*

The dynamic type set of a type, *t*, is the set of all possible dynamic types of a reference with static type *t*. The dynamic type set of a type *t* is denoted $\boldsymbol{T}(t)$.

The dynamic type set is the set of types a reference, constrained by substitutability, can denote instances of, i.e. $\boldsymbol{D}(ref) \in \boldsymbol{T}(\boldsymbol{S}(ref))$. Another view of the dynamic type set of a type *t* is the set of types in the subtree rooted at type *t*.

In the example schema (fig. 4.2) the dynamic type set of a reference with static type *Person* is the set of types {*Person*, *Employee*, *Supervisor*}.

4.4 Type resolution

Type resolution, i.e. selecting a resolvent of a function call to apply, is carried out at compile time[22]. The general rule in type resolution is to select the most specific resolvent of a particular function with a static type which is not more specific than the static type of the first argument to the function. To resolve which resolvent to select for an application, *fn*(*a*), there exists a function, *resolvent* [30], which given a function name and a type returns the applicable resolvent.

Definition 8: *resolvent*: $\boldsymbol{NM} \times \boldsymbol{Tp} \rightarrow \boldsymbol{FNM}$

$\boldsymbol{NM}$ is the set of all names, $\boldsymbol{Tp}$ is the set of all types and $\boldsymbol{FNM}$ is the set of all resolvent names, a subset of $\boldsymbol{NM}$.
If $resolvent(fn, \boldsymbol{S}(arg)) = t_j.fn$, then there must not exist any other resolvent, $t_k.fn$, such that:
i $\boldsymbol{S}(arg) \leq \boldsymbol{S}(t_k.fn)$ and
ii $\boldsymbol{S}(t_k.fn) < \boldsymbol{S}(t_j.fn)$
If no resolvent, $t_j.fn$, exists the result is the special value NIL.

Definition 8 ensures that the returned resolvent is the most specific resolvent defined for a supertype or equal to the type of the first argument the function is applied to.

Also note that the domain $\boldsymbol{FNM}$ is a subset of $\boldsymbol{NM}$. Hence it is possible to use elements from $\boldsymbol{FNM}$ where elements from $\boldsymbol{NM}$ are expected to determine if a certain resolvent is applicable to some argument. To exemplify, consider:

22. Even when functions are late bound, the type resolution that detects the requirement of late binding is performed at compile time.

```
SELECT name(e) FOR EACH Employee e
WHERE plays(e)='Bass';
```

Example 34: Simple query

To resolve which implementation of the function *name* to use in the above example the *resolvent* function is called as: *resolvent*(*name*,***S***(*e*)). Given the example hierarchy (fig. 4.2) evaluates to *Person.name*. The implementation of the function named *plays* is obtained by *resolvent*(*plays*,***S***(*e*)) which evaluates to *Person.plays*.

4.5 Invertibility

A highly desirable feature is the ability to use the functions of a database schema in the inverse direction. For clarification, consider the *name* function (fig. 4.2) used as:

```
SELECT p FOR EACH Person p
WHERE name(p)='Ralph';
```

Example 35: Simple query

In the query above the result of the *name* function is known and the arguments that are mapped by the *name* function onto the character string 'Ralph' are sought.

One way to execute this query is to scan the entire extent of the *Person* type and apply the *name* function to select the objects with 'Ralph' as their name.

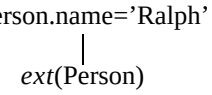


Figure 4.4: Query tree

In this query tree (recall the example algebra from section 1.4.2) the extent of the *Person* type, *ext*(*Person*), is scanned and to each object, *o*, the selection condition *name*(*o*)='Ralph' is applied to accept only those objects whose name property has the value 'Ralph'.

Another way to execute the query is to use the inverse of the *Person.name* resolvent, $Person.name^{-1}$, as:

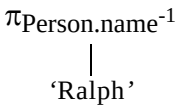


Figure 4.5: Query tree with inverted function

Note that the algebra operator select (fig. 4.4) is replaced by a form of project (defined in section 1.4.2 to be analogous to a function application).

By being able to use an inverted function, a scan of an extent has been eliminated. If the resolvent *Person.name*$^{-1}$ is given an efficient implementation, e.g. by using a secondary index, a dramatic performance improvement has been achieved.

4.5.1 Inverse of stored, derived and foreign functions

There are three function types: stored function, derived functions [50](chapter 2.5), and *foreign functions* [37]. Foreign functions are defined using an auxiliary programming language such as C, C++ or Lisp and then introduced in the database query language by associating the auxiliary definitions with a resolvent name.

To make foreign functions invertible their inverse must be explicitly defined. To exemplify, consider the following figure:

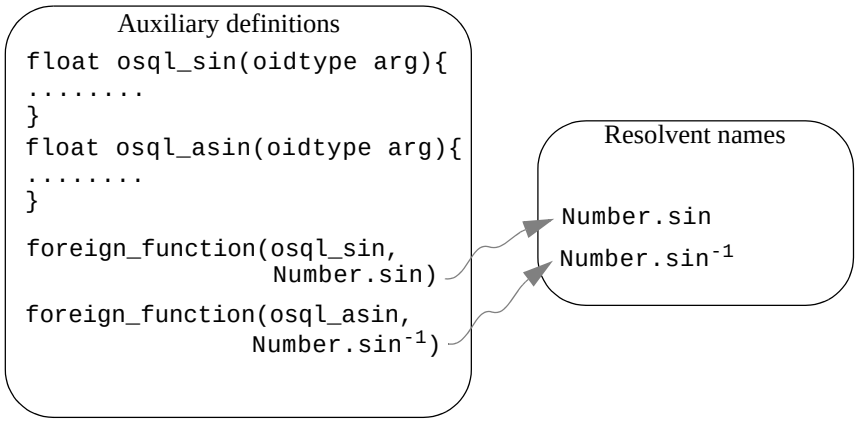


Figure 4.6: Creating an invertible foreign function

In this figure two functions, *osql_sin* and *osql_asin*, are defined in some auxiliary programming language. These functions are bound to the query-language function resolvents, *Number.sin* and *Number.sin*$^{-1}$, and the resolvents become visible to the query processor. The inverse remains hidden from the user but the non-inverted, resolvent *Number.sin*, is visible in the query language.

For stored functions the inverse is created by the system when the function is defined. When a stored function is created, a data structure is created which enables fast access to data, for example a hash table. When the stored function is populated (by using *add* or *set* [34]) data is inserted into the data structure. The internals of a stored function is some kind of access mechanism that utilizes any fast access paths available. The inverse of a stored function is an access strategy to data through an object typed as the result type of the stored function. To make the execution of inverse of a stored function efficient, an index on the result should be created.

To exemplify inverses of stored functions consider the following figure:

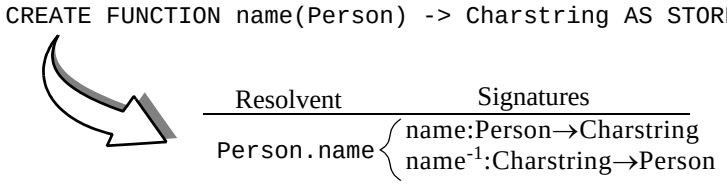


Figure 4.7: Creating the inverse of a stored function

In above figure a resolvent, *Person.name*, is created. This resolvent is visible in the query language. Along with the resolvent its inverse is created, *Person.name*$^{-1}$; the inverse is, however, not visible in the query language. The inverse is invoked by the system. This is because the user must be relived from having to explicitly specify when to use the inverse and the declarativness of the language would be lost.

The inverses of derived functions are inferred by the system. By having access to the body of a derived function, its inverse can be generated by the system by, for example, using the inverses of the functions in its body. To exemplify, consider the following database schema where indexes are created on the result of all stored functions.

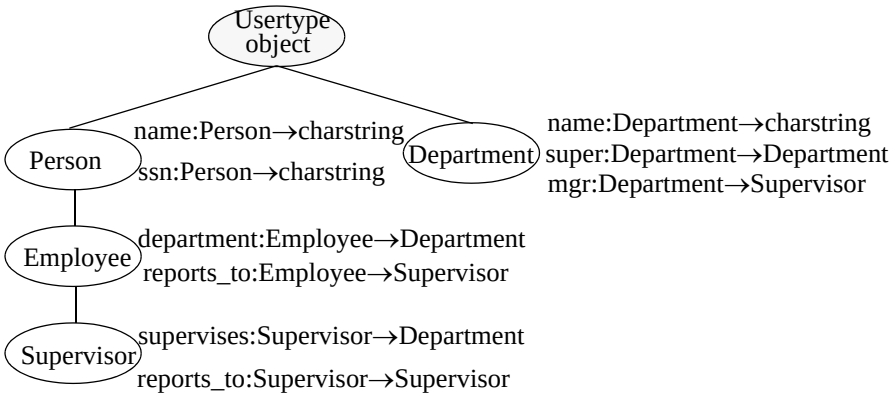


Figure 4.8: Database schema

For example, *Employee.reports_to* is defined as:

```
CREATE FUNCTION reports_to(Employee e)->Supervisor AS
SELECT mgr(department(e));
```

Example 36: Definition of Employee.reports_to

First, the query is rewritten to eliminate nested function applications. The body of *Employee.reports_to* is rewritten to become:

```
Employee.reports_to(Employee e)->Supervisor
SELECT _G2
WHERE  _G1=Employee.department(e)and
       _G2=Department.mgr(_G1);
```

Example 37: Body of derived function

Both *Department.mgr* and *Employee.department* are stored functions, so their inverses are created by the system. The signatures of the resolvent inverses are:

Resolvent inverse	Signature of inverse
Employee.reports_to^{-1}	*reports_to*$^{-1}$*: Supervisor→Employee*
Department.mgr^{-1}	*mgr*$^{-1}$*:Supervisor→Department*
Employee.department^{-1}	*department*$^{-1}$*:Department→Employee*

Figure 4.9: Signatures of resolvent inverses

One possible[23] definition of the inverse of *Employee.reports_to* may be inferred by the system to become:

```
Employee.reports_to-1(Supervisor e)->Employee
SELECT _G1
WHERE  _G2=Department.mgr-1(e)and
       _G1=Employee.department-1(_G2);
```

Example 38: Body of inverted derived function.

Any derived functions referenced from within a function body are substituted by their bodies so the inverses are described in terms of only stored and foreign function inverses. If a foreign function lacks an inverse, the functions referencing the uninvertible foreign function may also lack an inverse.

The case where an uninvertible function may cause the referencing function to become uninvertible occurs when the inverse provides the only way of executing the query.

The query tree of the inverted function (ex. 38) is pictured below:

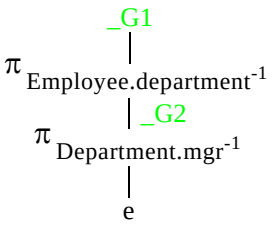


Figure 4.10: Query tree of an inverted derived function

23. There are several possible definitions of the inverses

Consider another possible inverse of *Employee.reports_to* where the body consists of non-inverted stored functions:

```
Employee.reports_to-1(Supervisor e)->Employee
SELECT _G1
WHERE  e=Department.mgr(_G2)and
       _G2=Employee.department(_G1);
```

Example 39: Body of inverted derived function using non-inverted stored functions

One possible query tree representation that corresponds to this query[24] (fig. 4.11) is considerably more complex than the query tree for the same inverted derived function using inverted functions (fig. 4.10):

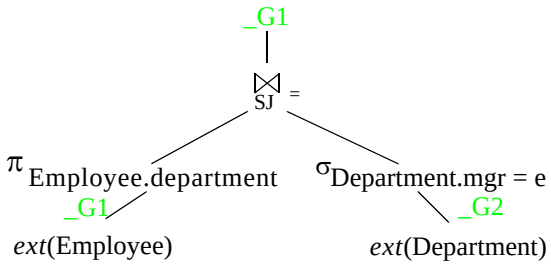


Figure 4.11: Query tree of inverted derived function with non-inverted body

In the above query tree the body consists of non-inverted stored functions. Two extents have to be scanned in order to execute the query which is a large impairment compared to the query tree where inverted stored functions were used with indexes on the result (fig. 4.10). In the latter case execution is performed in constant time[25] whereas in the former, where extents are scanned, execution time is proportional to the *cardinality*[26] of the extents.

The inverse of a derived function is generated by the optimizer which uses the inverse or regular variant of the referenced functions to find the best possible inverse of the overall derived function.

In chapter 5 optimization will be further explained along with our method to manage late bound functions. Our approach involves among several issues a strategy to execute inverted late bound functions and an approach to optimization of late bound function calls.

24. The translation to algebra is outside the scope of this thesis
25. Assuming hash index on the result
26. The number of elements in a set, denoted *card*(***S***) where ***S*** is a set.

5 Object-oriented query processing

In this chapter *cost based optimization* [47] is reviewed and our approach to optimizing late bound functions is presented.

Our approach to the management of late bound functions is to substitute the late bound call by a special algebra operator, *DTR* (*Dynamic Type Resolver*) [30], which is invertible and optimizable. This substitution is made by the query processor so the query processor must resolve the cases where late bound functions must be used.

Having a query compiler that performs this resolution means that whenever a new instance of overriding is introduced or removed, the database schema must be adapted to this change in order to remain in a consistent state. Thus, the query compiler must be able to function incrementally [29] to recompile the effected functions.

5.1 Query optimization

Good query optimization is extremely important to the overall performance of a database system since optimization can reduce the amount of resources required for executing a query dramatically. Query optimization is the task of selecting, among all equivalent execution plans, the one which requires the least resources to execute. The resources include CPU-time, time required for disk I/O, communication time over a network and memory requirement. The objective of the optimizer is to minimize the total resource requirement [47] of executing a query. The resource requirement of executing a query can be seen as the cost of execution.

By assigning a cost to the resources spent by the execution of a query and a way of determining a priori which resources a certain query will require during its execution, it is possible to judge whether a certain execution plan is cheaper than another. Thus, in order to be able to perform query optimization, the system must be able to:

- Generate all equivalent execution plans
- Determine the cost of utilizing a resource
- Determine a priori which resources the system needs to execute a query
- Calculate the overall cost of executing a query
- Compare costs to select the cheapest.

5.1.1 Complexity of optimization and heuristics

The problem of query optimization is a problem with worse than polynomial complexity [32] over the size of the query, i.e. an NP-hard problem. This is because the only way to find the optimal solution to a given problem is by exhaustive search of the search space of the possible query trees

The complexity of optimization prohibits any optimizer from finding the cheapest execution plan for an arbitrarily large query. This is because all possible execution plans must be examined in order to make this guarantee.

For relational optimizers it is often the case that the queries are small enough for the optimizer to explore all possible execution plans to find the best plan [59].

To be able to handle large queries despite the complexity, *Randomized heuristic methods*[59][32] can be used. A heuristic method is a method that does not guarantee an optimal solution. A good heuristic method is a method that is experimentally proven to often produce good plans.

As an example of such a method, there is a randomized heuristic method, *Iterative-Improvement* [32] which works as follows:

i Select random starting state in the solution space
ii Select random *neighbour*
iii If the cost of the neighbour is lower than the current cost select the neighbour as the current state and proceed with step *ii*. If not, select another neighbour and proceed with *iii*.

Steps *iii* and *ii* are repeated until a *local minimum* is reached, i.e. no neighbours are cheaper. By keeping the cheapest plan and repeating steps *i* to *iii*, until some stopping criterion is fulfilled, a query plan that is cheaper than many other plans is produced. As stopping criteria the number of iterations or elapsed time can be used.

In step *ii* of the algorithm a neighbour is selected. A neighbour is an adjacent state according to some neighbour function. The neighbour function must be easy to compute and produce a relatively smooth cost distribution for good performance of the system. The requirement of easy computation of the neighbour function exists because the computation is performed at least once in every iteration of the algorithm. The requirement of smooth cost distribution exists because increasing the number of local minimums reduces the chances of finding a global minima.

Randomized methods have been proven to work well with queries that contain many joins [59]. In such queries the neighbour is a reordering of the joins in the join tree [32]. The following figure illustrates the algorithm.

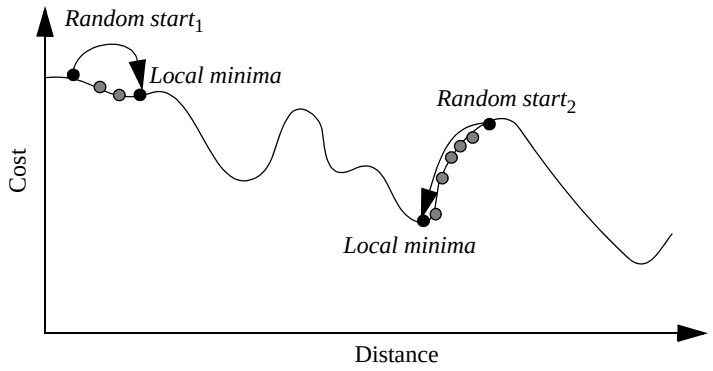


Figure 5.1: Iterative-Improvement

In figure 5.1, two examples of the algorithm are represented and marked by an arrow originating at the random start state and ending at a local minima. By performing more iterations of the algorithm, the likelihood of finding a better local minima is increased. In practice, only a limited number of iterations can be performed.

5.1.2 Query plan generation and global optimization

Generation of equivalent query plans is closely related to the algebra chosen to represent the query internally.

One way of generating all equivalent plans is to define transformation rules for the algebra and then start from a certain query tree and apply these generation rules to the tree.

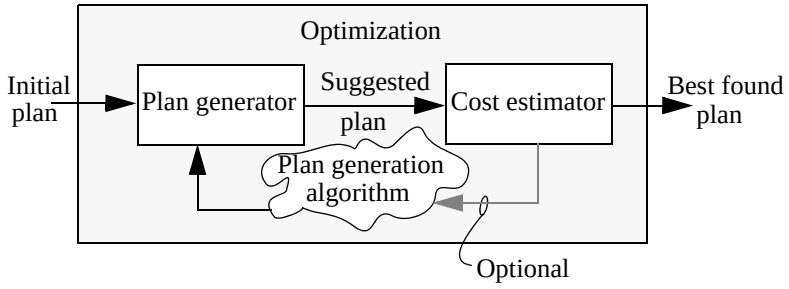


Figure 5.2: Optimization

Above (fig. 5.2), a schematic view of optimization is given. Recall from section 1 the overall view of query processing (fig. 1.2), where the input to the optimizer was an initial query tree produced by the algebra translator. By applying transformation rules and calculating the cost of execution, query optimization is performed. Plan generation is often guided by some algorithm[27],

e.g. Iterative-Improvement, to produce interesting plans.

Plan generation is a rather technical matter which is closely related to the algebra, (for a survey of relational algebra query plan generation the reader is directed to some textbook on the subject, e.g. [26]). In [54] an algebra and execution plan generation for an object-oriented model is presented.

To provide the optimizer with access to all possible query plans the implementation of all referenced functions must be accessible to the query plan generator. This means that the optimizer must be a trusted system component which is allowed to break encapsulation [22][39] to access the implementation of types and functions [47]. In relational systems this corresponds to view expansion [55] where the definitions of all referenced views are accessed. To exemplify, consider the following two AMOSQL statements in the context of the schema in figure 4.8:

```
CREATE FUNCTION reports_to(Supervisor s)->Supervisor AS
SELECT mgr(super(department(s)));

SELECT reports_to(s) FOR EACH Supervisor s
WHERE department(s)=:jd;
```

Example 40: Example definitions

In this example a derived function, *reports_to*, is defined and later used in an ad hoc query. The variable *:jd* used in the ad hoc query is assumed to have an object of the *Department* type assigned to it. When the ad hoc query is optimized the implementation of the selected *reports_to* resolvent is accessed and the overall query is optimized. In this example the resolvent *Supervisor.reports_to* is selected and the rewritten resolvent body is substituted by the function call as[28]:

27. The optional feedback loop in the figure depends on the plan generation algorithm used. If the current plan is used by the algorithm to generate the next plan, the feedback loop is present.

28. Recall from section 1.5.4 the denotation of intermediate query representation

```
reports_to(Supervisor s)->Supervisor
SELECT _G1
WHERE _G1=Department.mgr(_G2) AND
      _G2=Department.super(_G3) AND
      _G3=Employee.department(s);


SELECT _G1 FOR EACH Supervisor s
WHERE _G1=Department.mgr(_G2) AND
      _G2=Department.super(_G3) AND
      _G3=Employee.department(s) AND
      Employee.department(s)=:jd;
```

Example 41: Substitution of function names by resolvent implementations

In this example, the rewritten resolvent *Supervisor.reports_to* is first given and then substituted by the call to the function *reports_to* in the ad hoc query (ex. 40). As the example clearly shows, the query can be simplified to:

```
SELECT _G1
WHERE _G1=Department.mgr(_G2) AND
      _G2=Department.super(:jd);
```

Example 42: Simplified query

If the body of the resolvent *Supervisor.reports_to* was not accessed by the optimizer, this simplification would not be detectable. Not only are simplifications are hidden, any available indexes will also remain hidden if the implementations of referenced functions are not accessed. Hence global optimization is important to achieve good system performance.

Consider another example of the same database schema (fig. 4.8):

```
CREATE FUNCTION reports_to(Employee e)->Supervisor AS
SELECT mgr(department(e));

CREATE FUNCTION reports_to(Supervisor s)->Supervisor AS
SELECT mgr(super(department(s)));

SELECT reports_to(e) FOR EACH Employee e
WHERE department(e)=:jd;
```

Example 43: Example with overridden functions

In this example the function *reports_to* is overloaded with two resolvents: *Employee.reports_to* which is overridden by *Supervisor.reports_to*. The ad hoc query calls the function *reports_to* with an argument of the *Employee* type. A resolvent of the call to function *reports_to* cannot be selected here since late binding is required, thus the function call cannot be substituted by a resolvent body and global optimization is obstructed.

The obstruction of global optimization can cause the performance of a system to degrade significantly when the inverted variant of a late bound function is beneficial, e.g. by utilizing any available indexes. To clarify, consider an example:

```
CREATE FUNCTION supervises(Supervisor s)->Supervisor AS
SELECT s1 FOR EACH Supervisor s1
WHERE s=reports_to(s1);
```

Example 44: Definition of function supervises over managers

In the above function the optimizer will choose the inverted variant of the resolvent *Supervisor.reports_to* to avoid scanning the entire extent of the *Supervisor* type. The globally optimized function will substitute the call to the resolvent *Supervisor.reports_to* by the body of its inverse as:

```
supervises(Supervisor s)->Supervisor
SELECT s1
WHERE _G2=Department.mgr-1(s) AND
      _G3=Department.super-1(_G2) AND
      s1=Supervisor.department-1(_G3);
```

Example 45: Inlined inverted derived function

Since indexes exist on the result of all stored functions in the example (section 4.5.1), a scan of the extent of the *Supervisor* type is eliminated by using the inverse of the derived function resolvent *Supervisor.reports_to*.

If on the other hand the function *reports_to* had to be late bound, the inverse would not be accessible through global optimization as in:

```
CREATE FUNCTION supervises(Supervisor s)->Employee AS
SELECT e FOR EACH Employee e
WHERE s=reports_to(e);
```

Example 46: Definition of function supervises over employees

Here the inverses of *Employee.reports_to* and *Supervisor.reports_to* are beneficial but inaccessible since the late bound call obstructs global optimization. The consequence is that the improvement achieved by being able to use inverted functions is lost whenever late binding must be used. Hence by being able to utilize the inverses when the function is late bound, a significant performance improvement can be achieved.

5.1.3 Cost models

That the cost model reflects the actual cost of executing a query is crucial to achieve good optimization.

A cost function and a model to predict the expected number of output objects is assigned to each algebra operator. The overall cost of an execution plan is then calculated by aggregating over the execution plan.

The total cost of an algebra operation is the cost of the operation times the number of objects it has to be applied to. For example, consider the following query and its corresponding query tree:

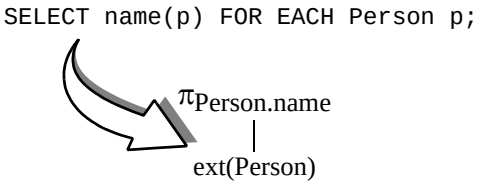


Figure 5.3: Query and query tree

In this query tree the entire extent of the *Person* type has to be scanned and the resolvent *Person.name* applied. The cost of executing the query is the cardinality of the *Person* extent times the cost of invoking the *Person.name* resolvent.

If a condition on the objects is introduced on the query above, the query and query tree will become:

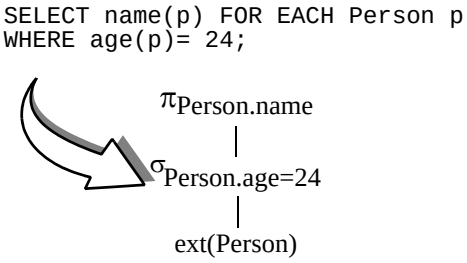


Figure 5.4: Query and query tree

In this query a condition is introduced, which reduces the number of objects to which the *name* function is applied and it will be cheaper to perform the projection. However, there is a cost of making the selection which has to be considered. The cost of executing the above tree is the sum of the selection and the

projection. It is not necessarily the case that the query tree above (fig. 5.4) is more expensive than the previous (fig. 5.3) since the cost of the selection may be very low[29] and the selectivity may be low, i.e. very few objects have the age 24 and it is very cheap to access those objects. The cost functions for each algebra operator is closely related to the underlying storage of the objects.

5.2 Managing late bound functions using the DTR operator

The problem of late bound functions is recognized as one which presents a challenge [22] and has been addressed in [21][60]. The main difficulties with having late bound functions in the execution plan are:

i How to execute inverted late bound functions, (section 4.5.1).

ii Late bound functions obstruct global optimization, (section 5.1.2).

The combination of late bound functions and inverted functions has, to the best of our knowledge, only been addressed in [30]. The problem of optimizing query plans with late bound functions has been addressed in, e.g. [21][60].

Our approach is to substitute each late bound function call in the execution plan by a special operator, DTR, which is invertible and optimizable using any cost based optimizer. During optimization the DTR is regarded as an expensive predicate [31], i.e. the cost is not negligible and standard pushing down the query tree is not applicable. Alternative approaches are evaluated in section 5.5.

5.2.1 Construction of a call to DTR

The DTR is invoked by a call as:

```
DTR(possible_resolvents,in_args)
```

Example 47: DTR call

The *possible_resolvents* is a sequence of all resolvents that may be invoked at runtime and *in_args* are the arguments of the possible resolvents.

Consider the following example from the example database schema (fig. 4.8) with the definitions of the *reports_to* resolvents (ex. 43)

```
CREATE FUNCTION supervises(Supervisor s)->Employee AS
SELECT e FOR EACH Employee e
WHERE s=reports_to(e);
```

Example 48: Definition of function supervises over employees

29. By having an index on the result of resolvent Person.age and using Person.age^{-1}.

In this function definition the call to *reports_to* must be late bound and the function call is substituted by a call to the DTR operator as:

```
CREATE FUNCTION supervises(Supervisor s)->Employee AS
SELECT e FOR EACH Employee e
WHERE  s=DTR([Supervisor.reports_to,Employee.reports_to],e);
```

Example 49: Substitution of late bound call by DTR

The possible resolvents are *Supervisor.reports_to* and *Employee.reports_to* and the argument, to the resolvent selected at runtime, is *e*. To see exactly how the DTR call is created, recall the definitions from section 4.3 and from section 4.4.

To construct a DTR call the set of all resolvents that are eligible for execution must be retrieved. This retrieval is carried out by the *resolvent** function defined as:

Definition 9: $resolvent^*$: $\boldsymbol{NM} \times \boldsymbol{Tp} \rightarrow \{\boldsymbol{FNM}\}$

If $resolvent^*(fn, \boldsymbol{S}(arg)) = \{t_j.fn, t_k.fn, \ldots, t_l.fn\}$ then, for all resolvents, $t_n.fn$, in the result set it must hold that

i $\boldsymbol{S}(t_n.fn) \in \boldsymbol{T}(\boldsymbol{S}(arg))$ or

ii $t_n.fn = resolvent(fn, \boldsymbol{S}(arg))$

The set of possible resolvents of a function call, *fn*(*arg*), is given by the above definition of $resolvent^*(fn, \boldsymbol{S}(arg))$. The possible resolvents is the set of resolvents of *fn* defined for types in the dynamic type set of the static type of argument *arg*. Furthermore, if the resolvent applicable to the static type of the argument is inherited, then it must also be added to the set since it is not defined for a type in the dynamic type set of the static type of the argument.

The set of possible resolvents is then sorted into a partial order where more specific resolvents precede less specific ones. The sorted resolvent sequence can then be used by DTR. This sorting is important in achieving good performance since the algorithms for executing DTR and DTR inverse use this sort order.

5.3 DTR

For a non-inverted, *regular*, late bound function call, *fn(arg)*, where the argument, *arg*, is bound, the resolvent is selected on the basis of the dynamic type of the argument. Thus the resolvent to apply is obtained by computing $resolvent(fn, \boldsymbol{D}(arg))$.

Recall that the first argument to DTR is a sorted sequence of resolvents with more specific resolvents early. The DTR operator implements the computation of $resolvent(fn, \boldsymbol{D}(arg))$ by selecting the first resolvent, *t.fn*, in the sorted sequence that satisfies $\boldsymbol{S}(t.fn) \geq \boldsymbol{D}(arg)$[30]. This is a sufficient criterion for applying the correct resolvent since the algorithm starts with any of the most specific

resolvents and then tries more general ones until an applicable resolvent is found.

In this way the overhead of resolvent resolution is *O(n)* in the worst case and *O(n/2)* on average, where *n* is the cardinality of the DTR resolvent sequence. The cardinality *n* is always less than or equal to the total number of resolvents of a given function name in the database schema. It is possible to perform the dispatch in constant time by using dispatch tables [2] but the benefit of dispatch tables compared to DTR is negligible if the number of possible resolvents is small. The algorithm for DTR is:

```
resolvents=DTR resolvent sequence
resolvent=first(resolvents)
while resolvent!=NULL
      if D(arg)≤S(resolvent)
          return(apply(resolvent,arg))
      end if
      resolvents=resolvents-resolvent
      resolvent=first(resolvents)
end while
```

Figure 5.5: The DTR algorithm

To make DTR optimizable using any cost based optimizer, it must be given a cost function and a model to predict the expected number of result objects. The execution cost of a late bound call must be estimated on the basis of the possible resolvents that can be applied. For example, estimating the cost of the DTR call in example 49 is performed using the execution cost and selectivity of resolvents *Supervisor.reports_to* and *Employee.reports_to*.

Avoiding bad execution plans is more important than finding the optimal plan. Therefore, we have adopted the conservative approach of using the maximum cost and the maximum *fanout*, *F*, of the possible resolvents as estimates of the cost and fanout of a DTR call. The fanout is the expected number of output objects calculated as the *selectivity* of the function times the cardinality of the argument set.

Let $c_1 \ldots c_n$ be the execution cost and $f_1 \ldots f_n$ the fanout of the possible resolvents $t_1.fn(arg) \ldots t_n.fn(arg)$, respectively. The cost, *C*, and fanout, *F*, of DTR is defined as:

30. The first resolvent which is defined to a supertype or equal to the dynamic type of the argument.

Definition 10: *Cost and fanout of DTR*

$$C = max(c_i) + k \times n \qquad F = max(f_i)$$

The constant *k* in above cost formula is the overhead of performing dynamic type dispatch among *n* resolvents. This cost will be very small compared to the cost *C*.

5.4 DTR^{-1}

An inverted late bound function call is more complex than a regular one, since the types of the result objects are used to determine which resolvent the objects should be a result of.

A call to DTR^{-1} is constructed analogously to the construction of a call to DTR but where all resolvents in the resolvent sequence are inverted. Therefore, to be able to optimize DTR^{-1}, all resolvents in the resolvent sequence must be optimized for inverse execution. If any resolvent in its resolvent sequence lacks an inverse then the DTR^{-1} also lacks an inverse and is considered uninvertible.

Recall the definition of the function *supervises* (ex. 48) where the function *reports_to* is late bound. The call can be substituted either by DTR (fig. 49) or by DTR^{-1} as:

```
CREATE FUNCTION supervises(Supervisor s)->Employee AS
SELECT e FOR EACH Employee e
WHERE  e∈DTR-1([Supervisor.reports_to-1,
                Employee.reports_to-1],s);
```

Example 50: Substitution of late bound call by DTR^{-1}

The possible resolvents and sort order are identical for DTR and for DTR^{-1}. The difference is that DTR^{-1} is described in terms of the inverses of the possible resolvents, whereas DTR is described in terms of the regular (non-inverted) resolvents.

In order to define an execution strategy for DTR^{-1} the expected result of an inverted late bound call must first be defined. An inverted late bound function call, $\boldsymbol{R}=fn^{-1}(res)$, is considered correct if each object in the result set, ***R***, when used as argument to the function *fn*, produces the value of the reference *res*.

Definition 11: *Correctness of inverted late bound function call*

Let $\boldsymbol{R}=r_1 \cup r_2 \cup \ldots \cup r_k=\{o_1\ o_2 \ldots o_n\}$, where $\boldsymbol{R}=fn^{-1}(res)$ is the result of executing the inverse of $fn(arg)=res$ and r_i is the result of executing a resolvent, $t_i.fn^{-1}$, in the resolvent sequence. For an inverted late bound function call to be considered correct then for all objects, o, there must exist a resolvent, $t_i.fn$, such that the following holds:

i $o \in r_i$ and

ii $t_i.fn=resolvent(fn, \boldsymbol{D}(o))$ and

iii $res=t_i.fn(o)$

The correctness definition states that for all objects in the result set of a possible resolvent, $t_i.fn^{-1}$, the applicable resolvent is $t_i.fn$, which, when applied to the object, returns the value. This definition is necessary in order to produce the correct result as will be illustrated by an example in the next section.

5.4.1 DTR^{-1}- an example

For this example of the result of using inverted late bound functions consider again the example database schema (fig. 4.8) and the definitions of function *reports_to* (ex. 43) and function *supervises* (ex. 48).

The database is populated as:

Type	OID	Properties
Employee	e1	department(e1)=d2
	e2	department(e2)=d2
	e3	department(e3)=d1
Supervisor	s1	department(s1)=d1
	s2	department(s2)=d2
Department	d1	super(d1)=d2, mgr(d1)=s1
	d2	super(d2)=NULL, mgr(d2)=s2

Figure 5.6: Example database population

All employees that report to a certain supervisor are sought by calling the *supervises* function with an arbitrary object of the *Supervisor* type.

```
supervises(s2);
```

Example 51: Invocation of function supervises with argument s2

The resolvent body of the *supervises* function with a call to DTR^{-1}(ex. 50) is invoked.

All objects of the *Employee* type which, when applied by the *reports_to* function is mapped to supervisor s2, are sought. Thus all objects that are instances of any subtype of the *Employee* type are also sought. This means that all resolvents in the resolvent sequence have to be executed in order to produce the desired result.

In this example there are two resolvents in the DTR^{-1} call: *Supervisor.reports_to*$^{-1}$ and *Employee.reports_to*$^{-1}$, whose intermediate forms are:

```
Employee.reports_to-1(Supervisor e)->Employee
SELECT _G1
WHERE  _G2=Department.mgr-1(e)and
       _G1=Employee.department-1(_G2);
```

```
Supervisor.reports_to-1(Supervisor s)->Supervisor
SELECT _G1
WHERE _G2=Department.mgr-1(s) AND
       _G3=Department.super-1(_G2) AND
       _G1=Employee.department-1(_G3);
```

Example 52: Inverses of Employee.reports_to and Supervisor.reports_to

The result of executing the resolvent inverse *Employee.reports_to*$^{-1}$ with argument *s2* returns the result set {*e1, e2, s2*} and the result of the resolvent inverse *Supervisor.reports_to*$^{-1}$ is the set {s1}. Notice that the correct result of DTR^{-1} is not achieved by taking the union result of the possible resolvent inverses. The union result of executing both resolvents is the set {*e1, e2, s2, s1*}. To see that this result is incorrect, recall the definition of correctness, definition 11. Using this definition for each object in the result set produces the following table:

OID(o)	Result of	*resolvent*(reports_to, *D*(o))	fn(o)
e1	Employee.reports_to^{-1}	Employee.reports_to	s2
e2	Employee.reports_to^{-1}	Employee.reports_to	s2
s2	Employee.reports_to^{-1}	Supervisor.reports_to	NULL
s1	Supervisor.reports_to^{-1}	Supervisor.reports_to	s2

Figure 5.7: Properties of objects in the result of inverted function call

Using the correctness definition on all objects of the result set we can see that all objects except the object s2 are correct result objects. Object s2 belongs to the result set of the resolvent inverse *Employee.reports_to*$^{-1}$ but the applicable resolvent is *Supervisor.reports_to*$^{-1}$ which violates the correctness criterion.

5.4.2 DTR^{-1} execution strategy

As shown in the previous example the result of DTR^{-1} is not achieved by computing the union of the possible resolvent inverses. In order to compute DTR^{-1} each result set has to be filtered to remove the objects that violate the correctness prior to calculating the union.

The result of DTR^{-1}([t_1.*fn*$^{-1}$,…, t_n.*fn*$^{-1}$], *val*)= *arg* is the union result of a special execution strategy, E_{DTR}, applied to the result of each of the possible resolvents to generate correct result sets, $\boldsymbol{r}_i$ the union of which the overall result set ***R*** can be formed.

Definition 12: *DTR*$^{-1}$

$$DTR^{-1}([t_i fn^{-1},...,t_n fn^{-1}], \mathrm{res}) = \bigcup_{i=1}^{n} E_{DTR}(t_i fn^{-1}(res), [t_i fn^{-1},...,t_n fn^{-1}])$$

We introduce E_{DTR} to remove the objects in the result of a resolvent that belong to the extent of a more specific resolvent. Without E_{DTR} the previously defined correctness criteria will be violated.

To satisfy the formula given in definition 12 the result of executing each resolvent in the resolvent sequence must not include any object *o* that is in the extent of the static type of a more specific resolvent in the resolvent sequence.

Definition 13: E_{DTR}

For each $t_i.fn^{-1}$ in the resolvent sequence, *rs*, of DTR^{-1}, the result of $E_{\mathrm{DTR}}(t_i.fn^{-1}(res), rs)$ is defined as:
Let $\boldsymbol{r}_i=\{o_1, o_2 \ldots o_n\}$, where $\boldsymbol{r}_i=E_{\mathrm{DTR}}(t_i.fn^{-1}(res), rs)$. For all objects, *o*, in the result set $\boldsymbol{r}_i$ and for all possible resolvents, $t_j.fn$ such that $\boldsymbol{S}(t_j.fn) < \boldsymbol{S}(t_i.fn)$ then
i $o \in ext(\boldsymbol{S}(t_i.fn))$ and
ii $o \notin ext(\boldsymbol{S}(t_j.fn))$

DTR^{-1} algorithm

From the above, an execution algorithm can be devised for DTR^{-1}. The algorithm uses a function *typeof* that, given an object, returns the most specific type of the object (definition , section 4.3).

As the sequence of resolvents in DTR^{-1} is sorted into a partial order with more specific resolvents early, the DTR^{-1} execution algorithm starts with any of the most specific resolvents and then proceeds with more general ones. All objects in all result sets of the resolvent inverses are checked to see if any object violates the correctness criterion (definition 11). Since the resolvents are sorted, this check can be performed by simply checking that the most specific type of the object is not a subtype of or equal to the static type of any previously executed resolvent. The algorithm is:

```
types={}
resolvents=DTR-1 resolvent sequence
result={}
For Each resolvent in resolvents   /* execute all resolvents */
      tmpres=apply(resolvent, res)
      For Each o in tmpres         /* check type of all objects in temporary result */
            validresult=TRUE
            For Each t in types
               If typeof(o) ≤ t then        validresult=FALSE
               end if
            end For Each
   If validresult then  result=result∪ o    /* add object o to res*/
   end For Each
   types=types∪S(resolvent)
end For Each
return(res)
```

Figure 5.8: DTR^{-1} algorithm

The above algorithm executes every resolvent in the DTR resolvent sequence and removes those objects that should be the result of a more specific resolvent. Such removal is performed in the if-statement, where the set *result* is extended with the object *o* if that object is an instance of a type that is not a subtype of or equal to any type in the set *types*.

To make DTR^{-1} optimizable using a cost-based optimizer, a cost model must

be defined. Since all resolvents are executed, the cost of DTR^{-1}, *C*, is the sum of the costs of the possible resolvents plus the cost of executing DTR^{-1} itself. The fanout, *F*, is estimated as the sum of the fanout of the possible inverse resolvents.

Let $c_1 \ldots c_n$ be the execution cost and $f_1 \ldots f_n$ the fanout of $t_1.fn^{-1}(res) \ldots t_n.fn^{-1}(res)$, respectively. The cost and fanout of DTR^{-1} are then estimated as:

Definition 14: *Cost and fanout of* DTR^{-1}

$$C = \sum_{i=1}^{n} k \times f_i + \sum_{i=1}^{n} c_i \qquad F = \sum_{i=1}^{n} f_i$$

In the cost formula the constant, *k*, is the overhead of checking the type of every object *o* emitted from each resolvent. The cost and fanout are used in the cost-based optimizer to decide when DTR^{-1} is favourable compared to DTR.

5.4.3 Performance analysis

To see how DTR^{-1} can improve performance, a study has been conducted where the benefit of DTR^{-1} is shown by reducing a complexity of $O(n)$ to $O(1)$ which is a dramatic improvement.

The test was performed using the example database (fig. 4.8) populated randomly. The database was scaled up in each test for supervisors / employees as 1/10 2/40 5/100 25/500 50/1000 250/5000 500/1000. Over the populated database the function *supervises* (ex. 48) was called with a randomly chosen supervisor as argument. Two variants of the *supervises* function where tested. One variant used DTR (ex. 49) and one used DTR^{-1} (ex. 50). The two variants were tested on the same database with the same manager as argument and the execution time was measured.

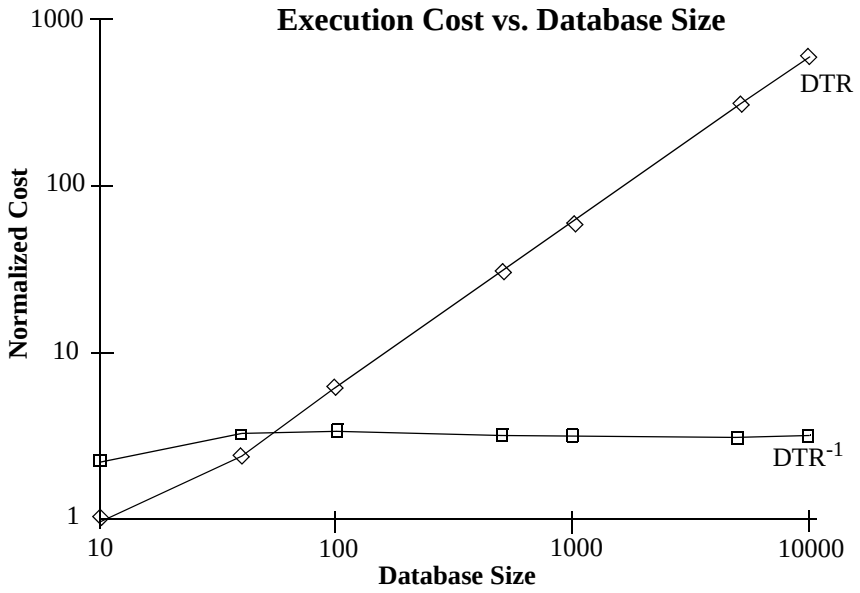


Figure 5.9: Performance study

The result (fig. 5.9) shows the normalized[31] execution time of the two variants of the *supervises* function versus the database size.

The measurements verify that the cost of executing the *supervises* function using DTR^{-1} is constant when there are more than 40 employees, as it should be, since there are constantly 20 employees per department and the cost of DTR^{-1} is proportional to the fanout of the resolvents plus the fixed execution cost. The cost of executing each resolvent is constant since hash indexes are used. By contrast, the execution of the *supervises* function using DTR is linear to the cardinality of the extent of the *Employee* type as expected. Note that selecting DTR in favour of DTR^{-1} is cheaper when the cardinality of the extent of the *Employee* type is less than 50. With a proper *k* value in the cost model of DTR^{-1}, definition 14, the optimizer will choose the correct strategy.

31. Each measured value is normalized by dividing it with the minimum value.

5.5 Other approaches to managing late bound functions

In [60] an approach to managing non-inverted late bound functions is discussed. The approach consists of two alternative strategies: a runtime dispatch or a union strategy where all possible resolvents are inlined together with type dispatch expressions. The idea is to select the union approach when the late bound functions are large and require optimization; otherwise a runtime dispatch is performed.

Since optimization is an NP-hard problem, inlining several large query trees with their type dispatch expressions enlarges the search space and it is not certain that this will be beneficial. The idea was not implemented, thus it is hard to judge whether it is better to inline all possible query trees or to perform a simple runtime dispatch.

An approach to optimization that can be applied to late bound functions is proposed in [21] where some optimization effort remains to be performed at start-up time. This approach is primarily targeting at the problem when the runtime bindings which are unknown at compile time, have significant impact on the performance of the execution. In the case for late bound functions much optimization may be required at runtime prior to the invocation of a function.

For a query that contains a late bound function one cannot choose the optimal plan at start-up time of the query since which resolvent to choose depends on the state of the database, ant not only on the runtime bindings of the parameters to the query.

5.5.1 Alternative DTR^{-1}

Recall that filtering had to be performed in the DTR^{-1} algorithm. An alternative is to inline everything into the query tree. For each possible resolvent, type dispatch expressions according to definition 12 and the resolvent body must be inlined into the query tree. The result of the inverted late bound function is the result of executing each resolvent with their type dispatch expressions and calculate the union. This approach suffers from the same problems as described above for the proposal of how to manage non-inverted late bound functions in [60].

5.6 Incremental compilation and resolution of late binding

In order to be able to use DTR in a system, the system must resolve when a function must be late bound. Such a system must also respond to schema changes to maintain the schema in a consistent state [29][63]. The concept of *schema evolution* covers all types of changes that can occur to a schema, e.g. type changes, changes to the inheritance graph and changes to the functions in the schema. The schema changes considered here are the changes that may cause inconsistencies among resolvents:

- Introduction of overriding
- Deletion of a resolvent or a database *procedure*[32].
- Redefinition of a resolvent.

Database procedures may be redefined freely since the implementation of procedures are not accessed in optimization due to the possible side-effects which may be caused during their execution. Hence procedures are treated as black-box routines by the optimizer.

Whenever any of the listed schema changes occur, the system has to respond to perform certain actions to maintain schema consistency.

5.6.1 Resolution of late binding

To make the binding policy transparent to the user, the system must resolve when a function needs to be late bound. It is important to bind late only when it cannot be avoided since late bound functions are more costly to execute. Thus, for performance reasons functions are, whenever possible, bound early. The same approach is found in O_2 [61].

Using the definition of *resolvent** in definition 9, a sufficient criterion can be defined for determining when late binding must be used.

Definition 15: *late_binding*:$\boldsymbol{NM}$ x $\boldsymbol{Tp} \rightarrow \boldsymbol{B}$

$\boldsymbol{B}$ is a set containing two values: TRUE and FALSE. A function call, *fn*(*arg*), must be late bound if the predicate *late_binding*(*fn*, $\boldsymbol{S}$(*arg*)) evaluates to TRUE where *late_binding*(*fn*, $\boldsymbol{S}$(*arg*)) evaluates to true iff *cardinality*(*resolvent**(*fn*, $\boldsymbol{S}$(*arg*)))>1, to FALSE otherwise.

5.6.2 Incremental compilation

To give the optimizer more choice, function calls appearing inside any other resolvent body are substituted by their appropriate resolvent bodies. Any resolvent using an updated resolvent must consider this change since it contains the old body of the updated resolvent. If this update is not considered, the resolvent will be inconsistent with the updated resolvent, thus leaving the schema in an inconsistent state. The approach implemented in AMOS to make the schema consistent after an update is to recompile any resolvents that must consider the change.

By having the resolvent objects maintain a set of references to the resolvents that use them, it is easy to retrieve the set of resolvents that need to respond to a certain update. These references are named *UsedByFunction*. In O_2 the dependencies are stored at the class level instead [63]. The types of updates that are relevant to system resolution of late bound functions are as follows:

32. Procedures are invoked as functions but may contain side-effects.

- Redefinition of a function resolvent, *t.fn*, must be reflected in all resolvents that use the redefined resolvent. By substituting the old definition in all function resolvents in the transitive closure of the *UsedByFunction* set of *t.fn* by the new definition of *t.fn* and re-optimizing, the database schema has adapted to the change.

- Deletion of a function resolvent, *t.fn*. All resolvents in the *UsedByFunction* relation of *t.fn* must be re-examined. If there exists another resolvent *j.fn* where $\boldsymbol{S}(j.fn) > \boldsymbol{S}(t.fn)$, then the function resolvents in the *UsedByFunction* set is re-optimized to use *j.fn*. If no such resolvent exists, the resolvents in *UsedByFunction* is deleted and any resolvent in their *UsedByFunction* relation is re-examined.
  Another case occurs s when the deleted function resolvent, *t.fn*, participates in a DTR call. If the removed resolvent is the most general one, the function resolvent with the DTR call has to be deleted since no applicable resolvent exists for the most general objects. If, on the other hand, the DTR call consists of only one possible resolvent after the deletion of *t.fn*, the DTR call is replaced by an early bound function call to the remaining resolvent.

- Introduction of a new overriding resolvent, *t.fn*, will cause all functions using the overridden resolvent to either select the new resolvent or to use DTR with the two resolvents as a possible resolvents. If the overridden resolvent is already overridden, all functions using the overridden resolvent late rebuilds their DTR calls to incorporate the new resolvent in their resolvent sequences to DTR.

All functions that need to be re-examined are identified and sorted into a partial order according to the *UsedByFunction* relation. The partial order starts with the resolvents that do not use any resolvents in the partial order. If resolvent *f* has resolvent *g* in its *UsedByFunction* relation then g must precede *f* in the order. Circular dependencies must not exist among the resolvents.

Having the resolvents sorted in this order, the correct actions will be performed by the query compiler when recompiling the functions to maintain consistency.

Having an incremental query compiler with resolution of late binding is essential to be able to use DTR and to make the binding policy transparent to the user.

6 Multi-functions in an object-oriented system

As pointed out in section 2.4, the basic object-oriented model of function invocation using message passing is not sufficient to describe certain relationships among types. In this section the basic OO model is extended with multi-functions and type resolution, and late binding of multi functions is described.

6.1 Multi-functions

As indicated in section 2.4 message passing, which is overloading on the first argument only, is not sufficient to describe bilateral relationships between objects as, for example, the distance relation (ex. 9). To overcome this shortcoming of pure object-orientation, *multi-functions* are incorporated into the model of AMOS, our research platform. Multi-functions are analogous to multi-methods [1][2][3][12][20][58] but are here called multi-functions to maintain consistency with the terminology in previous chapters.

To distinguish multi-functions from functions invoked with a message passing style, the latter are referred to as *mono-functions.*

When multi-functions are invoked, all argument types and result types participate in the resolution of which resolvent to select.

Issues that have to be addressed when multi-functions are incorporated include:

- To which type does a multi-function belong?
- When is a multi-function overloaded, overridden and when must late binding be used?
- How can multi-functions be inverted?
- How can the DTR approach be extended to apply to multi-functions?

To see the benefit of using multi-functions, recall the distance example from section 2.4 (ex. 9) which could not be expressed using overloading on the first argument. Consider the following hierarchy of types:

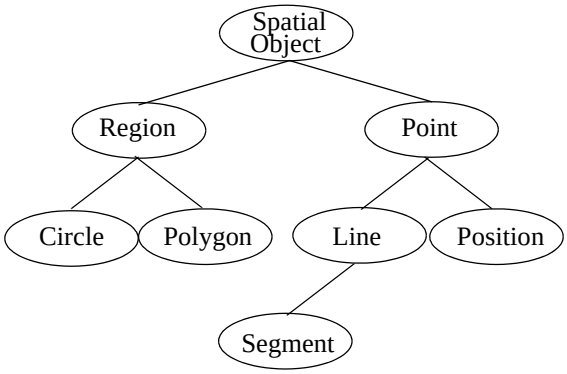


Figure 6.1: Type hierarchy of spatial types

The distance example can be expressed using AMOSQL.v1[33] as:

```
CREATE FUNCTION distance(Polygon p, Line l)->Number AS
/* Function implementation */ ;


CREATE FUNCTION distance (Polygon p, Segment s)->Number AS
/* Function implementation */ ;
```

Example 53: Distance relations in AMOSQL

The two resolvents of the *distance* functions can be used to select the objects that are not further apart than 25:

```
SELECT p, l FOR EACH Polygon p, Line l
WHERE distance(p,l)<25;
```

Example 54: Using the distance relations

In the query defined above, the *distance* function will be late bound since it is overridden on the second argument. The query will span the extent of the *Polygon* type and the extent of the *Line* type. The extent of the *Line* type has a subset that is the extent of the *Segment* type. During execution of the query the appropriate resolvent of the distance function will be selected according to the dynamic type of the reference *l*.

33. AMOSQL.v1 supports multi-functions and is the successor to AMOSQL.v0 described in section 4

Multi-functions introduce more complexity into the system since ambiguities may arise in the type resolution phase of query processing, e.g. on the type hierarchy above (fig. 6.1) two multi-functions are defined:

```
intersects(Region r, Line l)->Point;
intersects(Circle c, Point p)->Point;
```

Example 55: Multi-functions

A function application is performed as:

```
SELECT p FOR EACH Point p, Circle c, Line l
WHERE intersects(c, l)=p;
```

Example 56: Application of multi-function

It is more complex to resolve which resolvent of *intersects* that is the correct resolvent to apply in the example above when multi-functions are used since all arguments are equally important in type resolution. In above example the two resolvents are equally applicable and an ambiguity has arisen.

In [1] the problem of how to perform the disambiguation of ambiguous multi-function calls is addressed. In the approach chosen for AMOS, at this stage ambiguous calls result in an error which the programmer has to respond to, i.e. *explicit disambiguation* [1].

The remainder of this section will address the first two issues in the above list, i.e. encapsulation and type checking of multi-functions. The two remaining issues, i.e. invertibility and DTR, are future research areas that will have to be addressed to achieve an extended object-oriented database system with efficient management of late bound functions.

6.2 Types and multi-functions

In pure object-orientation the implementation of functions are encapsulated in the type to which they belong. As pointed out in OODAPLEX [24], deciding which type a multi-function should be encapsulated within is a problem. In OODAPLEX it is suggested that multi-functions should not be encapsulated within any type. This approach is chosen in AMOS as well. It is arguable that the required encapsulation exists anyway since the only way of accessing data is through stored functions or foreign functions whose implementation are hidden from the user.

A function with one argument may be encapsulated within the type of its argument. This type of function is analogous to an attribute of a type. Functions with several arguments are not encapsulated within any type. Since multi-functions do not belong to any type, they cannot be inherited from supertype to subtype. Still multi-functions are applicable to arguments that are subtypes of the

declared argument types of the function.

Rather than viewing multi-functions as a replacement for mono-functions, multi-functions can be viewed as a generalization of mono-functions and the two variants can coexist in the data model [3].

The name of a resolvent of a multi-function is an annotation of the function name with the names of all argument types and result types. For example:

```
CREATE FUNCTION intersection(Region p, Region q)->
                Region AS /* Implementation */;
```

Example 57: Creation of multi-function

creates a multi-function resolvent named[34]:

```
Region.Region.intersection->Region
```

Example 58: Multi-function resolvent

The name of a multi-function resolvent is only used by the programmer when explicit disambiguation is required, otherwise the generic name is used and the query compiler will select the appropriate resolvent for any invocation.

6.2.1 Type resolution of general multi-functions

As already emphasized, multi-functions are a generalization of mono-methods and therefore the basic data model needs only a few generalizations to incorporate multi-functions.

Multi-functions are not inherited from one type to another, but through inclusion polymorphism and substitutability multi-functions are applicable to other than the declared argument types. Consider the following function defined for the schema illustrated in figure 6.1:

```
m-fn(Region r,Point p)->Point
```

Example 59: Multi-function resolvent

The multi-function *m-fn* (ex. 59) is applicable in any context where the actual type of any argument of the function is a subtype of or equal to the declared type of the argument and the expected result type of the multi-function is a supertype to the declared result type of the multi-function.

34. The name is just a syntactic construct and the variant proposed in this thesis reflects the current implementation of AMOS.

To clarify, consider a nested function application as:

```
g(Point p)->Point
f(Integer i)->Position
g(f(5));
```

Example 60: Nested function application

In above application $g(f(5))$, the expected result type of *f* is the *Point* type, which is the declared argument type of *g*, and the declared result type of *f* is the *Position* type. Thus the expected result type of *f* is a supertype of the declared result type of *f* which makes the application legal.

Also, the number of arguments a multi-function is defined to accept must match the number of actual arguments passed to the function. This leads to the definition of the applicability of multi-functions.

Definition 16: *Applicability of multi-functions*

Let $fn(t_1, \ldots,t_n)\rightarrow t_r$ denote a multi-function with *n* arguments and a result where the type of the *i*:th argument is t_i and the result type is t_r. The multi-function $fn(t_1, \ldots,t_n)\rightarrow t_r$ is applicable to a tuple of actual arguments $<a_1, \ldots,a_m>$ and expected result a_r if

i The arity of the multi-function match the number of actual arguments, i.e. $m=n$

ii The result type t_r is a subtype of or equal to the expected result type, $\boldsymbol{S}(a_r)$.

iii For each actual argument, a_i, $\boldsymbol{S}(a_i)\leq t_i$.

If several multi-function resolvents are applicable, the *most specific applicable* (MSA) multi-function resolvent is selected as the one to apply for a given call. The definition uses subtype and static type of tuple types, these are defined after the definition of MSA.

Definition 17: MSA: $\{\boldsymbol{FNM}\}\text{x}\boldsymbol{Tp}^n\text{x}\boldsymbol{Tp} \rightarrow \boldsymbol{FNM}$

Let $ar=\{fn(t_1^1 , \ldots,t_n^1)\rightarrow t_r^1 ,\ldots ,fn(t_1^k , \ldots,t_n^k)\rightarrow t_r^k\}$ denote the set of all applicable multi-function resolvents of a particular function call $fn(a_1,\ldots,a_n)$ where the types of the actual arguments are: $at_1\ldots at_n$. and the expected result type is at_r. If $MSA(ar,<at_1\ldots at_n>,at_r) = fn(t_1^i , \ldots,t_n^i)\rightarrow t_r^i$ then for all multi-function resolvents $\boldsymbol{S}(fn(t_1^i , \ldots,t_n^i)\rightarrow t_r^i)$ in *ar* it must hold that:

i $\boldsymbol{S}(fn(t_1^j , \ldots,t_n^j)\rightarrow t_r^j) < \boldsymbol{S}(fn(t_1^i , \ldots,t_n^i)\rightarrow t_r^i)$

ii If $\boldsymbol{S}(fn(t_1^j , \ldots,t_n^j)\rightarrow t_r^j) = \boldsymbol{S}(fn(t_1^i , \ldots,t_n^i)\rightarrow t_r^i)$ then $\boldsymbol{S}(t_r^i) \leq \boldsymbol{S}(t_r^j)$

In this definition, the most specific applicable resolvent is defined to have a static type that is a subtype of the static type of all other applicable resolvents. If the static types of two resolvents are equal, the MSA is the resolvent with the

most general result type is. To conclude the definition above, the notion of the static type of a multi-function and the definition of a subtype have to be revised to address type tuples.

The static type of a multi-function is a tuple of types rather than a single type as was the case for the static type of a mono-function. The static type of a multi-function named $fn(arg_1, \ldots,arg_n)\rightarrow t_r$ is $\boldsymbol{S}(fn(arg_1, \ldots,arg_n)\rightarrow t_r)=< t_1, \ldots,t_n>$. Therefore, the previous definitions of static type, dynamic type and the dynamic type set from section 4.3 must be extended to address type tuples as follows:

- $\boldsymbol{S}(<ref_1,\ldots,ref_n>) = <\boldsymbol{S}(ref_1),\ldots,\boldsymbol{S}(ref_n)>$
- $\boldsymbol{D}(<ref_1,\ldots,ref_n>) = <\boldsymbol{D}(ref_1),\ldots,\boldsymbol{D}(ref_n)>$
- $\boldsymbol{T}(<ref_1,\ldots,ref_n>) = \{<t_1,\ldots,t_n> \mid t_1 \in \boldsymbol{T}(ref_1),\ldots,t_n \in \boldsymbol{T}(ref_n)\}$

Definition 18: *Type tuple subtype*

If $<t_1^i,\ldots,t_n^i> < <t_1^j,\ldots,t_n^j>$ then, for each pair, t_k^i , t_k^j , it must hold that:
i $t_k^i \leq t_k^j$ for all k, and
ii $t_p^i < t_p^j$ for any p

Thus, for a type tuple to be considered a subtype of another type tuple the tuples must have the same arity and each type in one tuple must be a subtype of the corresponding type of the other tuple and for at least one pair, a strict subtype relation must hold.

The definition of *MSA* above is applicable to mono-methods as well, thus definition 8, *resolvent*, can be revised to use *MSA* by retrieving all applicable resolvents of a given function name and then using *MSA* to select the most specific applicable resolvent.

Definition 19: *resolvent*: $\boldsymbol{NM}$ x$\boldsymbol{Tp}^n$x$\boldsymbol{Tp} \rightarrow \boldsymbol{FNM}$

If $resolvent(fn, <t_1,\ldots,t_n>, t_r) = fn(t_1, \ldots,t_n)\rightarrow t_r$ then,
$MSA(rs,<t_1,\ldots,t_n>, t_r) = fn(t_1, \ldots,t_n)\rightarrow t_r$ where *rs* is the set of all applicable resolvents.
If no such resolvent exists, the result is an error value.

The set of all applicable resolvents, *rs*, in the above definition can be retrieved by using the applicability condition, definition 16, on all resolvents of the particular function name, *fn*.

Overriding of multi-functions is no longer restricted to the type of the first argument of the function and thus all arguments must be considered when deciding if late binding is required. Still, it is the case that functions are bound early whenever possible.

By revising the definition of *resolvent**, definition 9 from section 5.2.1, to consider tuple types instead of single types, the existing resolution of late binding can be reused. The new definition of *resolvent** of a call $fn(a_1,\ldots,a_n)$ with expected result type, *r*, becomes:

Definition 20: $resolvent^*$: $\boldsymbol{NM}$ x$\boldsymbol{Tp}^n$x$\boldsymbol{Tp} \rightarrow \{\boldsymbol{FNM}\}$

If $resolvent^*(fn, \boldsymbol{S}(<a_1,\ldots,a_n>,r)) =$
$\{fn(t_1^1, \ldots,t_n^1)\rightarrow t_r^1,\ldots,fn(t_1^k, \ldots,t_n^k)\rightarrow t_r^k\}$ then, for all resolvents, $fn(t_1^j, \ldots,t_n^j)\rightarrow t_r^j$, in the result set, it must hold that

i $\boldsymbol{S}(fn(t_1^j, \ldots,t_n^j)\rightarrow t_r^j) \in \boldsymbol{T}(\boldsymbol{S}(<a_1,\ldots,a_n>))$ and
$\boldsymbol{S}(t_r^j) \in \boldsymbol{T}(\boldsymbol{S}(r))$ for all *i*, or

ii $fn(t_1^j, \ldots,t_n^j)\rightarrow t_r^j = resolvent(fn,\boldsymbol{S}(<a_1,\ldots,a_n>,r))$

Using the revised definition of $resolvent^*$ the criteria for when late binding is required remains the same, i.e. $resolvent^*$ is not a singleton set. Note especially that the result type can cause late binding. This is something controversial and the reason is the requirement of invertibility. There are two options with respect to overloading on the result: to allow it or to prohibit it. This will be further analysed in section 6.2.2.

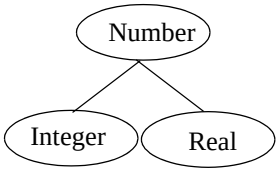
To exemplify these definitions, recall the definitions of the multi-function *intersects* (ex. 55) and the application of *intersects* (ex. 56). Both resolvents, *intersects*(*region, line*)→*point* and *intersects*(*circle, point*)→*point*, are applicable to <*point, circle*> since

- <*circle, point*>≤$\boldsymbol{S}$(*intersects*(*region, line*)→*point*)
- <*circle, point*>≤$\boldsymbol{S}$(*intersects*(*circle, point*)→*point*)

But neither of them is an *MSA* since neither of the resolvents has a static type that is a subtype of the static type of the other resolvent. Thus an ambiguity has arisen.

6.2.2 Overloading on the result

To illustrate the consequences of allowing the result type to cause late binding, consider this very simple, but perhaps unnatural example[35]:




```
CREATE FUNCTION f(Integer i)->Number AS
/*function implementation*/ ;

CREATE FUNCTION f(Integer i)->Integer AS
/*function implementation*/;

CREATE FUNCTION fn()->Number AS
SELECT n FOR EACH Integer i, Number n
WHERE n=f(i);
```

Figure 6.2: Example of late binding due to result type

At compile time, before optimization, it must be decided whether to bind function *f* late or early. During optimization it is decided whether to use *f* or f^{-1}. The following variant of function *fn* uses f^{-1}:

35. For clarity only one argument is used.

```
fn()->Number n
SELECT n
WHERE f-1(n)=i
```

Example 61: Intermediate query representation

There are two variants of f^{-1},

```
f-1(Number)->Integer
f-1(Integer)->Integer
```

Example 62: Resolvent inverses

Given these two variants it is clearly the case that late binding must be used, since the dynamic type of *n* may be Integer or Number.

The difference between multi-functions and standard message-passing invoked functions in this example is that using message passing, only the first argument is used to resolve which implementation to use and that would be impossible and lead to an exception during type resolution.

Using a non-inverted call to function *f* will result in the following variant:

```
fn()->Number n
SELECT n
WHERE f(i)=n
```

Example 63: Intermediate query representation

Both resolvents of function f must be executed in this case since both are applicable and both will produce results with correct types. The execution of the two resolvents are over subsets of the argument domain. These subsets need not be disjoint. The resolvent *f*(*Integer*)→*Number* is executed over those integers that are mapped to the instances of *Number* that not are *Integer*. The resolvent *f*(*Integer*)→*Integer* is applied to those instances that are mapped to instances of type *Integer* as:

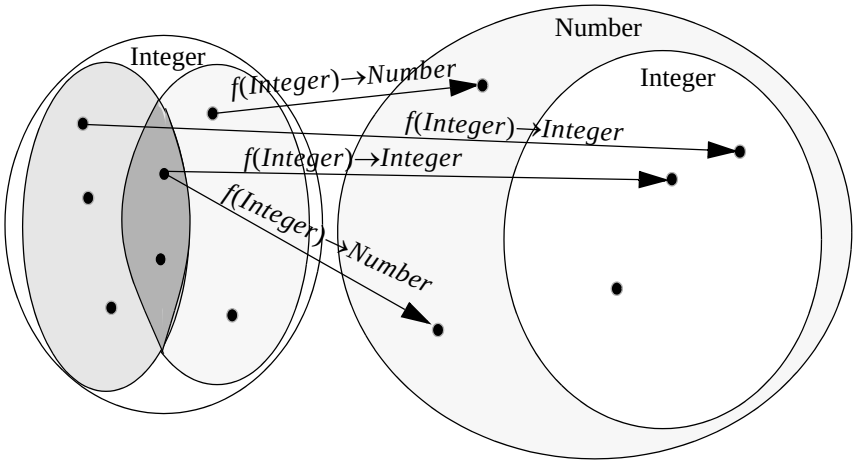


Figure 6.3: Mapping of function overridden on the result.

The figure above shows how the extent of *Integer* is divided into two non-disjoint subsets where each subset is mapped by any of the two resolvents.

Overriding on the result cause nested function applications to become more complex. Consider the following example:

```
CREATE FUNCTION g(Number n)->Integer AS STORED;
CREATE FUNCTION g(Integer i)->Integer AS i+1;
CREATE FUNCTION f(Integer i)->Number AS STORED;
CREATE FUNCTION f(Integer i)->Integer AS STORED;

SELECT g(f(i)) FOR EACH INTEGER i WHERE i=5;
```

Example 64: Nested function application with overridden result

In this example, selecting which resolvent of *f* to apply is not possible unless the expected result type of *f* is known. The expected result type of *f* is the type of the argument of function *g* where the result if *f* of is to be used. In the example above there are two resolvents of *g* that are possible.

Guided by the previous example (ex. 64) the execution strategy of the two resolvents of f is given. To the result of the two resolvents of *f*, apply the appropriate resolvent of *g* selected at runtime.

For practical reasons we suggest that the result type alone cannot cause late binding. The motivations for this suggestion are:

- Hard to find any application where it is desirable
- The data model will lose its conceptual naturalness.

6.2.3 Type resolution of restricted multi-functions

In AMOS we have decided not to allow overloading on the result. Thus in the example (fig. 6.2) it will be ambiguous as to which resolvent of the function *f* to choose.

The definition of MSA must be revised to signal an ambiguity if the types of the arguments of two different resolvents are identical and the result types of the resolvents are both subtypes of the expected type. It is possible to overload on the result type but not to use functions that are overridden on the result type. The restricted version of MSA is:

Definition 21: *Most specific applicable resolvent, MSA, restricted*

Let $ar=\{fn(t_1^1, \ldots,t_n^1)\rightarrow t_r^1, \ldots, fn(t_1^k, \ldots,t_n^k)\rightarrow t_r^k\}$ denote the set of all applicable multi-function resolvents to a particular function call $fn(a,\ldots,n)$ where the types of the actual arguments are: $at_1 \ldots at_n$ and the expected result type is at_r.
If $MSA(ar,<at_1 \ldots at_n>,at_r) = fn(t_1^j, \ldots,t_n^j)\rightarrow t_r^j$ then for all multi-function resolvents $\boldsymbol{S}(fn(t_1^i, \ldots,t_n^i)\rightarrow t_r^i)$ in *ar* it must hold that:
$i \quad \boldsymbol{S}(fn(t_1^j, \ldots,t_n^j)\rightarrow t_r^j) < \boldsymbol{S}(fn(t_1^i, \ldots,t_n^i)\rightarrow t_r^i)$

The definition of *resolvent** for restricted multi-functions becomes more complex since it must now guarantee that an MSA can always be selected when late binding is used[36]. This guarantee can be made if there do not exist any resolvents with identical argument type declarations in the set returned by *resolvent**.

Our approach differs from the approach taken in [1][20] since we allow different result types of resolvents with identical argument types. In AMOS the following overloading is allowed:

```
CREATE FUNCTION f(Integer i)->Integer AS ...;
CREATE FUNCTION f(Integer i)->Charstring AS ...;
```

Example 65: Function creation

During type checking the query processor will select the appropriate resolvent by having enough information on the expected result type. Clearly, if the necessary information regarding the expected result type cannot be derived, an ambiguity will arise.

Having multi-functions in a database system with late binding and invertibility is indeed a challenge where special execution strategies have to be invented to manage late bound multi-functions that are optimizable and invertible.

36. Which is when the set returned by *resolvent** contains several possible resolvents.

7 Summary and future work

In this last section of the thesis, a summary of the main contributions of this work and directions of future work are presented.

7.1 Summary

In this thesis the vast area of query processing has been outlined and some of the object-oriented aspects of query processing have been addressed. One particular problem in object-oriented query processing is the management of late bound function calls in the execution plan.

Query processing refers to all actions that need to be taken in order to translate and execute a high-level declarative query over some database. The main sub-areas of query processing include:

- Data model definition
- Algebra representation of queries
- Translation of declarative high-level queries to low-level algebra representation
- Optimization of algebra query trees
- Execution plan generation from algebra query tree
- Execution of plan

In this thesis the specific problem of allowing late bound functions in queries has been addressed in depth and an approach to managing late bound functions throughout all query processing steps has been presented. The main contributions of this thesis are:

- Making late bound functions invertible
- Making late bound functions optimizable
- Resolution of the requirement of late binding
- Incremental query compilation for transparent binding policy

By resolving when late binding must be used the late bound function is replaced with a call to a special algebra operator, DTR. The arguments of the DTR operator is the set of possible resolvents and their arguments.

By making DTR invertible and assigning it a cost model and fanout prediction, the DTR can be optimized using a cost-based optimizer. Thus the enhanced modelling capabilities that allowing late bound functions in the execution plan provides can be fully utilized with little or no performance degradation. The DTR and DTR inverse are defined in terms of the resolvents and resolvent inverses that are eligible for execution at runtime. The cost and

fanout prediction of DTR is made on the basis of the cost and fanout of the possible resolvents.

By performing a local optimization any available indexes within the late bound functions can be utilized and this will be reflected in the cost of the enclosing DTR.

Nevertheless, having the DTR in a database system there are remaining problems that need to be addressed.

7.2 Future work

There are a number of issues to address in order to present a complete approach that covers the area of query processing in an object-oriented database management system.

7.2.1 Multi-functions and DTR

Our proposal to make late bound functions optimizable and invertible involves functions overloaded on the first argument only. The DTR approach must be generalised to address multi-functions. A new problem that arises is that it is not meaningful to consider the inverse of a multi-function. To exemplify, consider the following figure:

```
CREATE FUNCTION m-fn(Number a,Number b)->Integer c AS
/* Implementation of function */;

SELECT b FOR EACH Number b WHERE 5=m-fn(3,b);
```

Example 66: Multi-function definition and application of multi-function

In the figure above a multi-function named *m-fn* is created. The multi-function is then applied where the result and the first argument of the multi-function are known and the second argument is sought. Obviously, when using multi-functions the inverse is just a special case of a more general concept of different configurations of bound and unbound arguments and result.

For each multi-function with n arguments and one result there are 2^{n+1} configurations of which some might be unexecutable. This means that the DTR operator must be capable of executing any of the 2^{n+1} variants.

7.2.2 Comparison operators other than equality

How can comparison operators other than equality, i.e. <, >, be used in combination with DTR and ordered indexes (various types of *search trees* [62]) in an efficient manner? Queries with such comparisons are called *range queries*.

Given a system that supports several different index structures, range queries have to be carefully optimized to utilize any search trees available. If a DTR is

present, not only the execution order (inverse or regular) has an impact the optimization of the possible resolvent, the comparison operator used must also be taken into account. To exemplify consider:

```
CREATE FUNCTION fn(Number b)-> Integer AS
/* Function Implementation */;

CREATE FUNCTION fn(Integer i)-> Integer AS
/* Function Implementation */;

SELECT i FOR EACH Integer i, Number n
WHERE i<fn(n);
```

Example 67: Late bound function in range query

In this example, the function must be late bound in the range query. For efficient execution the possible resolvents of DTR must be optimized with respect to the < operator if any efficient storage structures are utilizable. Also, inverting *fn* in the above example must consider the < operator.

Clearly, there are some issues that remain to be addressed but it is our belief that the DTR approach is viable and that the problems described here can be incorporated by generalizing and extending the basic DTR approach.

7.2.3 Algebra

The algebra used in the examples throughout this thesis is a very simple extension to the relational algebra. This algebra is only used for the purpose of exemplifying in an easily understood manner. It is important not to view it as a proposal for an algebra to be used in an object-oriented system.

It is crucial to the performance of a system that the algebra is easy to optimize and designed for an easy and efficient translation from a declarative query language into the algebra. The special features of a system must be expressible in an efficient manner in the algebra. Along with the definition of the algebra, transformation rules must be defined that supports generation of semantically equivalent query trees.

The special features of AMOS which an algebra must be able to express include: invertibility, function applications, multi-functions, late binding and the usual object oriented features, including subtyping, inheritance and complex objects.

The algebra must also be designed so that new storage structures and retrieval primitives can be incorporated into the system. This means that the algebra must be able to use new access primitives and yet be optimizable.

The reason for wanting to incorporate new storage structures is that certain applications require special storage structures and access primitives to be efficient, e.g. spatial data, image data or multimedia data.

Clearly, designing an algebra that supports all these features is a challenge.

References

1 E. Amiel, E. Dujardin, “Supporting Explicit Disambiguation of Multi-Methods”, *Research Report n2590*, Inria, 1995.

2 E. Amiel, O. Gruber, E.Simon, “Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables”, *In Proc. of the 1994 OOPSLA Conference*, 1994.

3 R. Agrawal, L. G. DeMichiel, B. G. Lindsay, “Static Type Checking of Multi-Methods”, *In Proc. of the 1991 OOPSLA Conference*, pp. 113-128, 1991.

4 M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik, “The Object-Oriented Database System Manifesto”, *Proc. of the 1st Intl. Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, Dec. 1989.

5 M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traigner, B. W. Wade, V. Watson, “System R: Relational Approach to Database Management”, *ACM Transactions on Database Systems*, Vol. 1, No. 2, June 1976.

6 F. Bancilhon, G. Ferran, “ODMG-93 The Object Database Standard”, *IEEE Data Engineering*, Vol. 17, No. 4, Dec 1994.

7 J. Banerjee, et al., “Data model issues for object oriented aplications”, *ACM Transactions on Office Inform. Syst*., Vol. 5, no. 1, Jan. 1987.

8 J. Banerjee, W. Kim, K. C. Kim, “Queries in object oriented databases”, *in Proc. IEEE Data Engineering Conference*, Feb. 1988.

9 J. Banerjee, W. Kim, H. J. Kim, H. F. Korth, “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”, *Proc. of the ACM SIGMOD Conference* 1987.

10 E. Bertino, M. Negri, G. Pelagatti, L. Sbattella, “Object-Oriented Query Languages: The Notion and the Issues”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 3, June 1992.

11 A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, “Object Structure in the Emerald System”, in *Proc. of the 1986 OOPSLA Conference*, Oregon 1986.

12 D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel, “CommonLoops Merging Lisp and Object-Oriented Programming”, *In Proc. of the 1986 OOPSLA Conference*, 1986.

13 E. F. Codd, “A relational model for large shared data banks”, *Communication of the ACM, Vol. 13, No. 6, 1970*, pp. 377-387, 1970.

14 E. F. Codd, “Further Normalization of the Data Base Relational Model”, *Data Base Systems*, R. Rustin eds., Prentice Hall, 1972.

15 L. Cardelli, P. Wegner “On Understanding Types, Data Abstraction, and Polymorphism”, *ACM Computing Surveys*, Vol. 17, No. 4, 1985.

16 M. J. Carey, D. J. DeWitt, S. L. Vanderberg, A Data Model and Query Language for EXODUS, *Proc. of the 1988 ACM SIGMOD Conference*, pp. 413-423, 1988.

17 M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O.G. Tsatalos, S. J. White, M. J. Zwilling, “Shoring Up Persistent Applications*”, *Proc. of the ACM SIGMOD Conference*, May 1994.

18 R. G. G. Catell el. al, “The Object Database Standard: ODMG - 93” *Morgan Kaufmann Publishers*. 1993.

19 R. G. G. Catell “Object Data Management: Object-Oriented and Extended Relational Database Systems”, *Addison Wesley Publishers*, 1991.

20 C. Chambers, G. T. Leavens, “Typechecking and Modules for Multi-Methods”, *In Proc. of the 1994 OOPSLA Conference*, Oct. 1994.

21 R. L. Cole, G. Graefe, “Optimization of Dynamic Query Evaluation Plans”, *Proc. of the ACM SIGMOD Conference*, May, 1994.

22 S. Daniels, G. Graefe, T. Keller, D. Maier, D. Schmidt, B. Vance, ”Query Optimization in Revelation, an Overview*” *IEEE Data Engineering bulletin*, Vol. 14, No. 2, pp. 58-62, June 1992.

23 H. Darwen, C.J. Date “The Third Manifesto”, *SIGMOD Record*, Vol 24, No1, March 1995.

24 U. Dayal, “Queries and Views in an Object-Oriented Data Model”, *Proc. 2nd Intl. Workshop on Database Programming Languages*, 1989.

25 O. Deux et. al., “The story of O_2”, *Building an Object-Oriented Database System The Story of O_2*, Morgan Kaufmann publishers, pp. 22-57, 1992.

26 R. Elmasri, S. Navathe, “Fundamentals of Database Systems” Benjamin/ Cummings, 1989.

27 G. Fahl, T. Risch, M. Sköld, “AMOS - An Architecture for Active Mediators” *Proc. Intl. Workshop on Next Generation Information Technologies and Systems*, Haifa, Israel, June 1993.

28 D. Fishman, et. al, ” Overview of the Iris DBMS”, *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley Publishers, 1989.

29 S. Flodin “An Incremental Query Compiler with Resolution of Late Binding” *Research Report LITH-IDA-R-94-46*, Department of Computer and Information Science, Linköping University, 1994.

30 S. Flodin, T. Risch, “Processing Object-Oriented Queries with Invertible Late Bound Functions”, *Proc. of the 1995 Conference on Very Large Databases*, Sept. 1995.

31 J. M. Hellerstein, M. Stonebraker, “Predicate Migration Optimizing Queries with Expensive Predicates”, *Proc. of the 1993 ACM SIGMOD Conference*, 1993.

32 Y. E. Ioannidis, Y. C. Kang, “Randomized Algorithms for Otimizing Large Join Queries”, *Proc. of the 1990 ACM SIGMOD Conference*, 1990.

33 P. Kanellakis, C. Lecluse, P. Richard, “Introduction to the Data Model”, *Building an Object-Oriented Database System The Story of O_2*, Morgan Kaufmann publishers, pp. 61-76, 1992.

34 J. S. Karlsson, S. Larsson, T. Risch, M. Sköld, M. Werner, “AMOS User’s Guide”, *CAELAB Memo 94-01*, Linköping University, 1994.

35 W. Kim, N. Ballou, H. Chou, J. F. Garza, D. Woelk, “Features of the ORION Object-Oriented Database System”, *Object-Oriented Concepts, Databases, and Applications*, pp. 251-282, ACM Press, 1989.

36 C. Lecluse, P. Richard, F. Velez, “O_2, an Object-Oriented Data Model”, *Proc. of the 1988 ACM SIGMOD Conference*, pp. 424-433, 1988.

37 W. Litwin, T. Risch, “Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates”, *IEEE Transactions on Knowldge and Data Engineering, Vol 4, No. 6*, December 1992.

38 P. Lyngbaek, “OSQL: A Language for Object Databases”, *Technical Report HPL-DTD-91-4*, Hewlett-Packard Company, 1991.

39 D. Mayer, S. Daniels, T. Keller, B. Vance, G. Graefe, W. McKenna, “Challenges for Query Processing in Object-Oriented Databases”, *Query Processing for Advanced Database Systems*, Morgan Kaufmann Publishers, 1994.

40 J. Melton, A. R. Simon, “Understanding the New SQL: A Complete Guide”, *Morgan Kaufman Publishers*, 1993.

41 B. Meyer, “Object-Oriented Software Construction”, Prentice Hall, 1988.

42 B. Meyer, “Eiffel The Language”, Prentice Hall, 1992.

43 O. Niertrasz, “A Survey of Object-Oriented Concepts”, *Object-Oriented Concepts, Databases, and Applications*, pp. 3-22, ACM Press, 1989.

44 J. E. Richardson, M. J. Carey, “Programming Constructs for Database System Implementation in EXODUS”, *Proc. of the 1987 ACM SIGMOD Conference*, pp. 208-219, 1987.

45 L. A. Rowe, M. R. Stonebraker, “The POSTGRES Data Model”, *The POSTGRES Papers, Memorandum No. UCB/ERL M86/85*, University of California, Berkley, June 1987.

46 T. Risch, M. Sköld, “Active Rules based on Object Oriented Queries”, *IEEE Data Engineering Bulletin*, Vol. 15, No. 1-4, pp. 27-30, Dec. 1992.

47 P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, “Access Path Selection in a Relational Database Management System”, 1979 *ACM SIGMOD Conference*, pp. 23-34, 1979.

48 G. M. Shaw, S. B. Zdonik, “Object-oriented queries: Equivalence and Optimization”, *Proc. 1st Conference Deductive and OO Databases, 1989*, pp. 264-278, 1989.

49 G. M. Shaw, S. B. Zdonik, “A query Algebra for Object-Oriented Databases”, *IEEE*, 1990.

50 D. W. Shipman, “The Functional Data Model and the Data Language DAPLEX”, *ACM Transactions on Database Systems, Vol. 6, No. 1*, March 1981.

51 M. Sköld, “Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques”, *Lic Thesis No 452*, Department of Computer and Information Science, Linköping University, 1994.

52 D. D. Straube, “Queries and Query Processing in Object-Oriented Database Systems”, *Ph. D Thesis*, Department of Computing Science, University of Alberta, 1991.

53 D. D. Straube, M. T. Özsu, “Queries and Query Processing in Object-Oriented Database Systems”, *ACM Transactions on Information Systems, Vol. 8, No. 4*, October 1990.

54 D.D Straube, M. T. Özsu. “Query optimization and Execution Plan Generation in Object-Oriented Data Management Systems”, To be published in *IEEE Transactions on Knowledge and Data Engineering*, April 1995.

55 M. Stonebraker, “Implementation of Integrity Constraints and Views by Query Modification”, *Proc. ACM SIGMOD Conference on Management of Data*, San Jose, Calif., May 1975.

56 M. Stonebraker, L. Rowe, “The design of POSTGRES”, *Proc. of the 1986 ACM SIGMOD Conference*, pp. 340-355, 1986.

57 B. Stroustrup, “The C++ Programming Language”, pp. 189-191, Addison Wesley Publishers, 1991.

58 V. Turau, W. Chen, “Efficient implementation of Multi-Methods through static analysis”, TR-95-053, ICSI, Berkeley, Sept. 1995.

59 A. Swami, “Optimization of Large Join Queries: Combining Heuristics and Combinatorial Techniques”, *Proc. of the 1989 ACM SIGMOD Conference*, 1989.

60 S. Vandenberg, D. DeWitt, “Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance”, *Proc. of the 1991 ACM SIGMOD Conference*, 1991.

61 F. Velez, G. Bernard, V. Dairns, “The O_2 Object Manager: An Overview”, *Building an Object-Oriented Database System The Story of O2*, Morgan Kaufmann publishers, pp. 343-368, 1992.

62 N. Wirth, “Algorithms & Data Structures”, Prentice Hall, 1986.

63 R. Zicari, “A Framework for Schema Updates in an Object-Oriented Database System”, *Building an Object-Oriented Database System The Story of O_2*, Morgan Kaufmann Publishers, pp. 146-182, 1992.