University of Alberta

The Design and Implementation of TIGUKAT User Languages

by

Anna Lipka

Technical Report TR 93–11

July 1993

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

UNIVERSITY OF ALBERTA

The Design and Implementation of TIGUKAT User Languages

BY

Anna Lipka

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1993

Abstract

To meet the data management requirements of new complex applications, object management systems are emerging as the most likely candidate. The general acceptance of this new technology depends on the increased functionality it can provide, and one measurement is the power of its query model. Users of these systems must have a declarative language to formulate queries on "what" information is required without specifying "how" to efficiently retrieve the information. Therefore, the formal query model should define a declarative calculus that can be used to formulate queries to the objectbase and an equivalent procedural algebra to execute them efficiently. In addition, a user-level language should be provided which has the same expressive power as the formal languages.

This thesis presents the new TIGUKAT Language that was designed and implemented within the framework of the TIGUKAT project. It is a high level user language which provides declarative access to the underlying objectbase. It is divided into three parts: TIGUKAT Definition Language (TDL), TIGUKAT Query Language (TQL), and TIGUKAT Control Language (TCL). The syntax of this language and the main design choices where influenced by SQL while the semantics is defined in terms of the object calculus. Queries operate on collections and they always evaluate to new collections, thus the results of queries are queryable. Furthermore, queries can be used in the predicates of other queries (i.e., nested queries). Path expressions which allow easy navigation through the schema are supported. Finally, the language is orthogonal to persistence, meaning that all objects are queryable regardless of whether they are persistent or transient.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to those who helped me throughout the course of this study. I am particularly grateful to my supervisor Dr. Tamer Özsu for his assistance, support and personal concern for myself and the study. His expectations for maintaining steady progress and producing quality work have been driving forces for me. I have benefited from his knowledge, experience and willingness to discuss many different aspects of the project.

A sincere appreciation is expressed to my co-supervisor Dr. D. Szafron for his technical assistance and support for this study. His insight into the area of the object-oriented design was the source of many fruitful discussions.

Special thanks to Randal Peters who have done a great job proof-reading earlier drafts of the thesis. His comments and suggestions helped to remove some of the rough edges of this thesis. Also, I would like to thank the whole Database Research Group for the support and valuable comments.

I am also grateful to the members of the examine committee, Dr.P. Sorensen, Dr. B. Nault for the time they took to read and comment on this thesis.

I would like to thank my friend, Mike Carbonaro for his support and encouragement during all my studies. Finally, I would like to thank Grzegorz Kondrak for his patience and my mom for her love. Thank you!

Contents

List of Figures

Chapter 1

Introduction

This thesis is a part of an ongoing TIGUKAT[1] project on the design and implementation of an object management system. The definition of the formal object model (TIGUKAT object model) that conforms to many requirements outlined in the object-oriented database manifestos [ABD+89, SRL+90] is the first result of that research [PÖS92]. The main characteristics of the TIGUKAT object model are as follows. First, it takes the uniform approach to objects which includes meta-information as primitive objects. Second, there is a clean separation and precise formal definition of many object model features. Finally, a formal specification and integration of both the behavioral and structural aspects of the object model with the necessary power for handling advanced database functionality (object creating query languages, schema evolution, updatable views, rules, etc.) is given.

The establishment of the formal object model has provided a theoretical foundation to investigate other object database features. Currently, various extensions are being added to the TIGUKAT object model in order to provide database system functionalities. These extensions include the design and implementation of the query model with the declarative facilities (formal object calculus, object algebra, and the user language), object views and view management with update semantics, dynamic schema evolution, storage management and persistence, transaction management, and the temporal aspects.

1.1 Motivation

It is well recognized that a declarative query facility is an essential component of any database management system, and the power of such a system is measured by it. Therefore, the first extension provided to TIGUKAT object model is the query model definition. This includes the specification of the formal object calculus which defines a declarative base to formulate queries, the equivalent

[1] TIGUKAT is a term in the language of the Canadian Inuit people meaning "object".

object algebra that allows them to execute efficiently, and the user-level language.

The main focus of this thesis is the design and the implementation of a high-level, user language which provides a declarative interface to the underlying object model. The language is proven equivalent to the formal languages of the query model (calculus and algebra) making it easy to perform logical transformations and argue about safety of user specified queries.

1.2 Thesis Overview

The thesis is divided into six chapters. A brief introduction to the problem of object-oriented database management systems and a short summary of the current research within these systems is given in the subsequent sections of this chapter. The discussion emphasizes the related work on object query languages. Some of the existing languages are described, and compared with the designed frameworks developed for object query languages. Chapter 2 describes the TIGUKAT object model. The main features of this model are outlined and the unique concepts are explained. Furthermore, the description of the query model which includes the formal definition of the object calculus and equivalent object algebra is given in this chapter. Chapter 3 presents the TIGUKAT language, its syntax which is given as grammar rules, and the formal semantics which is defined by the object calculus. It is shown in that chapter, that there is a complete reduction from the TIGUKAT Query Language to the object calculus. Thus, the formal semantics of the language is well defined. In Chapter 4 the integration issues between the TIGUKAT language and the model are discussed. Two extensions which facilitate this integration are described. Chapter 5 explains the implementation details. The algorithms used to implement the language translators are outlined, and the main data structures are described. Finally, in Chapter 6, conclusions and future research directions are discussed.

1.3 Related Work

The need for a technology to support the organization, control and manipulation of collections of structured data resulted in the development of database management systems (DBMSs). The 1970s witnessed the birth of hierarchical and network types of DBMSs which are still widely used today. In the 1980s, the relational model, which was proposed by Codd [Cod70, Cod71] in the beginning of the 1970s, became the base for new database systems. The relational model is characterized by the simplicity, uniformity and a strong mathematical foundation. The simplicity of the model permitted the development of powerful, non-procedural (declarative) query facilities which provided an elegant interface to the underlying model. One of the most popular query languages in those systems was SQL (Structured Query Language), which eventually became an international standard for the definition and management of relationally structured data [Com86]. It consists of three parts:

the Data Definition Language, the Data Manipulation Language and the Data Control Language.

The 1990s seem to require new directions in database development to fulfill the demands of new challenging applications. However, new technologies can compete with the previous ones only if they provide tools that are comparable in power. In order to provide at least the functionality of relational systems, next generation DBMSs must consistently extend the power of the relational query model and SQL. Therefore, one of the problems facing object-oriented system designers is the definition of an object query model and a language for these systems.

Different approaches have been taken to design new object models which support object-oriented features and conform to the object-oriented philosophy. Among them are those that have been heavily influenced by the advances in programming languages such as Smalltalk, C++, and CommonLisp (CLOS). The Gemstone [BMO+89] system is a classical example of the extension of Smalltalk to include complete database facilities. It supports multiple user access, it has the ability to accommodate large volumes of objects, and it provides persistence, query facilities, and transaction management. ORION [BCG+87, KBC+89, KGBW90] is another example of a system based on an existing programming language - in this case CLOS. Similar to Gemstone, it extends the language with database capabilities which include persistent and sharable storage, transaction management, associated queries, and database integrity control. POSTGRES [SR86, RS87], on the other hand, is a database management system based on the relational model that has been extended with object-oriented features. Thus, it supports abstract data types, inheritance, definition of rules, as well as definitions of data in form of the procedures. Finally, other models are defined independently from any programming languages or database models. Such an approach was taken by the designers of the EXODUS system [CDF+88, CDV88] as well as by the designers of the O_2 system [LRV88, Deu90, Deu91]. A similar approach is followed in TIGUKAT.

With a variety of object models comes a variety of different object query languages. This is a result of the inseparability of the data model and the query language that has to provide a declarative interface to it. Therefore, there is a tight integration between the two. Since no consensus exist for one universally accepted object model, there is presently no universally accepted object query language. In Section 1.3.1, the design principles of object query languages are discussed followed by a summary, in Section 1.3.2, of various object query languages.

1.3.1 Object Query Language Design Issues

Although there is no universally accepted object model, a core set of features has been identified and presented in number of manifestos [ABD+89, SRL+90]. Similar guidelines for the design of an object query language have appeared recently [Kim89, BTA91, BNPS92, Str91]. They are summarized below.

1. An object query language should provide a high-level, declarative interface to the underlying model. The user should not have to be aware of implementation details while specifying queries [ÖS91, Str91].

2. Similar to relational query languages, its semantics should be well defined. In other words, an object query language should be based on some formal object calculus [Str91, ÖS91].

3. It should be optimizable. The language should have an underlying object algebra defined [BCD89]. In addition, the object algebra should have the closure property meaning that results of queries should be also queryable [KKS92].

4. The language should allow queries to be arguments of predicates of other queries. Thus, the concepts of nested queries (subqueries) should be supported [Bla91].

5. The syntax of the language should be based on the SQL *select-from-where* structure. However, this can be relaxed, as the syntactic approach is subjective and depends on the designers' taste [Bla91, ASL89].

6. Path expressions, which are also called *implicit joins* as well as explicit joins should be supported by the language [Kim89, BNPS92].

7. The well known problem of impedance mismatch should not occur in object-oriented systems [Bla93, BCD89]. The object model of the object-oriented database and the type system of the programming language should be compatible.

8. Queries should be orthogonal to all data model extensions meaning that all objects should be queryable regardless of whether they are transient, persistent, distributed and so on [Bla91].

9. The language should support the syntax for the application of aggregate functions in specifying queries. These functions could be either used in target lists, as predicates, or in both [Bla91].

10. Finally, the query language whether used on an ad hoc basis, or embedded in application programs should not violate encapsulation. Data abstraction is one of the most important concepts in object-oriented systems, therefore, it should be maintained [Bla91].

1.3.2 Object Query Languages

SQL3 [Gal92], which is under development as an international standard, is expected to incorporate numerous object-oriented features. It will be a complete language for managing, creating and querying persistent objects. It will provide facilities for defining new abstract data types (ADT), creating new functions and accessing objects. However, as the language does not have any underlying object model, it contains many unnecessary and artificial constructs (objects are mapped to relational tables), while on the other hand, many important features are missing (definition of sets, classes or

other container objects). Moreover, in an effort to make the language computationally complete, non-declarative language statements are introduced (while-loop, if-statement, branch statement) which make the language unnecessarily complex. Since, SQL3 is still being designed, the standard specification is not expected to be released until 1995. A number of these problems may be resolved by then.

Blakeley [Bla91, Bla93] addresses the query-programming language integration problem in the context of an object-oriented database which uses the type system of an existing programming language C++ [ES90] as an object model. ZQL[C++] is an object query language based on the SQL paradigm. Query statements can be easily mixed with the programming language statements, and the syntax of these two languages is uniform. Therefore, the query language is well integrated with the database host language (C++) and the problem of impedance mismatch does not exists. Queries in ZQL[C++] are orthogonal to all extensions of the language. Objects can be queried regardless of whether they are transient, persistent, distributed, and so on. Query results can become inputs to other queries and can be used in the *from* and *where* clauses of other queries (i.e., nested queries). However, the formal semantics of the language is not defined, which raises question regarding the safety, completeness and optimization possibilities of the language.

A similar approach to ZQL[C++] is taken in CQL++ [DGJ92]. CQL++ is a declarative front end to Ode [AG92]. It combines an SQL-like syntax with the C++ class model. CQL++ is based on a closed object algebra which operates on sets of objects returning sets of objects as results. CQL++ is well integrated with O++ which is the host language in Ode. Moreover, queries are orthogonal to persistence, since the persistence is associated with objects.

In [BCD89, LR89b, LR89a] the main features of the query language for the O_2 [BCD89, LRV88] system are discussed. The syntax of the query language is based on the SQL *select-from-where* block, while the semantics of the language is defined as a partial mapping from sets of objects and values to a set objects and values. It is a functional language which is a subset of a host programming language. Thus the problem of impedance mismatch does not exists. The additional *flatten* operator is provided to enable the navigation through embedded sets and lists. However, the language violates the encapsulation principle when used on an ad hoc basis. Also, the semantics of the language is not based on any formal calculus.

EXCESS [CDV88], which is the query language for EXODUS [CDF+88], is different from ZQL[C++], CQL++, and O_2 languages in that it is based on QUEL syntax rather then SQL. Its main features include the uniform treatment of sets and arrays so that queries can operate on sets as well as on arrays, a type-oriented treatment of range variables, and support for update syntax. EXCESS allows path expressions to simplify the task of formulating queries. Queries in EXCESS work on sets of objects, values or tuples, and they return sets as results. Therefore, the closure property holds. Finally, EXCESS supports aggregate functions which add computational power to the language.

OSQL [Ken91] is a database language developed for the IRIS object-oriented database system.

Its design has been largely influenced by standard SQL. As a result, OSQL serves as an object description, object manipulation and query language. Furthermore, its query part has an SQL syntax. Queries are modeled as functions whose domains are either types (equivalent to the concept of classes in TIGUKAT), or bags of instances of types (collections in TIGUKAT). They always return bags as results, therefore they can become inputs to other queries. However, the syntax for nested queries in the *from* and *where* clauses is not supported.

A quite different design ideology is presented in the object query language for ObjectStore [LLOW91, OHMS92]. The C++ programming language is adopted as a host language in the system, and queries are expressed using C++ extensions supported by the C++ compiler. In other words, queries are integrated with the host language by a special query operator ([::]) whose operands are either collections or predicates. Thus, one cannot talk about the query language based on any known language like SQL or QUEL. However, the same expressive power is achieved by the queries, nested queries and path expressions in ObjectStore. Queries in this system operate on collections or predicates, and they evaluate to collections, single objects or booleans.

Finally, in [ASL89] OQL is a somewhat unorthodox object query language for an object-oriented database. The concept of a subdatabase is introduced. A subdatabase is defined as a portion of the operand database (which can be either an original database or another subdatabase that has been established by another query). It consists of an intensional association pattern (which is a network of classes) and the extensional association pattern (which is a network of instances that belong to those classes). Queries operate upon subdatabases, and they return subdatabases as results. However, since the syntax of OQL is not based on any known structure (neither SQL, nor QUEL) it is not very intuitive.

Chapter 2

TIGUKAT Overview

In this chapter an overview of TIGUKAT is given. Section 2.1 outlines the main characteristics of the TIGUKAT object model, including a description of such concepts as objects, types, classes, behaviors, functions, and the relationships among them. Section 2.2 describes the TIGUKAT query model which provides the declarative query facilities to the object model. Two formal languages are defined: an object calculus and an equivalent object algebra.

2.1 Object Model

The TIGUKAT object model [PÖS92] is defined *behaviorally* with a uniform semantics. The model is *behavioral* in the sense that the access and manipulation of objects is restricted to the application of behaviors (operations) upon objects. The model is *uniform* in that every concept within the model has the status of a *first-class object*. An object is a fundamental concept in TIGUKAT. Every component of information, including its semantics, is uniformly represented by objects in TIGUKAT. This means that at the most basic level, every expressible element in the model incorporates at least the semantics of our primitive notion for "object".

The model defines a number of primitive objects which include: *atomic entities* (such as reals, integers, strings, characters, etc.); *types* for defining and structuring features of common objects; *behaviors* for specifying the semantics of the operations which may be performed on objects; *functions* for specifying the implementation of behaviors over various types forming the support mechanism for overloading and late binding; *classes* for the automatic classification of objects based on type; and *collections* for supporting general heterogeneous user-definable groupings of objects.

The primitive type system is shown in Figure 2.1 with the `T_object` type as the root of the lattice and the `T_null` type as the base. `T_null` binds the lattice from the bottom. It is a subtype of every other type in the system. `T_null` is introduced in the model to provide an object which can be returned by behaviors that have no result.

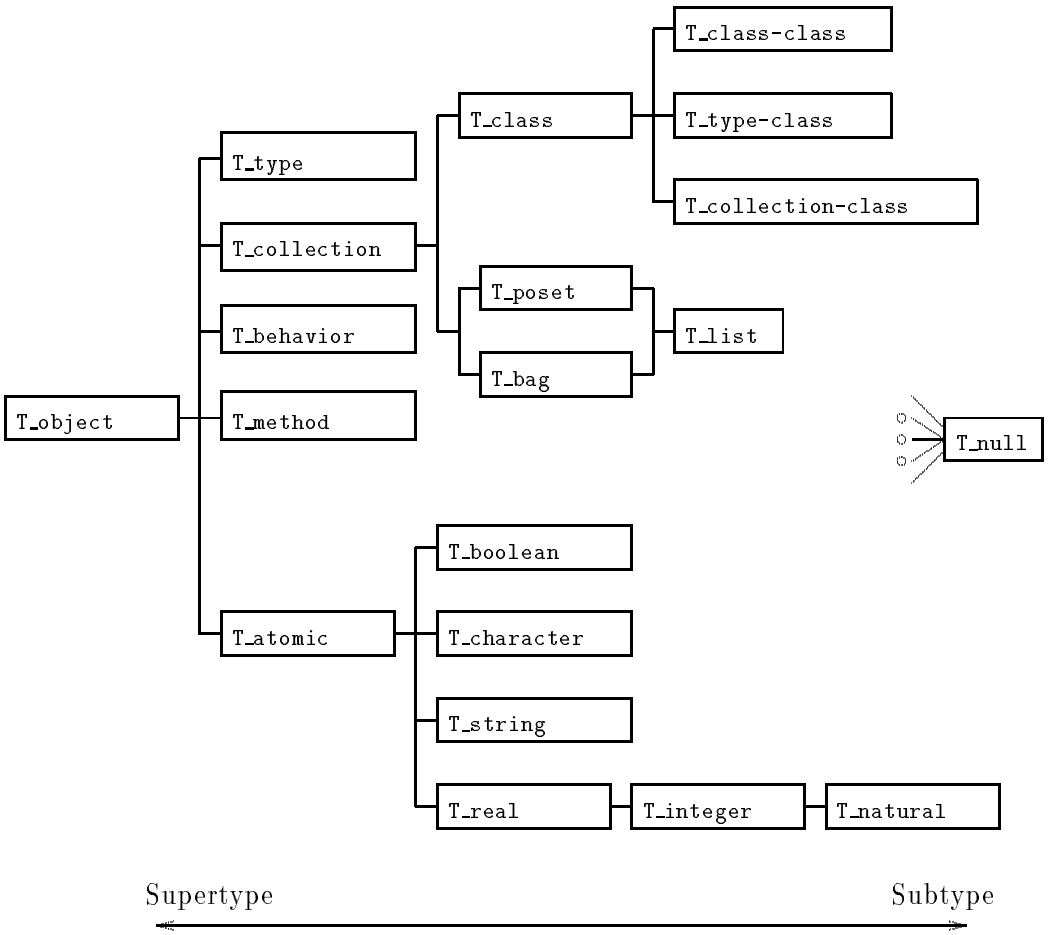

Figure 2.1: Primitive type system

Objects are defined as *(identity, state)* pairs where *identity* represents a unique, immutable system managed object identity, and *state* represents the information carried by the object. Thus, the model supports *strong object identity* [KC86], meaning that every object has a unique existence within the model and is distinguishable from every other object. On the other hand, the *state* of an object *encapsulates* the information carried by that object. Conceptually, every object is a *composite* object in TIGUKAT meaning that every object has references to other objects.

There is a separation of means for defining the characteristics of object (i.e., a *type*) from the mechanism for grouping of instances of a particular type (i.e., *class*). A *type* specifies behaviors. It encapsulates hidden implementation and state for all objects that are created by using the type as a template. The set of behaviors defined by a type is referred to as a set of *native* behaviors, and it describes the *interface* of the objects of that type. Types are organized into a lattice structure using the notion of *subtyping*. TIGUKAT supports *multiple inheritance*, meaning that one type can be an immediate subtype of several other types.

A *class* ties together the notion of *type* and *object instance*. A *class* is responsible for managing all instances that are created by using a specific type as a template. Objects of a particular type cannot exists without an associated class and every class is uniquely associated with a single type. Object creation occurs only through a class using its associated type as a template for the creation.

A *collection* is another grouping construct in TIGUKAT. It is defined as a general user-definable construct. It is similar to a *class* in that it also represents an extent of objects, but it differs in the following respects. First, no object creation can occur through a collection; object creation occurs only through classes. Second, an object may exist in any number of collections, but it is a member of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the subtype lattice, whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (*shallow extent*), along with the extensions of all its subtypes (*deep extent*). Therefore, the elements of a class are homogeneous up to inclusion polymorphism. On the other hand, a collection may be heterogeneous in the sense that it can contain objects which may be of different types.

The subtypes of `T_class` namely, `T_class-class`, `T_type-class` and `T_collection-class`, are part of the *meta* system. Their placement within the type system itself directly supports uniformity of the model. A full explanation of these types can be found in [PÖS92].

Two other fundamental notions of TIGUKAT are *behaviors* and *functions* that implement the behaviors. In the same way that an object's specification (types) is separated from the grouping of its elements (classes), the definition of a behavior is separated from its possible implementations (function/methods).

The semantics of each operation on an object is specified by a behavior defined on its type. A function implements the semantics of each behavior. The implementation of a particular behavior

may vary over the types which support it. Nevertheless, the semantics of the behavior remains constant and unique over all types supporting that behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code. A *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavior application as the invocation of a function, regardless of whether the function is stored or computed.

The following example illustrates a geographic information system in the TIGUKAT object model. This example, taken from [PÖS92], will be used as a running example throughout this thesis.

Example 2.1 Object-orientation is intended to serve many application areas requiring advanced data representation and manipulation. A geographic information system (GIS) [Aro89, Tom90] has been selected as an example to illustrate the practicality of the concepts introduced and to assist in clarifying their semantics. A GIS was chosen because it is among the application domains which can potentially benefit from the advanced features offered by object-oriented technology. Specifically, a GIS requires the following capabilities:

1. management of persistent and transient data,
2. management of large quantities of diverse data types and dynamic evolution of types,
3. a seamless integration of sophisticated computer graphic images with complex structured attribute data,
4. handling of large volumes of data and performing extensive numerical tabulations on data,
5. management of differing views of data, and
6. the ability to efficiently answer a variety of ad hoc queries.

A type lattice for a simplified GIS is given in Figure 2.2. The example is sufficiently complex to illustrate the advanced functionality of the query model we present, yet simple enough to be understandable without an elaborate discussion. The example includes the root types of the various sub-lattices from the primitive type system in Figure 2.1 to illustrate their relative position in an extended application lattice. The additional types defined by the GIS example include:

1. Abstract types for representing information on people and their dwellings. These include the types `T_person`, `T_dwelling` and `T_house`.
2. Geographic types to store information about the locations of dwellings and their surrounding areas. These include the type `T_location`, the type `T_zone` along with its subtypes which categorize the various zones of a geographic area, and the type `T_map` which defines a collection of zones suitable for displaying in a window.

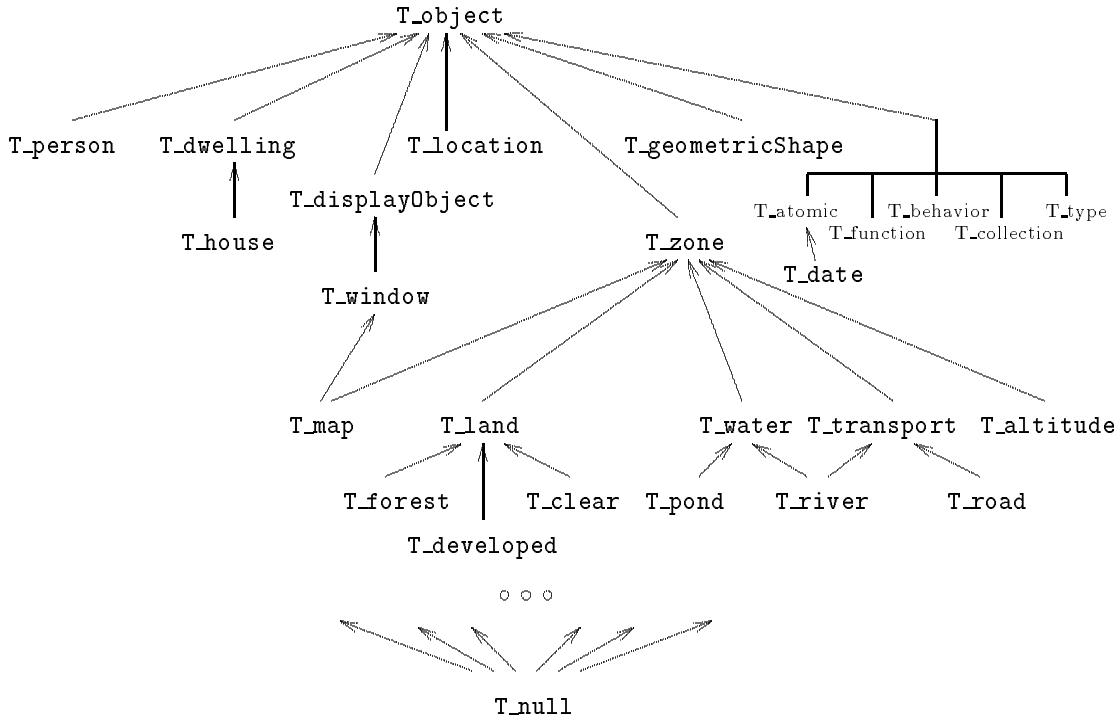

Figure 2.2: Geographic Information System in TIGUKAT object model.

3. Displayable types for presenting information on a graphical device. These include the types `T_displayObject` and `T_window` which are application independent and the type `T_map` which is the only GIS application specific object that can be displayed.

4. A type `T_geometricShape` that defines the geometric shape of the regions representing the various zones. For our purposes we will only use this general type, but in more practical applications this type would be further specialized into subtypes representing polygons, polygons with holes, rectangles, squares, splines and so on.

2.2 Query Model

A complete uniform behavioral object model has formed basis for an object query model that includes a complete algebra with an equivalent object calculus definition. An underlying characteristic of the TIGUKAT query model is that it is a direct extension to the object model. In other words, it is defined by type and behavior extensions to the primitive model.

The subsequent sections summarize the formal languages defined for the TIGUKAT query model. The full specification of the query model is given in [PLÖS93]. The first section presents the object calculus with the first-order semantics. The logical foundation of the calculus includes a definition of atoms, well-formed formulas, and a function symbol which incorporates the behavioral nature of

Type	Signatures	
T_location	*B_latitude*:	T_real
	B_longitude:	T_real
T_displayObject	*B_display*:	T_displayObject
T_window	*B_resize*:	T_window
	B_drag:	T_window
T_geometricShape		
T_zone	*B_title*:	T_string
	B_origin:	T_location
	B_region:	T_geometricShape
	B_area:	T_real
	B_proximity:	T_zone → T_real
T_map	*B_resolution*:	T_real
	B_orientation:	T_real
	B_zones:	T_collection⟨T_zone⟩
T_land	*B_value*:	T_real
T_water	*B_volume*:	T_real
T_transport	*B_efficiency*:	T_real
T_altitude	*B_low*:	T_integer
	B_high:	T_integer
T_person	*B_name*:	T_string
	B_birthDate:	T_date
	B_age:	T_natural
	B_residence:	T_dwelling
	B_spouse:	T_person
	B_children:	T_person → T_collection⟨T_person⟩
T_dwelling	*B_address*:	T_string
	B_inZone:	T_land
T_house	*B_inZone*:	T_developed[a]
	B_mortgage:	T_real

[a] Behavior was refined from supertype T_dwelling.

Table 2.1: Behavior signatures pertaining to example specific types of Figure 2.2

the object model. This allows the use of very general path expressions in the calculus. The safety of the calculus is based on the *evaluable* class of queries defined in [GT91]. The second section presents the object algebra which is proven equivalent to the object calculus. Target-preserving and target-creating algebraic operators are shown.

2.2.1 The Object Calculus

The alphabet of the calculus consists of object constants (a, b, c, d), object variables (o, p, q, u, v, x, y, z), monadic predicates (C, P, Q), dyadic predicates $(=, \neg, \in, \notin)$, an $n-$ary predicate $(Eval)$, a function symbol (β), logical connectives $(\exists, \forall, \wedge, \vee, \neg)$, and delimiters (&, (),).

Atoms are the building blocks of calculus expressions. The *atoms* of the calculus consist of the following:

Range Atom: $C(s)$ is called a range atom for s where C corresponds to a unary predicate representing a collection and s denotes a term. A range atom asserts true if and only if s denotes an object in collection C. When C defines a class, $C^+(s)$ is true if and only if s denotes an object in the *shallow extent* of class C.

Equality Atom: $s = t$ is a built-in predicate called an *equality atom* where s and t are terms. The predicate asserts true if and only if the object denoted by s is object identity equal to the object denoted by t.

Membership Atom: $s \in t$ is a built-in predicate called a *membership atom* where s and t are terms, and t denotes a collection. The predicate asserts true, if and only if the object denoted by s is an element of the collection denoted by t.

Generating Atom: Any equality atom of the form $o = t$ or membership atom $o \in t$, where o is an object variable and t is an appropriate term for the atom in which o does not appear, is called a *generating atom* for o. That means that the object denotation for o can be generated from t.

The *ground atom* is an atom that contains only ground terms.

From atoms, *well-formed formulas* (WFFs) are built to construct the declarative calculus expressions of the language. WFFs are defined recursively from atoms in the usual way [Cod71, Ull82] using the connectives $\wedge, \vee, \neg$ and the quantifiers $\exists$ and $\forall$.

A *target-preserving* query is an object calculus expression of the form $\{t|\phi(o)\}$ where t is a term consisting of a single object variable or an object variable indexed by a list of behaviors, ϕ is a WFF, and o is exactly the variable in t and it is the only free variable referenced in ϕ. Indexed object variables are of the form $o[\beta]$ where β is a set of behaviors defined on the type of variable o. The semantics of this construct is to project over the behaviors in β for o, meaning that after the operation only the behaviors given in β will be applicable to o. A *target creating* query is of the

form $\{t_1, ..., t_n | \phi(o_1, ..., o_n\}$ which is simply a generalization of the target preserving kind by allowing multiple target terms $t_1, ..., t_n$ over the multiple variables $o_1, ..., o_n$. The result of such a query is a collection of new object lists created from the cartesian product over ranges of variables $o_1, ..., o_n$ by following the selection using $\phi(o_1, ..., o_n)$.

Example 2.2 Target-preserving query: *Return all zones that are part of the same map. Project the result over B_title and B_area.*

$$\{o[B\_title, B\_area] | \exists p(\mathbf{C\_map}(p) \wedge o \in p.B\_zones)\}$$

o is a free variable generated by the generating atom: $o \in p.B\_zones$, and $t = o[B\_title, B\_area]$ is a target variable in form of the index variable.

Target-creating query: *Return all the people and their spouses such that both of them are older then 65 years old*

$$\{p, q | \mathbf{C\_person}(p) \wedge q = p.B\_spouse \wedge p.B\_age > 65 \wedge q.B\_age > 65\}$$

Since, there are two target variables in the target list, this is an example of a target-creating query.

2.2.2 The Object Algebra

The operands and results of the object algebraic operators are typed collections of objects. The algebra maintains the closure property since the results of any operator may be used as an operand of another. The object algebra defines both *target-preserving* and *target-creating* operators. The target preserving operators are defined as follows:

Set Operations The typical set **union, difference** and **intersection** operators are defined.

Select (denoted $P\sigma_{[F]} < Q_1, ..., Q_n >$): Select is a higher order predicate that accepts the predicate F, and the n+1-ary collection $P, Q_1, ..., Q_n$ as arguments. The result collection contains objects from P corresponding to the p components of each permutation $< p, q_1, ..., q_n >$ that satisfies F.

Map (denoted $Q_1 \gg_{mop} < Q_1, ...Q_n >$): where *mop* is a mop function [PLÖS93] over the elements of collections $Q_1, Q_2, ..., Q_n$, meaning it expects arguments $q_1, q_2, ..., q_n$ and they are type consistent with the membership types of the collections. For each permutation of objects $< q_1, q_2, ..., q_n >$ form from the elements of the argument collections $mop(q_1, q_2, ..., q_n)$ is applied and the resulting object is included in the result collection.

Project (denoted $P\Pi_\beta$): where P is a collection and β is a behavioral projection set with the restriction that it is a subset of the behaviors defined on the membership type of P. The β collection is automatically unioned with the behaviors of type `T_object` in order to ensure consistency. The result collection contains objects of P, but with the membership type coinciding with the behavior specification of β.

The full object algebra includes target-creating operators in order to provide necessary object formation operators. The result of these operations is a collection of new objects that are object identity distinguishable from the ones in the argument collection. The primary target-creating operator is *product*:

Product (denoted $P \times Q$): Product produces a collection containing product objects created from each permutation $< p, q >$ such that the left component is an object from P and the right component is an object from Q. Product may initiate the creation of a new type along with a new class to maintain the product objects.

The above collection of operators form the *primitive* algebra. They are fundamental in supporting the expressive power of the calculus and other expressions can be defined in terms of them. The following operators are added to the primitive algebra in order to provide functionality, and increase the expressive power.

Join (denoted $P \bowtie_{[F]} < Q_1, ..., Q_n >$): where $n \geq 1$. Join produces a collection containing product objects created from each permutation $< p, q_1, ..., q_n >$ that satisfies F.

Generate Join (denoted $Q_1 \gamma^{o}_{[g]} < Q_2, ..., Q_n >$): where g is a generating atom of the form $o\theta < \vec{q} > .\vec{b}$ (where θ is one of '=' or '$\in$') over the elements of collections $Q_1, Q_2, ..., Q_n$. Generate join produces a collection of product objects created from each permutation of the q_i's and extended by an object o in the following way. If θ is '=', the result contains product objects of the form $< q_1, ..., q_n, < q_1, ..., q_n > .\vec{b} >$ for each permutation of the q_i's. If θ is $\in$, the result contains product objects of the form $< q_1, ..., q_n, o >$ for each permutation of the q_i's and $o \in < q_1, ..., q_n > .\vec{b}$.

Reduce (denoted $P_{\vec{p}} \triangle_{\vec{o}}$): where P is a collection of product objects $\vec{p}$, and $\vec{o}$ is a list representing symbolic reference to the component of the product. The reduce operator has the effect of discarding the $\vec{o}$ components of the objects in P. That is, product objects of the form $< p_1, ..., p_i, \vec{o}, p_{i+1}, ..., p_n >$ are mapped to $< p_1, ..., p_i, p_{i+1}, ..., p_n >$.

Collapse (denoted $P \Downarrow$): Collapse is a unary operator which accepts a collection of collections P as an argument and it produces the extended union of the collections in P.

The following examples illustrate possible queries on the GIS defined in Example 2.1. Every query is given in form of an English sentence, then it is expressed in the object calculus which is followed by the equivalent algebraic expression. In the algebraic expressions, operand collections are subscripted by the variable that ranges over them. If the operand consists of product objects, the variables that make up the components of these objects are listed. The indexed variables are used as a symbolic reference to the elements of the collection as described in this section. Furthermore, the arithmetic notation for operations like *o.greaterthan(p)* and *o.elementof(p)* is used instead of

boolean Bspec equivalents. The execution of an algebraic expression is from left-to-right, except that parenthesized expressions are executed first.

Example 2.3 Return land zones valued over $100,000 or covering an area over 1000 units.

Calculus:

$$\{\ o \mid \mathbf{C\_land}(o) \wedge (o.B\_value > 100000 \vee o.B\_area > 1000)\}$$

Algebra:

$$\mathbf{C\_land}_o\ \sigma_{[o.B\_value > 100000\ \vee\ o.B\_area > 1000]}$$

Example 2.4 Return all zones that have people living in them (the zones are generated from person objects).

Calculus:

$$\{\ o \mid \exists q(\mathbf{C\_person}(q) \wedge o = q.B\_residence.B\_inzone)\}$$

Algebra:

$$\left(\mathbf{C\_person}_q\ \gamma^o_{o=q.B\_residence.B\_inzone}\right)_{o,q}\ \triangle_q$$

Example 2.5 Return the maps with areas where citizens over 65 years of age live.

Calculus:

$$\{\ o \mid \mathbf{C\_map}(o)\ \wedge\ \exists p(\mathbf{C\_person}(p)\ \wedge\ \exists q(\mathbf{C\_dwelling}(q)$$
$$\wedge\ p.B\_age \geq 65\ \wedge\ q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones))\}$$

Algebra:

$$\left(\mathbf{C\_map}_o\ \bowtie_{F_1}\ \langle \mathbf{C\_dwelling}_q,\ (\mathbf{C\_person}_p\ \sigma_{F_2}\ )_p\rangle\right)_{o,q,p}\ \triangle_{p,q}$$

where F_1 is the predicate $(q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones)$
and F_2 is the predicate $(p.B\_age \geq 65)$

Example 2.6 Return all maps that describe areas strictly above 5000 feet.

Calculus:

$$\{\ o \mid \mathbf{C\_map}(o) \wedge \forall p(\neg \mathbf{C\_altitude}(p) \vee \neg(p \in o.B\_zones)$$
$$\vee p.B\_low > 5000)\}.$$

Algebra:

$$\mathbf{C\_map} - \left(\left(\mathbf{C\_map}_o\ \bowtie_{F_1}\ (\mathbf{C\_altitude}_p\ \sigma_{F_2}\ )_p\right)_{o,p}\ \triangle_p\right)$$

where F_1 is a generating atom $(p \in o.B\_zones)$
and F_2 is a predicate $(\neg(p.B\_low > 5000))$

Example 2.7 Return the dollar values of the zones that people live in.

Calculus:

$$\{\ o \mid \exists p(\mathbf{C\_person}(p) \wedge o = p.B\_residence.B\_inzone.B\_value)\}.$$

Algebra:

$$\Big(\mathbf{C\_person}_p \ \gamma^{o}_{o=p.B\_residence.B\_inzone.B\_value}\ \Big)_{p,o} \ \triangle_p$$

Note that this has a simpler form using the map operator as follows:

$$\mathbf{C\_person}_p \gg_{p.B\_residence.B\_inzone.B\_value}$$

Example 2.8 Return the zones that are part of some map and are within 10 units from water. Project the result over *B_title* and *B_area*.

Calculus:

$$\{\ o[B\_title, B\_area] \mid \exists p \exists q(\mathbf{C\_map}(p) \wedge \mathbf{C\_water}(q)$$
$$\wedge o \in p.B\_zones \wedge o.B\_proximity(q) < 10)\}.$$

Algebra:

$$\Big(\big(\mathbf{C\_map}_p \ \gamma^{o}_{F_1}\ \big)_{p,o} \ \bowtie_{F_2} \ \mathbf{C\_water}_q\Big)_{p,o,q} \ \triangle_{q,p} \ \Pi_{B\_title,B\_name}$$

where F_1 is a generating atom ($o \in p.B\_zones$)

and F_2 is a predicate ($o.B\_proximity(q) < 10$)

Example 2.9 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

Calculus:

$$\{p, o \mid \exists q(\mathbf{C\_person}(p) \wedge \mathbf{C\_map}(q)$$
$$\wedge\ o = q.B\_title \wedge p.B\_residence.B\_inZone \in q.B\_zones)\}$$

Algebra:

$$\Big(\mathbf{C\_person}_p \ \bowtie_F \ \Big(\mathbf{C\_map}_q \ \gamma^{o}_{o=q.B\_title}\ \Big)_{q,o}\Big)_{p,q,o} \ \triangle_q$$

where F is a predicate ($p.B\_residence.B\_inZone \in q.B\_zones$)

Chapter 3

TIGUKAT Languages

The main function of the TIGUKAT language is to support the definition, the manipulation and the retrieval of objects in a TIGUKAT objectbase on an ad hoc basis. It is not a computationally complete language in that flow control statements for iteration and conditional execution are not supported. A complete objectbase programming language will be developed in the future, and it will subsume this work. The TIGUKAT language supports the features defined in the TIGUKAT object model. Thus, new types, classes, collections, behaviors and functions can be created using the language statements. Functions can be written in the TIGUKAT language as well as in other programming languages such as C++. The TIGUKAT language also supports the concept of composite objects, enabling querying, retrieving, and accessing them.

The TIGUKAT language consists of three separate parts: TIGUKAT Definition Language (TDL), TIGUKAT Query Language (TQL), and TIGUKAT Control Language (TCL). TDL supports the definition of metaobjects in a TIGUKAT objectbase. Types, collections, classes, behaviors and functions are created using TDL statements. TQL supports the retrieval of objects in a TIGUKAT objectbase. Its syntax are based on the SQL paradigm, while the semantics of the language is defined by the object calculus. Finally, TCL supports session specific operations like opening a session, saving a session, and making objects persistent. The description of each of these languages is given in the subsequent sections, while the full syntax of the TIGUKAT language is described in Appendix A.

3.1 Notation

The notation used throughout this chapter is as follows. All bold words and characters correspond to terminal symbols of the language (keywords, special characters, etc.). Nonterminal symbols are enclosed between '<' and '>'. Vertical bars '|' separate alternatives. The square brackets '[', ']' enclose optional material which consists of one or more items separated by vertical bars. Finally, all

the rules of the form < '*element*' *list* > are comma separated lists of '*elements*'.

3.2 TIGUKAT Definition Language

TDL supports the definition and the creation of metaobjects. All type, collection, class, behavior, and function objects in the objectbase are considered metaobjects. TDL is logically divided into six groups of statements: type declarations, collection declarations, class declarations, behavior manipulations, function declarations, and associations. Statements in the TIGUKAT language are separated by a semicolon.

A type declaration statement is used to create new type objects in a TIGUKAT objectbase. The general syntax of this statement is:

< *type declaration* > ::= **create type** < *new reference* >
under < *type list* >
< *behavior specification* >

The *create type clause* declares a reference to a new type. The *under clause* contains a type list which defines all direct supertypes of a new type. This list cannot be empty, as every type in TIGUKAT is at least a subtype of `T_Object` type. The last part of a type declaration statement is the behavior specification which is made up of public and private behaviors:

< *behavior specification* > ::= [< *public behaviors* >] [< *private behaviors* >]

where public and private behaviors are defined as follows:

< *public behaviors* > ::= **public** < *signature list* >

< *private behaviors* > ::= **private** < *signature list* >

Every behavior is declared either a **public** behavior or a **private** behavior. A public behavior is visible to all authorized users of the type, while a private behavior is totally encapsulated, and it is visible only within the definition of its type. All names of behaviors must be unique within a given type, and all its supertypes. Thus, a definition of a behavior which is already defined in one of the supertypes of a defined type cannot be repeated in that type. In order to redefine a behavior inherited from a supertype, a new association must be done between the behavior and some new function.

< *signature* > ::= < *behavior name* > [(< *type list* >)]: < *type reference* >

Each signature in the signature list consists of a behavior name which also becomes a behavior reference, the optional list of type references which define types of behavior parameters, and a single type reference, specified after the colon, which defines the type of the behavior result. The following example illustrates the creation of a new type `T_person` in the TIGUKAT objectbase:

create type `T_person`
under `T_Object`
public: *B_getName*: `T_string`,
B_setName(`T_string`): `T_string`,
B_getBrtday: `T_date`,
B_setBrtday(`T_date`): `T_date`

The new type `T_person` is defined as a direct subtype of `T_Object` which is a primitive type in TIGUKAT. The public interface of `T_person` type consists of four behaviors: *B_getName*, *B_setName*, *B_getBrtday*, *B_setBrtday*. It does not have any private behaviors. Type `T_string` is a primitive type in TIGUKAT and we assume that `T_date` has already been defined, so we can use it. It should be noted here that all behaviors specified in the type declaration statement are automatically created and associated with a defined type. Thus, the type declaration statement can also become an implicit behavior declaration statement.

Behavior manipulation statements are used to manipulate behaviors within existing types. New behaviors can be added to existing types, or native behaviors can be removed from them. The general syntax of these statements is as follows:

< *behavior manipulation* > ::=
add to < *type reference* > < *behavior specification* >
| **remove from** < *type reference* >
behaviors: < *name list* >

The first statement adds new behaviors to an existing type. The first component of this statement is a type reference which declares the type with which new behaviors are to be associated. The behavior specification declares the behaviors which must be added to a given type. The *remove* statement deletes behaviors from a given type. The type reference is a reference to an existing type from which the behaviors are to be removed. The *name list* in the behavior clause specifies behaviors which should be removed. However, only the native behaviors can be removed from a given type. They are automatically removed from all the subtypes of this type. In the following example, two new public behaviors are added to `T_person` type:

add to `T_person`
public: *B_age*: `T_natural`,
B_spouse(`T_person`):`T_person`;

Every behavior in the TIGUKAT objectbase must be associated with a function object which provides an implementation of the behavior semantics. The semantics of the behavior and the semantics of the corresponding functions must be the same. There are two kinds of functions in the TIGUKAT objectbase: stored functions and computed functions. Stored functions do not have any parameters,

and their result type can be inferred from the result type of the corresponding behavior, therefore they do not have to be declared explicitly. They are created when the association statement is invoked (see association statement in this section). Computed functions, on the other hand, must be explicitly declared using one of the following declaration statements:

< *function declaration* > ::= < *language* >**function**< *function signature* >
begin
< *function code* >
end
| **external function** < *function signature* >

Thus, there are two ways to declare computed functions. A user can either write a complete function, specifying the language used and providing the code of the function, or the user can declare a reference to an external function which has already been defined, and exists in the objectbase. The *language clause* in the first statement specifies the programming language which is to be used to write the function code. So far, there are two languages which can be used to write function code in TIGUKAT: TQL and C++. However, other languages will be supported in the future. The second statement for computed functions is used to declare references to function objects which already exist in the objectbase. The function signature specifies the semantics of the function, and the function object with the same signature (semantics) is bound to the local reference. Thus, in both function declaration statements, a function signature must be declared. A function signature has the format:

< *function signature* >
::= < *function name* > [(< *formal parameter list* >)]:< *type reference* >

A *function name* in the function signature specifies the unique name of the function and it becomes a reference to the function object. The formal parameter list is made up of formal parameters:

< *formal parameter* > ::= < *identifier* > : < *type reference* >

If the first parameter is referenced by the keyword **self**, then it declares the type of the receiver object. In other words, it defines a type with which the function can be associated. If it is not specified, the `T_Object` primitive type is assumed by default. All other parameters in this list define the parameters and their types. The last part of a function signature is a type reference specified after a colon. It defines the result type of the function. In the following example, two computed functions *f_age* and *f_spouse* are declared:

C++ function *f_age*(self:`T_person`): `T_integer`
begin
`T_date` today();

```
        today.initDate(); /* initDate() is a behavior defined on T_Date */
        return (today - B_getBrtday());
    end;

    external function f_spouse(self:T_person, p:T_person): T_person;
```

The first statement declares a new computed function which is written in C++ language. This function can be associated with a behavior in the `T_person` type, or a behavior in any of its subtypes. A new function object is created, and the reference *f_age* is bound to it. The second statement declares a local reference *f_spouse* and binds it to the external function object with the same semantics. If there are more then one object with the same semantics in the objectbase, then the system prompts the user about the ambiguity and it must be resolved.

To associate a behavior with a corresponding function, the *association statement* is used. The general syntax of the association statements is:

$$< association > ::= \textbf{associate in } < type\ reference > \\ [< computed\ list >] \\ [< stored\ list >]$$

where the *computed list* is a comma-separated list of computed function associations, and the *stored list* is a list of stored function associations. Each computed function association is defined as:

$$< computed\ association > \\ ::= < behavior\ reference\ list > \textbf{ with } < function\ reference >$$

and the stored function association is one of the following[1]:

$$< get\ association > ::= < behavior\ reference\ list > \textbf{ with } GET$$

$$< set\ association > ::= < behavior\ reference\ list > \textbf{ with } SET$$

The *type reference* in the *association clause* specifies the type within which the associations are to be defined. Thus, one association statement can be used to define associations between behaviors and function objects only within a single type. *Computed list* in this statement associates computed functions with the behaviors in a given type. Every element of this list consists of a *behavior reference list* and one *function reference*. Behavior names together with the type reference (from the *association clause*) uniquely specify behavior objects in the objectbase. Behaviors which are in the same *behavior reference list* are associated with the same function object whose reference is given after the *with clause*. In other words, they all have the same implementation. The *stored* clause in this statement associates stored functions with the behaviors in the given type. However, there are

[1] The full syntax of the association statement is given in Appendix A.

two different semantics of behaviors which can be associated with stored functions. The semantics of behaviors can be either to retrieve the object which is stored, or to store it (set its value). Thus, stored function association is made up either of the *get* sequence, or of the *set* sequence. Moreover, if there is one *get* (*set*) association, there must be at least one *set* (*get*) association and vice versa. Furthermore, there can be one or more get/set clauses within the same association statement, they all correspond to the same stored function. Thus, one association statement creates at most one stored function. In order to associate behaviors in a specific type with different stored functions, separate association statements must be used.

In the following example, the association statement is used for two different pairs of behaviors: one with the result type `T_string`, and the other with the result type `T_date`. It is incorrect, as the behaviors have different result type (semantics), thus they should be associated with two different stored functions.

associate in `T_person`
B_getName **with** *GET*, *B_setName* **with** *SET*,
B_getBrtday **with** *GET*, *B_setBrtday* **with** *SET*;

The example below illustrates associations which can be done within the `T_person` type. Two association statements are used to ensure that two different stored function are created.

associate in `T_person`
B_getName **with** *GET*, *B_setName* **with** *SET*;

associate in `T_person`
B_getBrtday **with** *GET*, *B_setBrtday* **with** *SET*,
B_age **with** *f_age*, *B_spouse* **with** *f_spouse*;

The first association statement creates a pair of stored functions. The function to retrieve the object is referenced by *GET*, while the function to store the object is referenced by *SET*. Thus, behavior *B_getName* is associated with *GET*, and behavior *B_setName* is associated with *SET* in the `T_person` type. The second statement creates a new pair of stored functions, and associates behaviors *B_getBrtday* and *B_setBrtday* with *GET* and *SET* respectively. This statement also associates behaviors *B_age* and *B_spouse* with computed functions referenced by *f_age* and *f_spouse* respectively.

The next TDL statement is a *class declaration* statement which is used to create a new class object in a TIGUKAT objectbase and to associate it with an existing type. When a class is created, it is assumed that the corresponding type is correctly and fully defined, meaning that all behaviors are specified and the associations between behaviors and functions are completed. An error condition is raised if there exists a behavior within a given type which does not have an associated function

defined when a request to create a class for this type is posted. The general syntax of the class declaration statement:

$$< class\ declaration > ::= \textbf{create class}\ [< new\ reference >]$$
$$\textbf{on} < type\ reference >$$

The class reference in this statement declares a reference to a new class object. However, this specification is optional; if not provided, the class can still be accessed through its type. The following example illustrates two different ways to create a class object for the `T_person` type.

create class C_person on `T_person`;

or the other way to create a class object is:

create class on `T_person`;

Both of these statements create class object for the `T_person` type. However, the first statement not only creates a class object and associates it with the type object `T_person`, but also declares a separate reference **C_person** to the class object. The type object and the class object, in this case, have unique direct references. The second statement creates a class object for the `T_person` type, and associates it with this type. Although, there is no direct reference to the class object, it can still be accessed through the *B_classof* behavior defined on the `T_type` type.

The last TDL statement is a *collection declaration* statement which creates new collection objects. The general syntax of this statement is as follows:

$$< collection\ declaration > ::= \textbf{create collection} < new\ reference >$$
$$\textbf{type} < type\ reference >$$
$$[\textbf{with} < object\ list >]$$

The *create collection* clause in this statement declares a new reference to a collection object. The *type clause* specifies the member type of collection elements, while the *with clause* initializes the collection with objects given in the list. The following example illustrate how to create a new collection in TDL.

Example 3.1 Let assume that the references: *john, paul* and *peter* reference the objects of the `T_person` type. A new collection from these objects can be created by using the following statement:

create collection *students*
type `T_person`
with *john, paul, peter*

In summary, TDL is used to create type, class, behavior and function objects in a TIGUKAT objectbase, and to define relationships among them. To create a new type object, the type reference

and the list of immediate supertypes must be given. Behaviors of a new type can be either defined during the type declaration, or later using behavior manipulation statements. There are stored and computed functions in the TIGUKAT objectbase. Stored functions cannot be explicitly declared, they are created during the association process. Computed functions are explicitly declared and created using computed function declaration statements. Associations between behaviors and functions are defined by association statements. Finally, class objects are created using a class declaration statement. However, a new class can be created for an existing type only if this type is completely defined, meaning that all behaviors have functions associated with them. Otherwise, an error occurs. Example 3.1 illustrates the complete process of creating new type, class, behavior and function objects in the GIS which is defined in Example 2.1i, and defining associations among them.

Example 3.2 Define two types `T_dwelling` and `T_house` for the GIS. The type `T_dwelling` is a direct subtype of the `T_Object` type, and it has two behaviors: *B_address* and *B_inZone*. The type `T_house` is a subtype of `T_dwelling` type that has one additional behavior: *B_mortgage*. *B_inZone* and *B_address* in `T_house` are inherited form the type `T_dwelling`. Thus, the definition of these two types in TDL is:

create type `T_dwelling`
under `T_Object`
public: *B_setAddr*(`T_string`):`T_string`,
B_getAddr:`T_string`,
B_inZone: `T_land`;

create type `T_house`
under `T_dwelling`
public: *B_setMortgage*(`T_real`):`T_real`,
B_getMortgage:`T_real`;

Since in type `T_dwelling`, *B_address* is to be associated with a stored function, two behaviors: *B_setAddr* and *B_getAddr* are defined instead of *B_address*. Although these behaviors have different semantics, they will be associated with the same stored function. Since we would like a slightly different implementation for *B_inZone* in `T_house` then the one in `T_dwelling`, we declare two different function objects for them:

external function *dw_inZone*(self:`T_dwelling`) **:** `T_land`;
external function *hs_inZone*(self:`T_house`) **:** `T_developed`;

We assume, that functions *dw_inZone* : `T_land` and *hs_inZone* : `T_developed` already exist somewhere in the system and now we have local references to them. There are two stored functions for

the '*address*' behaviors in `T_dwelling` type, and the *mortgage* behavior in the `T_house` type. They will be created during the association process. Now, the associations between behavior objects and function objects can be specified.

associate in `T_dwelling`
B_inZone **with** *dw_inZone*,
B_setAddr **with** *SET*,
B_getAddr **with** *GET*;

associate in `T_house`
B_inZone **with** *hs_inZone*,
B_setMortgage **with** *SET*,
B_getMortgage **with** *GET*;

Finally, as all associations are done, class objects for the newly created types can be created.

create class C_dwelling on `T_dwelling`;
create class C_house on `T_house`;

3.3 TIGUKAT Query Language

The main function of TQL is to retrieve and to manipulate objects in a TIGUKAT objectbase. Its syntax is based on the SQL *select-from-where* structure [Dat87], while its semantics is defined in terms of the object calculus. In fact, there is a complete reduction from TQL to object calculus, thus the semantics of the language is formally specified.

3.3.1 Design Decisions

TQL is based on the SQL *select-from-where* structure. We have decided to adopt this structure for various reasons. First of all, SQL is the standard language for relational systems. Second, current work on SQL3 attempts to extend its syntax and its semantics to fulfill requirements of object-oriented systems [Gal92]. Finally, any syntax of a query statement must provide a way to specify the three basic components of the query block. Instead of designing a new structure to achieve the same result, we have adopted the one which is already successful in other systems.

TQL extends the basic SQL structure by accepting path expressions (implicit joins [KBC+89]) whenever it makes sense. Thus, path expressions can be used in the *select clause* to navigate trough the schema. They can be used in the *from clause* if the result of the application of behaviors is a finite collection. They can also be used in the *where clause* as predicates. Since the object equality is defined on the primitive type `T_object`, explicit joins are also supported by TQL. Queries operate

on finite collections and they always return new collections as results. Thus, query results are queryable. Also, queries can appear in the *from* and *where clauses* of other queries (the concept of nested queries is supported). Objects can be queried regardless of whether they are persistent or transient. Finally, TQL is built on top of the object calculus, which makes the semantics of the language well defined.

It should be noted here, that the syntax for the application of aggregate functions is not explicitly supported by TQL. However, as the underlying model is purely behavioral, these functions are defined as behaviors on the `T_finCollection` primitive type. They can be applied to any collection including those returned as a result of a query.

3.3.2 The Syntax of TIGUKAT Query Language

There are four basic TQL operations: **select, insert, delete**, and **update**. In addition, there are three binary operations: **union, minus**, and **intersect**. Each of these statements operates on a set of input collections and returns a collection as a result. However, only the semantics of the *select, union, minus*, and *intersect* statements are currently well defined. The definition of the semantics for the *insert, delete* and *update* statements involves the specification of the update semantics in the TIGUKAT object model. These aspects of the object model and the associated language constructs are currently being developed and will be presented in future reports.

The basic query statement of TQL is the *select statement*. It operates on a set of input collections and it always returns a new collection as the result. The general syntax of the select statement is:

$<$ *select statement* $>$ ::= **select** $<$ *object variable list* $>$
[**into** [**persistent** [**all**]] $<$ *collection name* $>$]
from $<$ *range variable list* $>$
[**where** $<$ *boolean formula* $>$]

The *select clause* in this statement identifies objects which are to be returned in a new collection. There can be one or more object variables in this clause. They can be in the form of simple variables, path expressions (which are equivalent to Bspecs defined in Chapter 2, Section 2.2), index variables, or constants. They correspond to free variables in object calculus formulas. The *into clause* declares a reference to a new collection returned as a result of the query. If the *into clause* is not specified, a new collection is created; however, there is no reference to it. This is especially useful when a query is embedded in some other query and the collection returned as a result does not require an explicit reference. Also, as the TIGUKAT language supports the assignment statement, a variable reference can be bound to the result of a query. Therefore, the *into clause* can be omitted. In addition, the result collection can be made persistent by specifying it in the *into clause*. The *persistent clause* makes only the container object persistent in the objectbase, while a *persistent all* makes all elements of the collection persistent as well. If elements of the collections are themselves collections, *persistent*

all makes all the objects in those collections persistent in a recursive fashion. The *from clause* declares ranges of object variables in the *select* and *where* clauses. Every object variable can range over either an existing collection, or a collection returned as a result of a subquery, while a subquery can be either given explicitly, or as a reference to a query object. It is useful to distinguish between constant references to collections and variable references to collections. A constant reference is a reference which does not change during the execution of a query. In particular, it can be a reference to a collection that is a result of the evaluation of a subquery. A variable reference to a collection is a reference which can change during the execution of a query. The range variable in the *from clause* has the following syntax:

$$< \textit{range variable} > ::= < \textit{variable list} > \textbf{in} < \textit{collection reference} > [\ +\ ]$$
$$< \textit{collection reference} > ::= < \textit{term} >$$
$$\mid\ (\ < \textit{query statement} >\ )$$

The collection reference in the range variable definition can be followed by a plus '+' which refers to a shallow extent of a collection or a class. If it is not specified, a deep extent is assumed by default. In case of collections, the deep and shallow extents are equivalent.

The *term* in the collection reference definition is either a constant reference to a collection, a variable reference, or a path expression.

The *where clause* defines a boolean formula which must be satisfied by objects returned by a query. Boolean formulas in TQL are defined in a similar (recursive) fashion as the *formulas* of the object calculus. In fact, there is a complete correspondence between the formulas of the query language and the object calculus *formulas*. Boolean formulas of the TQL have the following syntax:

$$< \textit{boolean formula} > ::= < \textit{atom} >$$
$$\mid\ \textbf{not} < \textit{boolean formula} >$$
$$\mid\ < \textit{boolean formula} > \textbf{and} < \textit{boolean formula} >$$
$$\mid\ < \textit{boolean formula} > \textbf{or} < \textit{boolean formula} >$$
$$\mid\ (\ < \textit{boolean formula} >\ )$$
$$\mid\ < \textit{exists predicate} >$$
$$\mid\ < \textit{forAll predicate} >$$
$$\mid\ < \textit{boolean path expression} >$$

An atom in the TQL boolean formula is one of the following:

$$< \textit{atom} > ::=\ \ < \textit{term} > = < \textit{term} >$$
$$\mid\ < \textit{term list} > \textbf{in} < \textit{collection reference} > [+]$$

where the *term* is a variable reference, a constant reference or a path expression, and the *collection reference* is the same as in the range variable definition.

Two special predicates are added to boolean formulas of the query language in order to express existential and universal quantification. The existential quantifier is expressed by the *exists predicate* which is of the following format:

$$< exists\ predicate > ::= \textbf{exists} < collection\ reference >$$

The *exists predicate* is *true* if the collection returned by the subquery is not empty. Otherwise, the predicate is *false*. The *exists predicate* is unnecessary in the TQL, as every query with this predicate in the *where clause* can be transformed to the equivalent query without this predicate. However, we have decided to include it in TQL, so users are not forced to write queries in prenex normal form.

The universal quantifier is expressed by the *forAll predicate* which has the following structure:

$$< forAll\ predicate > ::= \textbf{forAll} < range\ variable\ list > < boolean\ formula >$$

The syntax of the *range variable list* is the same as in the *from clause* of the select statement. It defines variables which range over a specified collection. The *boolean formula* is evaluated for every possible binding of every variable in this list. Thus, the entire *forAll predicate* is *true*, if for every element in every collection in the range variable list, the boolean formula evaluates to *true*. If, on the other hand, there exists at least one element in any collection such that the formula evaluates to *false*, then the whole predicate is *false*.

Example 3.3

$$\textbf{forAll}\ p\ \textbf{in}\ P,\ q\ \textbf{in}\ Q\quad F(p,q)$$

This predicate is *true* if for every element of the collection P, and for every element of the collection Q, the formula $F(p,q)$ evaluates to *true* (the formal semantics of this predicate given in Section 3.3.3).

It should be noted here that collections in the *range variable list* can be given explicitly as constant references to collection objects, or implicitly as queries (just as it is in the *from clause* of the select statement).

The last part of the definition of the boolean formula is the *boolean path expression* which is equivalent to the following formula:

$$< path\ expression > = \textbf{TRUE/FALSE}$$

However, to avoid such artificial constructs, we include boolean path expressions in the definition of the TQL formula under two conditions. First, all invoked functions are *side-effect-free*. Second, the result type of the whole path expression is of a boolean type.

So far, a select statement with only one simple object variable in the *select clause* was discussed. There can be one or more objects of various formats in this clause. The object in the *select clause* has the syntax:

$$< object\ variable > ::= [(< cast\ type >)] < term > \\ \mid < index\ variable >$$

where a *term* is either a constant reference to an object, variable reference to an object, or a path expression. The first definition of the object variable corresponds to a standard reference to an object. The projection type enclosed in brackets (which is optional in this clause) defines the type of elements of a result collection. However, it makes sense only if this type is a supertype of the type of an object which is after the cast type. It acts as a behavioral projector or a generalization operator [Gal92]. The interface of objects returned in the result collection is a subset (not necessarily a proper one) of the interface of objects given in the *select clause*. This subset is defined by the interface of the type enclosed in brackets. The construct used to project behaviors is similar to the cast function in [Gal92], and equivalent to the cast operator in [Bla91]. If it is is not given, the type of the result collection is inferred from the types of collections defined as ranges. The second part of the definition of an object variable is an index variable. It has the following format:

$$< index\ variable > ::= < identifier > [< behavior\ name\ list >]$$

The role of an index variable is to specify the behaviors which are applicable to objects in the result collection. The idea is the same as in the projection type; however, all behaviors in an index variable must be given explicitly in the *behavior name list*. Thus, objects in the result collection can have different types then original ones.

TQL supports three binary operations: **union, minus**, and **intersect**. Similarly to a select statement, they operate on the collection of objects and always return new collections as result. The syntax of these statements is:

$$< collection\ reference > \quad \textbf{union} \quad < collection\ reference > \\ < collection\ reference > \quad \textbf{minus} \quad < collection\ reference > \\ < collection\ reference > \quad \textbf{intersect} \quad < collection\ reference >$$

A *collection reference* in a TQL binary statement is either a constant reference to a collection object, or it is a query.

3.3.3 The Formal Semantics of TQL

The semantics of TQL are defined in terms of the object calculus. It is shown in this section that every TQL statement corresponds to an object calculus expression; thus there is a complete reduction from TQL to the object calculus.

Throughout this section the following notation is used. Every TQL select statement of the form:

select $p_1,\ p_2,\ \ldots p_k$
into *newCollection*

from p_1 **in** P_1, ..., p_k **in** P_k, q_1 **in** Q_1, ..., q_n **in** Q_n
where $F(p_1, ..., p_k, q_1, ..., q_n)$

is referred to as $S(p_1, ..., p_k)$. In other words, queries can be modelled as functions $S(p_1, ..., p_k)$ which operate upon one or more collections, and return collections as results. A list which is returned as a result collection is made up of objects referenced by $p_1, ..., p_k$, and is denoted as $< p_1, ..., p_k >$. Furthermore, for groups of quantifiers like $\exists p_1, ..., \exists\ p_k$ or $\forall p_1, ..., \forall p_k$, the shorthand notation is used: $\exists < p_1, ..., p_k >$ and $\forall < p_1, ..., p_k >$ respectively. Finally, $< p_1, ..., p_k >=< x_1, ..., x_k >$ is a short notation for $p_1 = x_1, ..., p_k = x_k$.

It is shown in this section, that every select statement $S(p_1, ..., p_k)$ corresponds to the object calculus expression: $\{< p_1, ..., p_k > \mid \phi(< p_1, ..., p_k >)\}$. The *select clause* in $S(p_1, ..., p_k)$ defines the free variables of the object calculus formula. The *from clause* specifies the ranges of variables which can either be given explicitly as constant references to collections, or implicitly in the form of subqueries. If the range variable is defined over a constant collection reference, then it corresponds to a **range atom** (e.g. p **in C_person** $\equiv$ **C_person**(p)) in the object calculus. If it ranges over a collection defined by a variable or a path expression then it corresponds to a **membership atom** (p **in** $q.kids() \equiv (p \in q.kids)$). Otherwise, in case of subqueries, the semantics of the range variable is defined by a complex object calculus formula. However, as shown below, every query which has a subquery in the *from clause* can be rewritten as an equivalent flat query.

Theorem 3.1 Every TQL query $S_p(p_1, ..., p_k)$ with nested queries in the *from clause* can be rewritten as an equivalent flat query.

Proof: Every query with a subquery in the *from clause* is expressed in TQL as[2]:

$$
\begin{aligned}
S(p_1, ..., p_k) \equiv\ & \textbf{select}\ p_1, ..., p_k \\
& \textbf{from}\ p_1\ \textbf{in}\ \#P_1\ , ...,\ p_i\ \textbf{in}\ \#P_i, \\
& \quad p_{i+1}\ \textbf{in}\ S_{i+1}(q_{i+1})\ , ...,\ p_k\ \textbf{in}\ S_k(q_k), \\
& \quad r\ \textbf{in}\ \#R \\
& \textbf{where}\ F(p_1, ..., p_k, r)
\end{aligned}
$$

which is equivalent to the object formula:

$$
\begin{aligned}
\exists p_1 ... \exists p_k\ \ & (P_1(p_1) \wedge ... \wedge P_i(p_i) \wedge \\
& p_{i+1}\ \in\ S_{i+1}(q_{i+1}) \wedge ... \wedge p_k\ \in\ S_k(q_k) \wedge \exists r(R(r) \wedge F(p_1, ..., p_k, r)))
\end{aligned}
\tag{3.1}
$$

$P_1, ..., P_i$ in this query are constant references to collections, r represents all variables which appear in the query, but not in the *select clause*, and $S_{i+1}(q_{i+1}), ..., S_k(q_k)$ represent subqueries. Thus, every S_{i+j} $(j = 1, ..., k - i)$ is also a query, and it is represented in TQL as:

[2] For brevity, we assume that all collection references P in the $from\ clause$ are constants. It can be easily generalized to include other cases; however, this does not effects the proof.

$$S_{i+j}(q_{i+j}) \equiv \quad \textbf{select } q_{i+j}$$
$$\textbf{from } q_{i+j} \textbf{ in } \#\, Q_{i+j}, r_{i+j} \textbf{ in } \#\, R_{i+j}$$
$$\textbf{where } F_{i+j}(q_{i+j}, r_{i+j})$$

which is equivalent to the following object calculus formula:

$$S_{i+j}(q_{i+j}) \equiv \exists q_{i+j}(Q_{i+j}(q_{i+j}) \wedge \exists r_{i+j}(R_{i+j}(r_{i+j}) \wedge F_{i+j}(q_{i+j}, r_{i+j})))$$

Furthermore, every subformula in the *from clause* which is in the form: p_{i+j} **in** $S_{i+j}(q_{i+j})$ is equivalent to:

$$p_{i+j} \textbf{ in } S_{i+j}(q_{i+j}) \equiv \tag{3.2}$$
$$\exists q_{i+j}(Q_{i+j}(q_{i+j}) \wedge \exists r_{i+j}(R_{i+j}(r_{i+j}) \wedge F_{i+j}(q_{i+j}, r_{i+j})) \wedge \; p_{i+j} = q_{i+j})$$

In 3.2, every q_{i+j} $(j = 1, ..., k-i)$ can be replaced by p_{i+j} yielding an equivalent formula:

$$p_{i+j} \textbf{ in } S_{i+j}(q_{i+j}) \equiv Q_{i+j}(p_{i+j}) \wedge \exists r_{i+j}(R_{i+j}(r_{i+j}) \wedge F_{i+j}(p_{i+j}, r_{i+j}))$$

Thus, by replacing each p_{i+j} **in** $S_{i+j}(q_{i+j})$ in 3.1 the following equivalent formula is obtain:

$$\exists p_1 ... \exists p_k(P_1(p_1) \wedge ... \wedge P_i(p_i) \wedge \tag{3.3}$$
$$(Q_{i+1}(p_{i+1}) \wedge \exists r_{i+1}(R_{i+1}(r_{i+1}) \wedge F_{i+1}(p_{i+1}, r_{i+1})) \wedge ... \wedge$$
$$(Q_k(p_k) \wedge \exists r_k(R_k(r_k) \wedge F_k(p_k, r_k))) \wedge \exists r(R(r) \wedge F(p_1, ..., p_k, r)))$$

The formula 3.3 is in conjunctive form; therefore, changing the order of predicates results in a logically equivalent formula. Thus, in a new formula, all range atoms of the form $P_i(p_i),\ Q_i(q_i),\ R_i(r_i)$ are put together, and all well-formed formulas of the form $F(p_1, ...p_k, r), ..., F_i(p_i, r_i)$ are put together. The equivalent formula is as follows:

$$\exists p_1 ... \exists p_k(P_1(p_1) \wedge ... \wedge P_i(p_i) \wedge$$
$$Q_{i+1}(p_{i+1}) \wedge ... \wedge (Q_k(p_k) \wedge$$
$$\exists r_{i+1}(R_{i+1}(r_{i+1}) \wedge ... \wedge \exists r_k(R_k(r_k) \wedge$$
$$F_{i+1}(p_{i+1}, r_{i+1}) \wedge ... \wedge F_k(p_k, r_k) \wedge F(p_1, ..., p_k, r))))) ...)$$

Thus, the original query $S(p_1, ..., p_k)$ can be rewritten to the following form:

$$S'(p_1, ..., p_k) \equiv \textbf{ select } p_1, ..., p_k$$
$$\textbf{from } p_1 \textbf{ in } \#P_1 \ , ..., \ p_i \textbf{ in } \#P_{i+1},$$
$$p_{i+1} \textbf{ in } \#Q_{i+1} \ , ..., \ p_k \textbf{ in } \#Q_k,$$
$$r_{i+1} \textbf{ in } \#R_{i+1}, ..., r_k \textbf{ in } \#R_k, r \textbf{ in } \#R$$
$$\textbf{where } F_{i+1}(p_{i+1}, r_{i+1}) \wedge ... \wedge F_k(p_k, r_k) \wedge F(p_1, ..., p_k, r)$$

□

From now on, we assume that all ranges in the *from clause* are defined by either the constant references to a collection corresponding to the **range atoms** in the object calculus formulas, or by variable references corresponding to **membership atom** of the object calculus. Consider the following example:

Example 3.4

select p

from p **in** $\#P$,

q **in** ($\underbrace{\textbf{select } v \textbf{ from } v \textbf{ in } \#V, w \textbf{ in } \#W \textbf{ where } F_1(p,v,w)}_{S_p}$)

where $F_2(p,q)$

This query has a nested query (S_p) in the *from clause* which is in the format:

select $\underbrace{v}_{a}$

from $\underbrace{v \textbf{ in } \#V, w \textbf{ in } \#W}_{b}$

where $\underbrace{F_1(p,v,w)}_{c}$

Variables in the *select clause* correspond to free variables of the calculus expression (part (a)):

$$\{\underbrace{v}_{a} \mid \underbrace{V(v) \wedge \exists w(W(w)}_{b} \wedge \underbrace{F_1(p,v,w))}_{c}\}$$

The *from clause* specifies ranges of the object variables. In this case, all range variables correspond to *range atoms* of the object calculus, and build the second part (b) of the calculus expression. Finally, the *where clause* contains a boolean formula, which correspond to a well-formed formula of the calculus, and makes up the third (c) part of the query expression.

In a similar fashion, a calculus expression is built for the entire query. There is one variable p in the *select clause*, which corresponds to a free variable of the calculus formula. The *from clause* defines ranges of variables used in the *select* and *where clauses*. In this case the range of the variable p is a constant reference, while the range of q is given in the form of a subquery (S_p) which corresponds to the calculus formula:

$$(q \textbf{ in } S_p) \equiv \exists v\ (V(v) \wedge \exists w(W(w) \wedge F_1(p,v,w) \wedge q = v)).$$

The *where clause* adds the last part $F_2(p,q)$ to the calculus expression. Thus, the final form of this expression is:

$$\{\ p \mid P(p) \wedge \exists q(\exists v(V(v) \wedge \exists w(W(w) \wedge F_1(p,v,w) \wedge q = v))) \wedge F_2(p,q)\}$$

This formula can be transformed to:

$$\exists q(\exists v\ (V(v) \wedge \exists w(W(w) \wedge F_1(p,v,w) \wedge q = v))) \\ \equiv \exists v\ (V(v) \wedge \exists w(W(w) \wedge F_1(p,v,w))).$$

Thus, the calculus expression for the whole query is:

$$\{ \underbrace{p}_{a} \mid (\underbrace{P(p) \wedge \exists v(V(v) \wedge \exists w(W(w)}_{b} \wedge \underbrace{F_1(p,v,w) \wedge F_2(p,v)}_{c}))\}$$

The query can be rewritten in TQL as:

select $\underbrace{p}_{a}$

from $\underbrace{p \textbf{ in } \#P, v \textbf{ in } \#V, w \textbf{ in } \#W}_{b}$

where $\underbrace{F_1(p,v,w) \textbf{ and } F_2(p,v)}_{c}$

$\diamond$

Next, it is shown that there is a direct correspondence between a TQL *boolean formula* in the *where clause* and the object calculus well-formed formulas.

Theorem 3.2 Every boolean formula in the *where clause* of the select statement corresponds to a well-formed formula in the object calculus.

Proof: A boolean formula in TQL has the following syntax:

$< boolean\ formula > ::= < atom >$
$\mid < exists\ predicate >$
$\mid < forAll\ predicate >$
$\mid < boolean\ path\ expression >$
$\mid$ **not** $< boolean\ formula >$
$\mid < boolean\ formula >$ **and** $< boolean\ formula >$
$\mid < boolean\ formula >$ **or** $< boolean\ formula >$
$\mid$ **(** $< boolean\ formula >$ **)**

a) The atom in a TQL boolean formula is one of the following:

$< atom > \ ::= < term > = < term >$
$\mid < term\ list >$ **in** $< collection\ reference >$ [+]

The first atom is equivalent to the **equality atom** of the object calculus. If the term on the left hand side of the equality atom is a variable, then it corresponds to a **generating atom** of the object calculus. The semantics of the second atom depends on the *collection reference*. If it is a constant reference to a collection, then it corresponds to a **range atom** in the object calculus. Otherwise, it corresponds to a **membership atom**.

b) The existential quantifier in TQL is expressed by the *exists predicate*:

$$< \textit{exists predicate} > ::= \textbf{exists} < \textit{collection reference} >$$

The *exists predicate* is *true* if the referenced collection is not empty. Otherwise, the predicate is *false*. The collection reference in this predicate is either a constant reference to a collection object, or it is a query which returns a collection as a result. In the first case, the exists predicate has the format: **exists** P, and it is equivalent to the object formula $\exists x\ P(x)$. In the second case, when the collection reference is given implicitly by a query, the exists predicate has the form: **exists** $S(p_1, ..., p_k)$. Then, it corresponds to the following calculus formula:

$$\exists < x_1, ..., x_k > (\exists < p_1, ..., p_k > (S(p_1, ..., p_k)) \\ \wedge < p_1, ..., p_k >=< x_1, ..., x_k >)$$

However, the *exists predicate* is unnecessary in TQL. Every query with this predicate in the *where clause* can be transformed to an equivalent flat query. We decided to include it in the language, so users are not forced write queries in prenex normal form. Consider the example:

Example 3.5

$S(p) \equiv$ **select** p
from p **in** $\#P, r$ **in** $\#R$
where $F_1(p, r)$ **and exists**
($\underbrace{\textbf{select}\ v\ \textbf{from}\ v\ \textbf{in}\ \#V, w\ \textbf{in}\ \#W\ \textbf{where}\ F_2(p, v, w)}_{S(v)}$)

The subquery $S(v)$ in the *where clause* corresponds to the object calculus formula:

$$S(v) \equiv \exists v(V(v) \wedge \exists w(W(w) \wedge F_2(p, v, w)))$$

Thus, the entire query in the object calculus can be expressed by:

$$\exists p\ (P(p) \wedge \exists r(R(r) \wedge F(p, r) \wedge \exists v(V(v) \wedge \exists w(W(w) \wedge F_2(p, v, w)))))$$

Applying formula preserving transformations, the above formula can be rewritten:

$$\exists p(P(p) \wedge \exists r(R(r) \wedge \exists v(V(v) \wedge \exists w(W(w) \wedge F_1(p, r) \wedge F_2(p, v, w)))))$$

Thus, in TQL, $S(p)$ can be expressed as:

$S'(p) \equiv$ **select** p
from p **in** $\#P, r$ **in** $\#R, v$ **in** $\#V, w$ **in** $\#W$
where $F_1(p, r)$ **and** $F_2(p, v, w)$

$\diamond$

c) The universal quantifier is represented in TQL by the *forAll predicate*. It has the following format:

$$< forAll\ predicate > ::= \textbf{forAll} < range\ variable\ list > \\ < boolean\ formula >$$

This predicate is *true*, if for every element in every collection in the range variable list, the boolean formula evaluates to *true*. If, on the other hand, there exists at least one element in any collection such that the formula evaluates to *false*, then the whole predicate is *false*. Again, the collection references in the range variable list are either constant references to collection objects, or they are given by queries. Therefore, in the general case, this predicate is:

$$\textbf{forAll}\ p_1\ \textbf{in}\ \#P_1, ..., p_i\ \textbf{in}\ \#P_i, \\ p_{i+1}\ \textbf{in}\ S_{i+1}(q_{i+1}), ..., p_k\ \textbf{in}\ S_k(q_k) \\ F(p_1, ..., p_k)$$

where $P_1, ...P_i$ are constant references to collections, and $S_{i+1}(q_k), ...,$
$S_k(q_k)$ are queries. The following object calculus formula is equivalent to this predicate:

$$\forall\ p_1 ... \forall\ p_k((\neg P_1(p_1) \vee ... \vee \neg P_i(p_i) \\ \vee \neg(S_{i+1}(q_{i+1}) \wedge p_{i+1} = q_{i+1}) \vee ... \vee \neg(S_{i+1}(q_{i+1}) \wedge p_{i+1} = q_{i+1})) \\ \vee\ F(p_1, ..., p_k))$$

d) The next part of the definition of a boolean formula is a *boolean path expression*. In general, path expressions in the TIGUKAT language correspond to Bspecs defined in [PLÖS93]. Boolean path expressions are Bspecs which evaluate to objects of `T_boolean` type. A boolean formula which is given in the form of a boolean path expression is *true* if the path expression evaluates to a constant object **TRUE**. Otherwise, it is *false*. Therefore, the boolean path expression in the TQL boolean formula definition can be considered as a shorthand notation for an equality atom of the form:

$$< path\ expression > = \ \textbf{TRUE/FALSE}$$

Thus, boolean path expressions correspond to equality atoms in the object calculus.

e) The remaining definitions of TQL boolean formulas correspond directly to the recursive definition of a well-formed formula in the object calculus. Thus, every TQL boolean formula is equivalent to an object calculus well-formed formula. □

As shown in Section 3.3.2, the select clause is made up of one or more object terms. Each term is either a constant reference to an object, a variable reference to an object, path expression, or an index variable. In addition, each term can be proceeded by a cast type which extracts behaviors

from it. However, the object calculus allows constants, variables, Bspecs and index variable as free variables in its formulas as well. Thus, every constant reference in TQL corresponds to a constant in the object calculus, a variable reference is equivalent to a variable in the object calculus, and a path expression in TQL corresponds to a Bspec. TQL index variables extract certain behaviors from object's types, thus they correspond to **index variables** of the object calculus. Finally, each term can be preceded by a cast type which extracts (generalizes) behaviors from an object type. Thus, a TQL cast type and the following term correspond to an **index variable** in the object calculus as well.

Example 3.6

`T_person`: subtype of `T_object` has the following behaviors:

{*B_name*, *B_age*} plus all behaviors inherited from `T_object`

`T_student`: subtype of `T_person` has the following native behaviors:

{ *B_stId*, *B_department*, *B_gpa* }

Thus, the following TQL query:

select (`T_person`) p

from p **in C_student**

where $F(p)$

corresponds to the following object calculus formula:

$\exists p_{[B\_name, B\_age]}$ (**C_student**(p) $\wedge$ $F(p)$)

which corresponds to the following calculus expression:

$\{p[B\_name, B\_age] \mid \textbf{C\_student}(p) \wedge F(p)\}$

$\diamond$

Theorem 3.3 Every select statement in TQL has an equivalent object calculus expression.

Proof: It follows directly from Theorem 3.1 and Theorem 3.2. Every select statement can be expressed as:

select $p_1,\ p_2,\ \ldots p_k$

from p_1 **in** $\#P_1,\ \ldots, p_k$ **in** $\#P_k, q_1$ **in** $\#Q_1, \ldots, q_n$ **in** $\#Q_n$

where $F(p_1, \ldots, p_k, q_1, \ldots, q_n)$

where $p_1,\ p_2,\ \ldots p_k$ are free variables within the query, $P_1, \ldots, P_k, Q_1, \ldots, Q_n$ are constant references to collections, and $F(p_1, \ldots, p_k, q_1, \ldots, q_n)$ is a TQL boolean formula. Thus, the whole query corresponds to the object calculus expression of the form:

$\{\ p_1, \ldots, p_k \mid P_1(p_1) \wedge\ \ldots\ \wedge P_k(p_k) \wedge$

$\exists q_1, \ldots, \exists q_n (Q_1(q_1) \wedge\ \ldots\ \wedge Q_n(q_n) \wedge F(p_1, \ldots, p_k, q_1, \ldots, q_n))\ \}.$ $\square$

Summarizing, the *select clause* of the select statement defines the free variables of an object calculus formula, which correspond to variables of the target list in the object calculus expression. The *from clause* declares a range of variables which correspond to range atoms of an object calculus formula. Finally, the *where clause* specifies a boolean condition that corresponds to an object calculus well-formed formula. Therefore, the semantics of every select statement in TQL are well defined.

Theorem 3.4 Every binary operation in TQL has an equivalent object calculus expression.

Proof: The binary operations in TQL have the following syntax:

$$\begin{array}{lll} < collection\ reference > & \textbf{union} & < collection\ reference > \\ < collection\ reference > & \textbf{minus} & < collection\ reference > \\ < collection\ reference > & \textbf{intersect} & < collection\ reference > \end{array}$$

Thus, in the object calculus they are expressed by simple calculus expressions: $\{\ o \mid P(o) \vee Q(o)\}$, $\{\ o \mid P(o) \wedge \neg Q(o)\}$, $\{\ o \mid P(o) \wedge Q(o)\}$, where P is a reference to the first collection in the binary statement, and Q is a reference to the second collection. □

Theorem 3.5 The reduction from TQL to the object calculus is complete.

Proof: It follows directly from Theorems 3.1, 3.2, 3.3 and 3.4. □

The following examples illustrate queries, which are formally specified in Examples 2.3-2.9, expressed in TQL.

Example 3.7 The query in Example 2.3: *Return land zones valued over $100,000 or covering an area over 1000 units* is expressed in TQL as:

select *o*
from *o* **in C_land**
where (*o*.*B_value*() > 100000) **or** (*o*.*B_area*() > 1000)

Example 3.8 The query in Example 2.4: *Return all zones that have people living in them (the zones are generated from person objects)* is expressed in TQL as:

select *o*
from *q* **in C_person**
where (*o* = *q*.*B_residence*().*B_inzone*())

Example 3.9 The query in Example 2.5: *Return the maps with areas where citizens over the age of 65 years live* is expressed in TQL as:

select o
from o **in C_map**
where exists (**select** p
from p **in C_person**, q **in C_dwelling**
where (p.*B_age*() ≥ 65 **and** $q = p$.*B_residence*()
and q.*B_inzone*() $\in o$.*B_zones*()))

Example 3.10 The query in Example 2.6: *Return all maps that describe areas strictly above 5000 feet* is expressed in TQL as:

select o
from o **in C_map**
where forAll p **in** (**select** q
from q **in C_altitude**
where $q \in o$.*B_zones*())
p.*B_low*() > 5000

Example 3.11 The query in Example 2.7: *Return the dollar values of the zones that people live in* is expressed in TQL as:

select p.*B_residence*().*B_inzone*().*B_value*()
from p **in C_person**

Example 3.12 The query in Example 2.8: *Return the zones that are part of some map and are within 10 units from water. Project the result over B_title and B_area* is expressed in TQL as:

select o[*B_title*, *B_area*]
from p **in C_map**, o **in** p.*B_zones*, q **in C_water**
where o.*B_proximity*(q) < 10

Example 3.13 The query in Example 2.9: *Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map* is expressed in TQL as:

select p, q.*B_title*()
from p **in C_person**, q **in C_map**
where p.*B_residence*().*B_inZone*() $\in q$.*B_zones*()

3.4 TIGUKAT Control Language

The last part of the TIGUKAT Language is the TIGUKAT Control Language (TCL) which consists of operations performed on session objects. Since everything in TIGUKAT is treated as a first class object, sessions are also represented by objects in the objectbase. They can be referenced, opened, accessed and closed. Session objects are instances of the **C_session** class which is of `T_session` type. `T_session` is a direct subtype of `T_Object` type. Among others, it has the following behaviors: *B_openSession*, *B_closeSession*, *B_saveSession*, *B_quitObjectbase* (a more complete description of session objects and their behaviors is given in Chapter 4) which correspond to the TCL session specific statements.

Every TIGUKAT objectbase has at least one instance of the **C_session** class which is referred to as a *root session*. When a TIGUKAT objectbase is opened, a *root session* becomes the current session in the system. All other sessions can be opened and manipulated from this session by issuing TCL session specific operations. TCL consists of the following session specific operations: **open session, close session, save session, make persistent**, and **quit**.

The *open session* statement is used to open a session object which provides a workspace from which a user can perform operations on the objectbase. The syntax of this statement is:

$$< open\ session > ::= \textbf{open}\ \ < session\ reference >$$

The session reference is a reference to a session object in the objectbase. If a session object referenced by the *session reference* does not exists in the objectbase, a new session object is created, and it becomes the current session in the system. Otherwise, the object which is referenced by the *session reference* becomes the current session in the system.

The *save session* statement is used to save the session environment, and also the session object becomes persistent. The general syntax of this statement is:

$$< save\ session > ::= \textbf{save}\ [< session\ reference >]$$

All transient objects are saved, meaning that their references are stored in the session symbol table (they do not become persistent, however). Next time that session object is opened, the environment is restored, and the user can continue the previously closed session. Otherwise, if the session is closed without saving, all transient objects are lost.

The *close session* statement is used to close a current session without leaving an objectbase. The syntax of this statement is:

$$< close\ session > ::= \textbf{close}\ \ [< session\ reference >]$$

If the session environment has not been saved, all transient object are lost. If the session object has not been saved nor has been made persistent before this statement was issued, it is lost as well. If,

on the other hand, the session environment has been saved, next time this session object is opened, the entire environment is restored.

The *make persistent* statement is used to make transient objects persistent in the objectbase. The syntax of this statement is:

$$< \textit{make persistent} > ::= \textbf{persistent} < \textit{object reference list} > \\ \mid \textbf{persistent all} < \textit{collection reference} >$$

The first statement makes all objects specified in the object references list persistent in the objectbase. Persistence in TIGUKAT is associated with individual objects; therefore, if the referenced object is a collection or a class, only the container object is made persistent. All transient objects which are in this container stay temporary unless they are explicitly made persistent. To make all objects persistent within the container object, the second form of a statement must be used. If the elements of the collection are themselves collections, it recursively makes all objects persistent.

The last session specific statement in TCL is the *quit statement* which is used to quit the session without saving, and leave the TIGUKAT objectbase. The syntax of this statement is:

$$< \textit{quit objectbase} > ::= \textbf{quit}.$$

This statement can be invoked from any session. That means it can be invoked from the root session as well as from any other session. The request to close all sessions which are currently opened is sent. The objects which haven't been made persistent or saved in any opened session are lost.

The following example illustrates a sequence of the invocations of TCL statements in a typical TIGUKAT session.

Example 3.14 A user is in the UNIX environment. To invoke a TIGUKAT objectbase, he types *tigukat* and presses the *RETURN* key:

> tigukat

This statement opens the TIGUKAT objectbase. The TIGUKAT language translators are invoked. The system is ready to accept user requests (a new prompt - % is displayed). A root session object, which provides a workspace for user requests, is opened. A user can open new sessions (workspaces) from the root session by issuing the following statement:

% open *newSession*1

This statement opens a session object referenced by *newSession1*. If there is already a session object referenced by *newSession*1 in the objectbase, the *B_openSession* behavior is applied to it. As a result, it becomes the current session in the system. If, on the other hand, there is no session object referenced by *newSession*1, the *B_new* behavior is applied to the **C_session** object, a new object is created (this object is referenced by *newSession*1), and it becomes the current session in the system.

Thus, a user performs all modifications to the objectbase by issuing TDL and TQL statements in the $newSession1$ session (workspace). If the user wants to save the current session with the entire environment (transient objects which have been created since the opening of the session), the TCL save statement must be invoked:

% save $newSession1$

The behavior *B_saveSession* is applied to *newSession1* object. All transient objects are saved as the session environment, meaning that their references are stored in the session symbol table. Next time this session object is opened, the entire environment is restored, and the user can continue a previously closed session. The next step in the session sequence is to make the current session persistent in the objectbase (if it has been just created). This is done by the TCL *make persistent* statement:

% persistent $newSession1$

The *newSession1* object becomes persistent in the TIGUKAT objectbase. Next time the objectbase is opened, it can become a current session by simply invoking an open session statement with the reference $newSession1$. Finally, to close the current session *newSession1*, the TCL close statement must be invoked:

% close $newSession1$

The *B_closeSession* is applied to the object $newSession1$, and the root session becomes the current session in the objectbase. $\diamond$

In addition, TCL supports an assignment statement. Since TIGUKAT is a reference based model, objects are accessed through their references. To bind a reference to an object that is returned as the result of some query or execution of a behavior, the assignment statement can be used. It has the following structure:

$$< assignment > ::= \textbf{let} < left\ side > \textbf{be} < right\ side >$$

where the left side is:

$$< left\ side > ::= < object\ reference >$$

and the right side can be one of two things:

$$\begin{aligned} < right\ side > ::= & < TQL\ Statement > \\ & \mid < path\ expression > \end{aligned}$$

It should be noted here, that the current implementation of TCL is only preliminary. More statements will be added in the future, and they will be presented in forthcoming papers.

Chapter 4

Integration with TIGUKAT Object Model

One of the underlying characteristics of the TIGUKAT object model is its uniform object semantics. Everything in the model is treated as a *first-class object*. This property makes it easy to extend the model with features that support concepts that are unique to various applications. These features are added to the system through the creation of new types.

When the TIGUKAT primitive type system is augmented by additional behaviors defined on primitive types, it is referred to as a *core type system*. It can then be extended further by the addition of new types to support database functionality (transaction management, views management, query features, etc.). The core type system with the database extensions provides a sufficient base to build advanced OODBMs. In this chapter, two of the database functionality extensions: `T_session` and `T_query` are described. They both facilitate the integration of the TIGUKAT language with the object model. The `T_session` type provides the semantics to represent objectbase sessions as objects. The instances of `T_query` type, on the other hand, represent queries. In addition, the process of opening, accessing and querying the objectbase is outlined.

4.1 TIGUKAT Extensions

Two extensions which are included in the TIGUKAT extended type system: `T_session` and `T_query` facilitate the integration of the TIGUKAT language and TIGUKAT object model. In Section 4.1.1 the `T_session` type is described. The description includes the list of native behaviors defined on that type. A complete description of this type is given in Appendix B. In Section 4.1.2 the description of the `T_query` type is given in a similar way. Section 4.1.3 outlines the advantages of modeling sessions and queries as objects in the TIGUKAT objectbase.

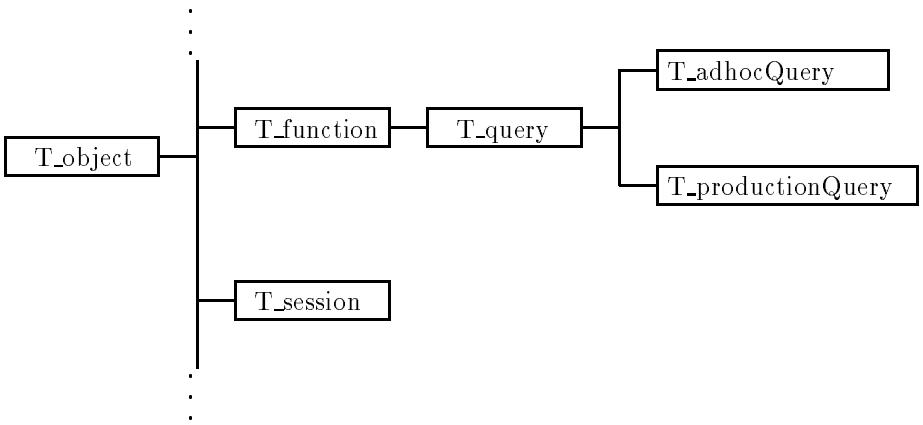

Figure 4.3: Type extensions to the primitive type system.

4.1.1 TIGUKAT Session Objects

Before issuing requests to an objectbase, access to the objectbase must be established. It is done through the session objects which are instances of `T_session` type. There can be one or more instances of that type in a TIGUKAT Objectbase System (TOBS). The main role of a session object is to provide a workspace from which a user can issue requests to the objectbase. Thus, each session object represents a single workspace, and the entire objectbase is accessible from that workspace. All objects created in a session are transient in that session, therefore they are not visible in any other session (there can be more then one session opened at the same time). However, as soon as the transient objects are made persistent, they become visible in all other sessions.

The `T_session` type is a subtype of the `T_object` primitive type as illustrated in Figure 4.3. Each object of this type represents a TIGUKAT objectbase session. The following behaviors provide the control over connection to the objectbase.

- *B_openSession* opens a new workspace and establishes a connection with the objectbase through the receiver session object. The receiver of the message becomes the current session in the system until the request to close it is issued, or a request to open another session is posted.

- *B_saveSession* saves the session environment as well as the receiver object becomes persistent in the objectbase. All transient objects are saved by storing their references in the session symbol table (they do not become persistent however). Next time that session object is opened, the environment is restored, so the user can continue a previously closed session. Otherwise, if the session is closed without saving, all transient objects are lost.

- *B_environment* returns the session symbol table that contains the transient objects (session environment). This is a private behavior which is never invoked by the user.

- *B_closeSession* closes the session (workspace). If the session environment has not been saved,

all transient object are lost. If the session object has not been made persistent before this statement was issued, it is lost as well. If, on the other hand, the session environment has been saved, next time this session object is opened, the entire environment is restored.

- *B_quitObjectbase* exists the TIGUKAT objectbase. The request to close all sessions which are currently opened is sent. The objects which haven't been made persistent or saved in any of the opened sessions are lost.

The behaviors listed above are necessary in the `T_session` type in order to provide the control over the connection to an objectbase in a single-user system. A complete description of the `T_session` type which includes the specification of full signatures and semantics of the behaviors defined in this type is given in Appendix B.

4.1.2 TIGUKAT Query Objects

A user query is in the form of a text string that contains a TQL statement (select, union, minus, intersect, etc.). It is evaluated by invoking a TQL compiler which parses the query, and if it is syntactically and semantically correct, it creates a new instance of `T_query` type. The TQL compiler is invoked by the current session object.

A `T_query` type is a subtype of the `T_function` primitive type as illustrated in Figure 4.3. This means that queries have the status of *first-class objects* and that they inherit all the behaviors and semantics of objects. More specifically, a query is a specialized function that can be asked for its source code, can be compiled and can be executed. In addition a query stores execution statistics, is optimizable, and can be materialized meaning that the result of the execution of the query can be accessed (a collection).

Since, the `T_query` type is a direct subtype of the `T_function` primitive type, it inherits all native behaviors defined on this type. However, some behaviors are redefined in `T_query` to reflect the semantics of queries in the objectbase:

- *B_source* returns the source code of a query in the form of a TQL statement. This behavior is implemented by a stored function.
- *B_compile* compiles the source code of a query. The query statement is translated into an algebraic tree, optimized, and an execution plan is generated.
- *B_executable* returns the execution plan generated by *B_compile*.
- *B_execute* executes the complied code. In general queries, this means submitting the execution plan to the storage manager for processing.
- *B_inputTypes* returns a list of types and the ordering of the query arguments. The types are either of type `T_collection` of a subtype of that type (`T_class`, `T_bag`, etc.).

- *B_outputType* returns the type of the result collection. It is either the `T_collection` type, or one of its subtypes.

Furthermore, queries have the following specialized (native) behaviors.

- *B_initialOAPT* returns an initial Object Algebra Processing Tree (OAPT) resulting from the calculus to algebra translation.
- *B_optimizedOAPT* returns an optimized Object Algebra Processing Tree resulting from the optimization process.
- *B_searchStrat* returns the search strategy which is used by the optimizer to control the optimization process.
- *B_transformations* returns a list of the transformation rule objects that were used when the query was optimized.
- *B_argMbrTypes* returns a list of member types of the target collections.
- *B_resultMbrType* returns the member type of the result collection.
- *B_optimize* starts the execution of the query optimizer on the receiver object. It uses the search strategy stored in that object.
- *B_genExecPlan* generates the "best" Execution Plan from the optimized OAPT. This behavior is invoked by *B_compile*.
- *B_budgetOpt* returns the budget for the optimization.
- *B_lastOpt* returns the date of the last query optimization.
- *B_lastExec* returns the date of the last query execution.
- *B_materialization* returns a reference to the materialized query result (i.e., the actual result collection itself).

The full specification of the `T_query` type is given in Appendix B.

4.1.3 Sessions and Queries as Objects

Incorporating sessions and queries as specialized objects is a very natural and uniform way of extending the object model to include control capabilities as well as declarative capabilities. The major benefits of this approach are:

1. Sessions and queries are first-class objects, so they are represented by the uniform semantics of objects.

2. Since they are objects, they can be queried and operated on by other behaviors. This is especially useful in case of queries, when the generation of statistics about performance is required to define optimization techniques.

3. Since queries are specialized functions, they are uniformly integrated with the operational semantics of the model so that queries can be used as implementations of behaviors (i.e. the result of applying a behavior to an object can trigger the execution of a query).

4. Both `T_session` and `T_query` types can be further specialized by subtyping, thus new concepts can be incorporated (multiuser system features, adhoc versus production queries etc.).

In the current implementation of the TIGUKAT system, there are two subtypes of the `T_query` type: `T_adhocQuery` and `T_productionQuery`, each having its own characteristics. The *ad hoc* queries are interpreted without incurring high compile-time optimization strategies since they are used on an ad hoc basis. The *production* queries, on the other hand, are usually compiled once and then executed many times. Thus, more time is usually spent on optimizing them and more sophisticated techniques are used. In the future, when TIGUKAT Objectbase System becomes a multiuser system, the `T_session` type can be further specialized to subsume additional features required by such systems.

4.2 TIGUKAT Objectbase Access

When a TIGUKAT Objectbase System is opened for the first time by the user, the TIGUKAT extended type system is built. It has one instance of the `T_session` type referred to as the *root* session object. The root session is automatically opened, and it becomes the current session in the objectbase. Every session object has its own symbol table which keeps the information about the session environment. In other words, references to transient objects which have been created and saved in that session are stored in this symbol table.

The objectbase can be either directly modified, accessed and queried from a root session, or new sessions can be opened from a root session (as well as from any other current session) and the connection to the objectbase can be established through them (see Example 3.14).

The TIGUKAT Language provides an interface to the TIGUKAT Objectbase System. It is invoked by the current session, so all user requests are processed on line. All session specific operations like opening a new session, closing a current session, quitting, as well as displaying the objects, making them persistent, etc. are processed by the TCL interpreter. In a similar way, all the object definition statements (creating new types, classes, collections etc.) are interpreted on line by invoking the TDL interpreter. Therefore, when a TDL statements is entered, it is parsed, and if the statement is correct, a new object is created and is accessible at once. All query statement are parsed, complied and executed by invoking the TQL compiler. The TQL compiler parses a

statement, generates an execution plan, and sends it to the object manager for execution. A new instance of `T_query` type is created and the information about a query is stored there.

Chapter 5

Implementation

As a part of the TIGUKAT project, the TIGUKAT language has been implemented and integrated with an existing implementation of the TIGUKAT object model. This chapter describes the language implementation details as well as design decisions that were made during the implementation of the final version of the TIGUKAT language. In Section 5.1 the main design choices are discussed. The arguments for implementing the language parser by hand instead of using available generators (LEX, YACC, BISON) are stated. In Section 5.2 the architecture of TDL interpreter is explained, and the integration with the TIGUKAT object model is presented. The process of TQL compilation is presented in Section 5.3. The translation from calculus to algebra is described and explained. Finally, Section 5.4 contains a short description of the TCL interpreter.

5.1 Design Decisions

Every computer language[1] is defined by a set of rules (grammar) which describes syntactic structures of valid language sentences (programs). A compiler reads a program written in one language called a *source* language, and translates it into an equivalent program written in another language called a *target* language. An interpreter, on the other hand, reads a program, and instead of producing a target program, it performs the operations on line. Both compilers and interpreters are referred to as translators. The TIGUKAT language uses both kinds of translators. TDL and TCL are supported by corresponding interpreters, while TQL has its own compiler.

The first phase of any translator is the syntax analysis of a program written in some source language. This phase, referred to as parsing, checks the syntactic correctness of the program. In practice, however, there is a number of other tasks which are conducted during parsing such as collecting information about various tokens into the symbol table, performing type checking, and other kinds of semantic analysis. In this chapter, the parsing phase is referred to as syntax analysis

[1] The term *computer language* is used broadly to include any language which provides an interface between a user and the machine. That includes all programming languages, query languages etc.

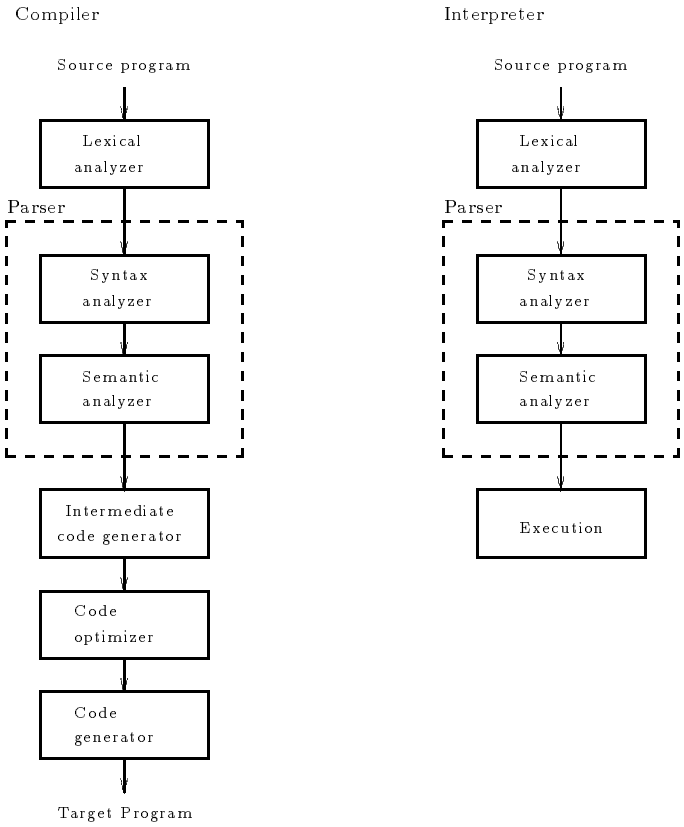

Figure 5.4: Compiler and Interpreter Architectures.

with some semantic checking (Figure 5.4). It is implemented by a language parser that obtains a string of tokens from a lexical analyzer and verifies that the string can be generated from the language grammar.

Three kinds of parsing methods can be distinguished:

- Universal parsing methods which can parse any context-free grammar (Cocke-Younger-Kasami algorithm, Earley's algorithm [ASU86],). However, they are very inefficient which makes them rather useless in practice.

- Top-down methods which attempt to find a leftmost derivation for an input string. They can be easily implemented by hand, however they cannot handle every context-free grammar. This method is used in the implementation of the TIGUKAT language translators.

- Bottom-up methods which attempt to find a rightmost derivation of the input string. One of the advantages of this method is that almost all commonly used context-free languages constructs can be recognized by the bottom up parsers. In addition, there are several tools available to generate bottom up parsers automatically.

The TIGUKAT language is implemented in the C++ language, and its implementation consists of three separate modules, each corresponding to one part of the TIGUKAT language. Recall that TIGUKAT language is divided into three parts: TDL, TQL, TCL, each supporting a different set of statements. The TDL module processes the object definition statements. It is implemented as an interpreter which reads the statement typed by a user, and if it is syntactically and semantically correct, it creates a new object (type, class, behavior, etc.). The TQL module processes query statements. It is implemented as a compiler, which reads the query from the standard input, checks the syntactic and semantic correctness and generates an execution plan. The TCL module processes session specific statements on line. Thus, it is implemented as an interpreter. The top-down parsing method described in Section 5.1.1 is used in the parsing phase of each module.

In subsequent sections, the factors which contributed to the choice of particular methods for the language implementation are discussed. Thus, the explanation for not using available parser generators is given in Section 5.1.1. Section 5.1.2 describes the structure of the symbol table used by the language translators. In Section 5.1.3 the benefits of using the C++ programming language are summarized.

5.1.1 Top-Down Parser

The TIGUKAT language parsing phase of each module is implemented by a recursive-descent parser without backtracking (predictive parser). The language grammar has been rewritten in order to make it suitable for this kind of top-down parser. As the first step, the left recursion has been eliminated from the grammar rules. Second, the grammar has been left factored. A complete description

of the transformations performed on the grammar is given in the technical documentation of the program. The choice of the predictive parsing was motivated by the fact that the TIGUKAT language grammar is simple enough to implement by hand. All non-terminal symbols in the grammar become procedure calls, and all terminal symbols are matched against the input stream. In other words, the parser attempts to match terminal symbols with the input stream, and makes potentially recursive procedure calls.

The error recovery employed by the TIGUKAT parser uses a *panic mode* strategy to restore the parser to a state where processing of the input stream can be continued. It is based on the the fact that statements of the language are separated by a semicolon. On the discovering an error, the parser discards input symbols, one at a time, until a semicolon is found. While the panic mode strategy often skips a considerable number of input symbols without checking for additional errors, it has the advantage of simplicity and it is guaranteed not to go into infinite loop.

5.1.2 Symbol Table

A symbol table is a data structure which supports dictionary access. It is used to keep track of scope and binding information of names (references). Usually, a symbol table satisfies the following requirements. First, a symbol table mechanism must allow the efficient addition of new entries and locating of existing entries. Second, it must be easy to maintain, since it is one of the most complex and frequently used structures in the compiler. Finally, it must be accessible in many different ways and support a variety of functions. For efficiency reasons, most symbol tables are implemented as hash tables.

A symbol table in the TIGUKAT system supports dictionary access to objects in the objectbase. It is implemented as a hash table such that every entry is a pointer to a linked list of symbol table records. Each record has the structure depicted in Figure 5.5. Thus, the *reference* corresponds to an identifier (character string) of an object, the *type* is a pointer to a type object in TOBS, and the *object* is a pointer to a "real" object in TOBS. A symbol table in TOBS can be considered as a handle to the objectbase through which the access to the objectbase is done. Each session object has its own local symbol table which keeps the information about the objects which are explicitly accessible in this session. There is also one global symbol table in TOBS which keeps the information about all the object references in the system. Throughout this chapter, all references which are kept in the symbol table are referred to as *explicit references* to objects in TOBS. There can also be *implicit references* to objects. An object is referenced by an implicit reference if it is only accessible through other objects in the objectbase (path expressions). In other words, there is no direct reference to that object in the local symbol table.

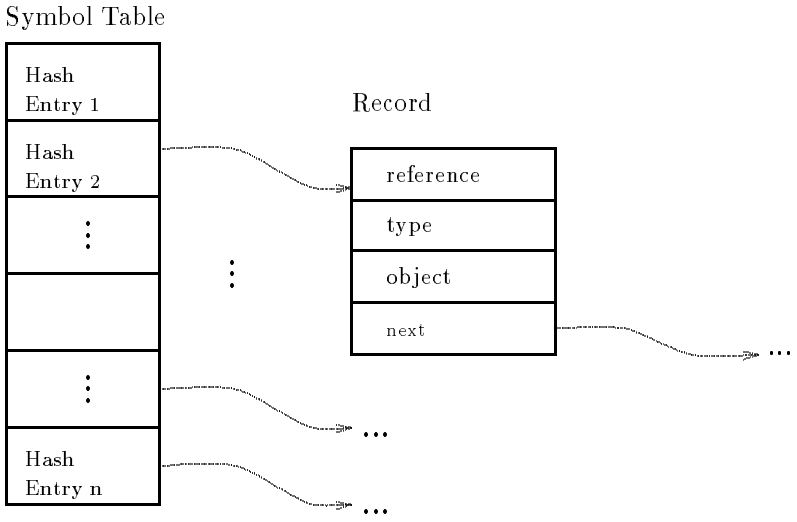

Figure 5.5: Symbol table structure and the corresponding record structure.

5.1.3 C++ Programming Language

The TIGUKAT language has been implemented in the C++ programming language. The following factors contributed to the choice of the C++ language for TIGUKAT implementation. First, C++ is an object-oriented programming language. As a result, it supports the notion of abstract data types, encapsulation, class hierarchies with multiple inheritance and polymorphism. Therefore, the key concepts of the TIGUKAT language could be easily mapped into C++ class structures. Second, being an object-oriented language, it supports good programming practices such as modularity, code reusability, information hiding, generic programming and extensibility. Since the TIGUKAT project involves several people implementing various parts of the system simultaneously, good modularity and extensibility are the key factors in successful integration. Finally, C++ is one of the most efficient object-oriented programming languages.

5.2 TDL interpreter

TDL supports the creation of metaobjects that include type, class, collection, behavior and function objects in a TOBS. Metaobjects are distinct in the system, because they require specialized behaviors in order to be created (the *B_new* behavior must be refined for them). TDL provides the syntax to express those specialized behaviors. TDL statements (create type, create class, etc.) are processed by the TDL interpreter. Each statement is parsed separately, and if it is syntactically and semantically correct, a request to create a new object is sent to the storage manager. A new object is created, and it is accessible throughout the session. If, in addition, the object is made persistent, it stays in the TOBS. The architecture of the TDL interpreter is given in Figure 5.6.

The TDL interpreter accesses a local symbol table which keeps the information about all explicitly

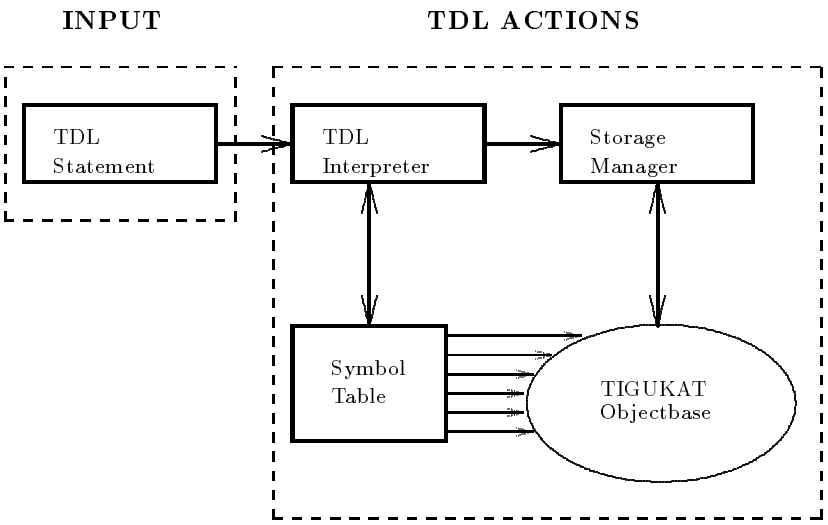

Figure 5.6: The TDL interpreter architecture.

referenced objects. Every time, a new statement is processed, the TDL interpreter must collect all the objects which are referenced in the statement (that include all explicitly and implicitly referenced objects). In case of an explicit reference, the TDL interpreter looks in the local symbol table to find the appropriate object, and if such reference is found, the corresponding object is returned. An implicit references which is given in the form of a path expression must first be type checked. If it is correct, the interpreter sends a request to the object manager [Ira93] to execute the behaviors for a given object (the first reference in the path expression is always an explicit reference to an object, so it can be found in the symbol table). As a result of the execution of a path expression, the object is returned by the object manager. When the TDL interpreter successfully collects all the objects, it sends a request to the object manager to apply the *B_new* behavior to an appropriate object (in case of a *create type* statement it is a **C_type** class object, in case of a *create class* statement, it is a **C_class** class object, etc.) passing collected object as parameters. A new object is created in the objectbase, and a record with the information about this object is added to the symbol table.

Example 5.1 The following TDL statement is a request to create a new type in the TOBS.

create type `T_person`
under `T_Object`
public: *B_getName*: `T_string`,
B_setName(`T_string`): `T_string`,

This statement is syntactically correct, thus the TDL interpreter creates a new type. It does this in two steps. First, it collects all objects which are referenced in the statement (`T_object`, *B_getName*,

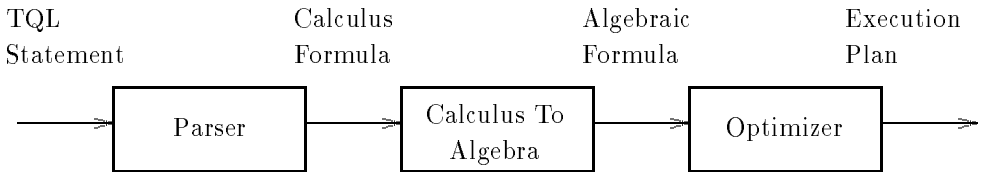

Figure 5.7: The TQL compiler architecture.

B_setName and `T_string`). Two new behaviors are created as a side effect of this statement by sending a request to the object manager to apply the *B_new* behavior to the **C_behavior** class twice with the following parameters respectively: (*getName*, {}, `T_string`), and (*setName*, {`T_string`}, `T_string`). In addition, two entries are created in the symbol table: *B_getName*, and *B_setName*. They are associated with the new behavior objects. After the behaviors are created, all objects in this statement are accessible through the explicit references (`T_object`, *B_getName*, *B_setName*). Thus in the second step, the TDL interpreter requests from the object manager to apply the *B_newtype* behavior to the **C_type** class object with the following parameters: ({`T_object`}, {*B_getName*, *B_setName*}). A new type object is created in the objectbase. Also a new entry is added to the symbol table with reference `T_person` and a new type object (returned by the object manager) is associated with it. ◇

5.3 TQL compiler

The TQL provides the interface which supports the retrieval and the manipulation of objects in an objectbase. The current implementation of the TQL consists of four basic statements: select, union, minus and intersect. The TQL compiler, which is illustrated in Figure 5.7 processes every TQL statement in three steps.

In the first step, the query statement is parsed by the TQL parser, and if it is syntactically correct, the object calculus expression in the form of a calculus tree is returned. The second step translates a calculus expression into an equivalent algebraic expression and returns it in the form of an algebraic tree. The algebraic tree is an input to the query optimizer which performs algebraic transformations on it, and generates an execution plan. However, the optimization and the execution plan generation is not a topic of this thesis, and therefore, it is not discussed any further. A complete specification of the optimizer and execution plan generation can be found in [Mun93]. The calculus formula generated by the parser is returned in the form of a calculus tree which has the following internal representation in C++. There are seven kinds of nodes in this tree. Every inner node represents a logical connective (and, or, negation, exists, forall), while the leaves of the tree represent atomic formulas of the calculus. The semantics of every logical connective and every atom is expressed in the C++ implementation by a separate class. However, all these classes have one common superclass *Formula* as illustrated in the Figure 5.8.

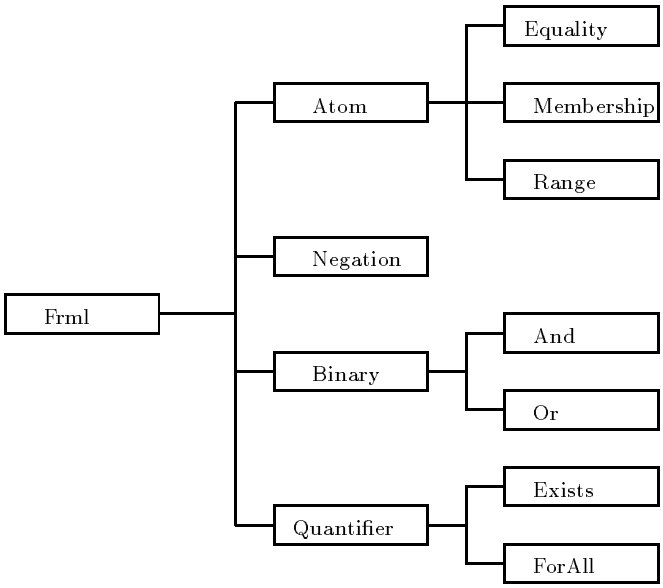

Figure 5.8: The class hierarchy for the internal representation of the calculus formula.

The *Formula* class is an abstract class in this implementation (so are *atom, binary,* and *quantifier* classes) which specifies the following common interface to all its subclasses (which represent various kinds of nodes in the calculus tree):

```
class Formula {
public:
   gen(var, Formula); check gen property G
   con(var, Formula); check con property G
   genAll();   check if the gen holds for all free variables
   conAll();   check if the con holds for all free variables
   evalify();  check if the formula is safe
   genify();   transform from evaluable to allowed form
   ANFify();   transform to an allowed normal form
   algebra();  transform to the algebraic formula
   evaluate(); evaluate and return the value
   pushNot();  push not
   freeVar();  return free variable list
   allVar();   return all variables
}
```

Thus, every formula knows its free variables (*freeVar()*), as well as all variables which appear in it (*allVar()*), it knows how to evaluate itself, how to apply *not* (*pushNot()*), and so on. This representation of a formula in C++ is a classical example of the power of the object-oriented paradigm.

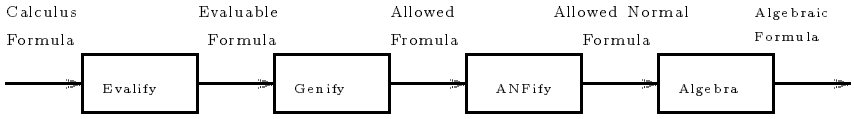

Figure 5.9: Translation algorithm from the calculus to algebra.

Formula specifies a common interface in the form of a list of public methods, while every subclass implements those methods differently, possibly adding some new ones. Thus, although, various formulas react differently when a method is applied, that fact is hidden and a correct answer is obtained (polymorphism).

The general form of a calculus formula, which is returned by a parser, becomes an input to the translation algorithm which implements the second phase of the TQL compiler. In this step, a formula is translated from the calculus to the algebra, is further optimized and an execution plan is generated. However, the optimization and the execution plan generation is not a topic of this thesis, and therefore, it is not discussed any further.

The translation algorithm consists of four steps as illustrated in Figure 5.9. The first step of the algorithm checks if the formula is safe. As shown in [PLÖS93], any calculus formula is safe if the evaluable property (defined below) holds for it. The following definitions and rules specify the required properties of calculus formulas.

Definition 5.1 *Evaluable:* A formula F is *evaluable* or has the *evaluable property* if the following conditions are met:

1. For every variable x that is free in F, $gen(x, F)$ holds.
2. For every subformula $\exists x A$ of F, $con(x, A)$ holds.
3. For every subformula $\forall x A$ of F, $con(x, \neg A)$ holds.

Definition 5.2 *Allowed:* A formula F is *allowed* or has the *allowed property* if the following conditions are met:

1. For every variable x that is free in F, $gen(x, F)$ holds.
2. For every subformula $\exists x A$ of F, $gen(x, A)$ holds.
3. For every subformula $\forall x A$ of F, $gen(x, \neg A)$ holds.

where the rules for *gen* and *con* are given in Figure 5.10. Intuitively, $gen(x, A)$ means that the formula A can *generate* all the needed values of variable x that contribute to making A true and that there are only a finite number of these values. Subsequently, $con(x, A)$ holds if the variable x is *constrained* in A meaning that the following conditions:

- $gen(x, A(x, \vec{y}))$ holds as above, or
- $A(x, \vec{d})$ is true for all bindings of x.

$gen(x, A, A)$	**if**	$edb(A)$ **and** $free(x, A)$
$gen(x, A, A)$	**if**	$gdb(A)$
$gen(x, \neg A, G)$	**if**	$gen(x, pushnot(\neg A), G)$
$gen(x, \exists y A, G)$	**if**	$distinct(x, y)$ **and** $gen(x, A, G)$
$gen(x, \forall y A, G)$	**if**	$distinct(x, y)$ **and** $gen(x, A, G)$
$gen(x, A \vee B, G_1 \vee G_2)$	**if**	$gen(x, A, G_1)$ **and** $gen(x, B, G_2)$
$gen(x, A \wedge B, G)$	**if**	$gen(x, A, G)$
$gen(x, A \wedge B, G)$	**if**	$gen(x, B, G)$
$con(x, A, A)$	**if**	$edb(A)$ **and** $free(x, A)$
$con(x, A, A)$	**if**	$gdb(A)$
$con(x, A, \bot)$	**if**	$notfree(x, A)$
$con(x, \neg A, G)$	**if**	$con(x, pushnot(\neg A), G)$
$con(x, \exists y A, G)$	**if**	$distinct(x, y)$ **and** $con(x, A, G)$
$con(x, \forall y A, G)$	**if**	$distinct(x, y)$ **and** $con(x, A, G)$
$con(x, A \vee B, G_1 \vee G_2)$	**if**	$con(x, A, G_1)$ **and** $con(x, B, G_2)$
$con(x, A \wedge B, G)$	**if**	$gen(x, A, G)$
$con(x, A \wedge B, G)$	**if**	$gen(x, B, G)$
$con(x, A \wedge B, G_1 \vee G_2)$	**if**	$con(x, A, G_1)$ **and** $con(x, B, G_2)$

where:

$edb(A)$ holds if formula A is either a finite range atom, or if formula A is an equality atom of the form $x = c$, where c is a ground term, or if formula A is a membership atom of the form $x \in c$ where c is also a ground term.
$gdb(x, A)$ holds, if variable x can be generated from A.
$free(x, A)$ holds if variable x is bound in A or it does not appear in the formula A at all.
$notfree(x, A)$ holds if variable x is not free in A.
$distinct(x, y)$ holds if x and y are different variables.

Figure 5.10: Extended rules of *gen* and *con* that produce generators.

$$
\begin{array}{rcl}
A(\vec{x}) \vee B(\vec{x}) & \Longrightarrow & (A'_{\vec{x}} \cup B'_{\vec{x}})_{\vec{x}} \\
A(\vec{x}) \wedge B(\vec{x}) & \Longrightarrow & (A'_{\vec{x}} \cap B'_{\vec{x}})_{\vec{x}} \\
A(\vec{x}) \wedge B(\vec{y}) & \Longrightarrow & (A'_{\vec{x}} \times B'_{\vec{y}})_{\vec{x},\vec{y}} \\
A(\vec{x}) \wedge \neg B(\vec{x}) & \Longrightarrow & (A'_{\vec{x}} - B'_{\vec{x}})_{\vec{x}} \\
A(\vec{x},\vec{y}) \wedge \neg B(\vec{y}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} - (A'_{\vec{x},\vec{y}} \ \bowtie_{\vec{y}=\vec{y}} \ B'_{\vec{y}})_{\vec{x},\vec{y}})_{\vec{x},\vec{y}} \\
A(\vec{x},\vec{y}) \wedge F(\vec{x}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \sigma_F \ )_{\vec{x},\vec{y}} \\
A(\vec{x},\vec{y}) \wedge B(\vec{x},\vec{z}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \bowtie_{\vec{x}=\vec{x}} \ B'_{\vec{x},\vec{z}})_{\vec{x},\vec{y},\vec{z}} \\
A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge F(\vec{y},\vec{z}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \bowtie_F \ B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z}} \\
A(\vec{u},\vec{w},\vec{x}) \wedge B(\vec{w},\vec{x},\vec{y}) \wedge F(\vec{u},\vec{w},\vec{y}) & \Longrightarrow & (A'_{\vec{u},\vec{w},\vec{x}} \ \bowtie_{\vec{w}=\vec{w}\wedge\vec{x}=\vec{x}\wedge F} \ B'_{\vec{w},\vec{x},\vec{y}})_{\vec{u},\vec{w},\vec{x},\vec{y}} \\
A(\vec{x},\vec{y}) \wedge o\theta mop(\vec{x}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \gamma^o_{o\theta mop} \ )_{\vec{x},\vec{y},o} \\
A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge o\theta mop(\vec{y},\vec{z}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \gamma^o_{o\theta mop} \ B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z},o} \\
\exists\vec{y}A(\vec{x},\vec{y}) & \Longrightarrow & (A'_{\vec{x},\vec{y}} \ \triangle_{\vec{y}})_{\vec{x}}
\end{array}
$$

Figure 5.11: Transformations from object calculus to object algebra.

In other words, x is constrained in A if it is generated in every disjunct in which it appears.

Thus, the safety of the formula can be determined syntactically, and it is done by the *evalify* algorithm [PLÖS93]. This algorithm recursively checks if the formula has an evaluable property, if so it returns TRUE, otherwise the formula is rejected and the appropriate message is displayed to the user.

In the second step, the evaluable formula is transformed to the allowed form. As shown in [GT91], every evaluable formula can be transformed to the equivalent allowed form. This is an important step of the algorithm as, some further transformations (i.e. distributed law transformation) which are necessary to transform the calculus formula to algebra, do not preserve the evaluability property, but as shown in [GT91] they do preserve the allowed property. Thus, the formula must have an allowed property before any transformations leading to the allowed normal form can be performed. This step is implemented by the *genify* algorithm [PLÖS93].

The next step of the translation algorithm is to normalize an allowed formula by putting it into Allowed Normal Form (ANF) which is done by the *ANFify* algorithm [PLÖS93]. The reason for converting a formula into ANF is to divide it into subformulas which are independent of atoms that appear outside the quantifier for that formula. In other words, in the final translation to the algebra, every subformula can be independently translated to an algebraic formula.

The final step of the translation algorithm is the transformation of an ANF formula into an equivalent series of object algebras. This is done by a simple application of transformation rules shown in the Figure 5.11 which are applied from the inner to the outer formula. The output of this step is an algebraic tree (initial object algebraic processing tree) which becomes the input to the query optimizer [Mun93].

The algebraic tree has the following representation. There are two kinds of nodes in that algebraic tree. Every inner node represents one of the algebraic operators (*union, minus, select, generate*, etc.). It has two children nodes: the target collection, and the argument collections each of which can be

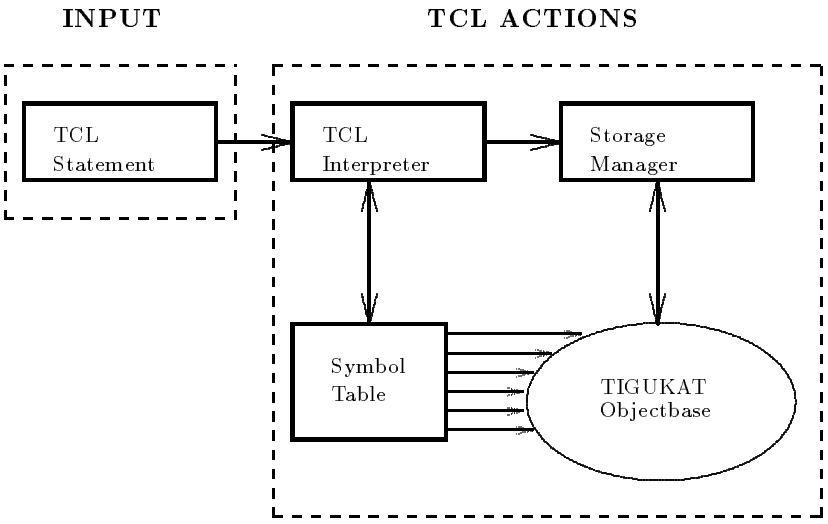

Figure 5.12: The TCL interpreter architecture.

either another algebraic operator (inner node of the tree), or a reference to a collection (a leaf node of the tree). In addition, every inner node has a *constraint* which is either a predicate (in case of the select and join operators), or list of behaviors and their arguments (in case of the map operator), or null (for all other cases). The leaf nodes of the algebraic tree are path expressions whose execution results in references to collections. The full specification and description of the algebraic trees are given in [Mun93].

5.4 TCL interpreter

TCL supports session specific operations. It is implemented by an interpreter whose architecture is given in Figure 5.12. Every TCL statement is parsed by the TCL interpreter, and if it is syntactically and semantically correct, it is executed.

Five statements are currently supported by the TCL interpreter. *Open* and *close session* statements operate directly on session objects. If they are encountered, the request to the object manager is sent by the TCL interpreter, and if a specified object exists it is opened or closed respectively. If it does not exists, then in the case of the *open* statement, it is first created, a new copy of the symbol table is initialized and associated with it, a new session object is returned and it becomes a current session. In case of the *close* statement, the error message is displayed. *Save, make persistent,* and *assign* statements perform operations mainly on the local symbol table, storing the information about new bindings, retrieving objects and making them persistent.

Chapter 6

Conclusion and Future Work

This thesis is part of the ongoing TIGUKAT project to develop a new object management system based on a uniform, behavioral object model. The first extension which is being added to the model is a query model and language. Its specification includes two formal languages: a declarative object calculus and a procedural object algebra; a user-level language; and the equivalence proof among the three of them.

The main goal of this thesis is the design and implementation of the user level language which has the same expressive power as the object calculus, and which conforms to the general object query language frameworks presented over the last years [Kim89, Bla91, BNPS92, Str91, ÖS91]. The TIGUKAT language is a high level user language which provides declarative access to the TIGUKAT objectbase. The design of this language was mainly influenced by SQL which is accepted as a standard query language in relational systems. It is divided into three parts: The TIGUKAT Definition Language, the TIGUKAT Query Language, and the TIGUKAT Control Language. The syntax of the query language is based on the SQL *select-from-where* structure, while the formal semantics are defined in terms of the object calculus. It is shown in this thesis that there is a complete reduction from TQL to the calculus, which makes the semantics of the language well defined and allows to specify the formal methods to check the safety of the user defined queries, and to perform the algebraic transformation on them. In addition, TIGUKAT language accepts path expression in the *select, from* and *where* clauses, thus both forms (implicit and explicit) of joins are supported. Queries operate on collections, and they always return collections as results. The results of queries are queryable, and they can be used as predicates or ranges in other queries (i.e., nested queries). Finally, TQL is orthogonal to all object model extensions. Persistence is defined on the object level, thus, queries can be formulated on transient as well as on persistent objects in a uniform way.

There are some extensions that can be added to the language presented in this thesis in order to increase its functionality and expressive power.

- TQL must be further extended to support the statements which perform updates on the objectbase. That includes the definition of the syntax and the formal semantics of *insert, update* and *delete* statements. Moreover, the syntax for bulk updates should be provided.

- The syntactic support for the application of aggregate functions (similar to those in the relational systems) should be added to TQL. Furthermore, the constructs for grouping of objects and defining the order (GROUP BY, ORDER BY) in the result collections should be also added.

- TCL can be extended to include flow control statements like a loop statement, if statement, case statement and others. That would make the language computationally complete and allow the specification of computed functions within the TIGUKAT language without the need to call external functions written in other languages. Also, TCL could be enhanced by statements supporting easy and fast browsing of the objectbase, editing query files and displaying the schema.

Bibliography

[AB91] S. Abiteboul and A. Bonner. Objects and Views. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 238–247, May 1991.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K.Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. 1st Int'l. Conf. on Deductive and Object-Oriented Databases*, pages 40–57, 1989.

[Abi90] S. Abiteboul. Towards a Deductive Object-Oriented Database Language. *Data & Knowledge Engineering*, 5:263–287, 1990.

[AG92] R. Agrawal and N.H. Gehani. Ode: The language and the data model. Technical report, AT&T Bell Laboratories, 1992.

[Aro89] S. Aronoff. *Geographic Information Systems: A Management Perspective*. WDL Publications, 1989.

[ASL89] A.M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 433–442, 1989.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[Ban92] F. Bancilhon. Understanding Object-Oriented Database Systems. In *Proc. 3rd Int'l Conf. on Extending Database Technology*, pages 1–9, March 1992.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 122–138, June 1989.

[BCG+87] J. Banerjee, H.T. Chou, J.F. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim. Data Model Issues for Object-Oriented Applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Bee90] C. Beeri. A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering*, 5:353–382, 1990.

[BK90] F. Bancilhon and W. Kim. Object-Oriented Database Systems: In Transition. *ACM SIGMOD Record*, 19(4):49–53, 1990.

[Bla91] J.A. Blakeley. DARPA Open Object-Oriented Database Preliminary Module Specification: Object Query Module. Technical report, Texas Instruments, December 1991.

[Bla93] J.A. Blakeley. ZQL[C++]:Extending the C++ Language with an Object Query Capability. Technical report, Texas Instruments, 1993.

[BMO+89] R. Brentl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison Wesley, 1989.

[BNPS92] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Transactions On Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[BTA91] J.A. Blakeley, C.W. Thompson, and A.M. Alashqur. Strawman Reference Model for Object Query Languages. *Computer Standards & Interfaces*, 13:185–199, 1991.

[CDF+88] M. Carey, D.J. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shekita. The Architecture of the EXODUS Extensible DBMS. In M. Stonebraker, editor, *Readings in Database Systems*, pages 488–501. Morgan Kaufmann Publishers, 1988.

[CDV88] M. Carey, D.J. DeWitt, and S.L. Vandenberg. A Data Model and Query Language for EXODUS. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 413–423, June 1988.

[CM84] G. Copeland and D. Maier. Making Smalltalk a Database System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 316–325, June 1984.

[Cod70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod71] E.F. Codd. Relational Completeness of Data Base Sublanguages. In *Courant Computer Science Symposium 6*, pages 65–98, May 1971.

[Com86] X3H2 (American National Standards Database Commettee). Database Language SQL. Technical Report ANSI X3.135-1986, American National Standards Institute, 1986.

[Dat87] C.J. Date. *A Guide To SQL Standard*. Addison-Wesley Publishing Company, 1987.

[Deu90] O. Deux, *et. al.* The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[Deu91] O. Deux, *et. al.* The O_2 System. *Communications of the ACM*, 34(10):34–48, October 1991.

[DGJ92] S. Dar, N.H. Gehani, and H.V. Jagadish. CQL++, A SQL for a C++ Based Object-Oriented DBMS. Technical report, AT&T Bell Laboratories, 1992.

[ES90] M.A. Ellis and B. Stroustroup. *The Annotated C++ Reference*. Addison Wesley, 1990.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[Gal92] L.J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proc. 1st International Conference on Information and Knowledge Management*, pages 17–26, November 1992.

[GR85] A. Goldberg and D. Robson. *SMALLTALK-80: The Language and its Implementation.* Addison-Wesley, 1985.

[GT91] A.V. Gelder and R.W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[Ira93] B. Irani. Implementation, Design and Developmnet of the TIGUKAT Object Model. Master's thesis, University of Alberta, 1993.

[KBC+89] W. Kim, N. Ballou, H.T. Chou, J.F. Garza, and D. Woelk. Features of the ORION Object-Oriented Database System. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison Wesley, 1989.

[KC86] S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. of the Int'l Conf on Object-Oriented Programming: Systems, Languages, and Applications*, pages 406–416, September 1986.

[Ken91] W. Kent. Important Features of Iris OSQL. *Computer Standards & Interfaces*, 13:201–206, 1991.

[KGBW90] W. Kim, J.F. Garza, N. Ballou, and D. Wolek. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[Kim89] W. Kim. A Model of Queries for Object-Oriented Databases. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 424–432, August 1989.

[Kim90] W. Kim. Research Directions in Object-Oriented Databases. In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, April 1990.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Quering Object-Oriented Databases. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 393–402, 1992.

[LLOW91] C. Lamb, G. Landis, J. Orenstien, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[LR89a] C. Lécluse and P. Richard. Modeling Complex Structures in Object-Oriented Databases. In *Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 360–368, March 1989.

[LR89b] C. Lécluse and P. Richard. The O_2 Database Programming Language. In *Proc. 15th Int'l Conf. on Very Large Data Bases*, pages 411–422, August 1989.

[LRV88] C. Lecluse, P. Richard, and F. Velez. O_2, an Object-Oriented Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 424–433, September 1988.

[MS87] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In *Research Directions in Object-Oriented Programming*, pages 355–392. M.I.T. Press, 1987.

[Mun93] A. Munoz. An Extensible Query Optimizer for TIGUKAT Object Management System. Master's thesis, University of Alberta, 1993.

[OHMS92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query Processing in the ObjectStore Database System. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 403–412, June 1992.

[ÖS91] M.T. Özsu and D.D. Straube. Issues in Query Model Design in Object-Oriented Database System. *Computer Standards & Interfaces*, 13:157–167, 1991.

[ÖV90] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice-Hall, 1990.

[PLÖS93] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The Query Model and Query Language of TIGUKAT. Technical Report TR93-01, University of Alberta, January 1993.

[PÖS92] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An Object Model for Query and View Support in Object Database Systems. Technical Report TR92-14, University of Alberta, October 1992.

[RS87] L.A. Rowe and M.R. Stonebraker. The POSTGRES Data Model. In *Proc. 13th Int'l Conf. on Very Large Data Bases*, pages 83–96, September 1987.

[Shi81] D.W. Shipman. The Functional Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SK91] M. Stonebraker and G. Kemnitz. The POSTGRES Next-Generation Database Management System. *Communications of the ACM*, 34(10):78–92, October 1991.

[Sny90] A. Snyder. An Abstract Object Model for Object-Oriented Systems. Technical Report HPL-90-22, Hewlett Packard Labs, April 1990.

[SÖ90] D.D. Straube and M.T. Özsu. Type Consistency of Queries in an Object-Oriented Database System. In *ECOOP/OOPSLA '90 Proceedings*, pages 224–233, October 1990.

[SR86] M. Stonebraker and L.A. Rowe. The Design of POSTGRES. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 340–355, May 1986.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[SRL+90] M. Stonebraker, L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P.Bernstein, and D. Beech. Third-Generation Data Base System Manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[Sto90] M. Stonebraker, et al. Third-Generation Data Base System Manifesto. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, page 396, June 1990.

[Str90] D.D. Straube. An Introduction to Object-Oriented Databases. In 15^{th} *Simposium Internacional de Sistemas Computacionale*, March 1990.

[Str91] D.D. Straube. *Queries and Query Processing in Object-Oriented Database Systems.* PhD thesis, University of Alberta, 1991.

[Tom90] C.D. Tomlin. *Geographic Information Systems and Cartographic Modeling.* Prentice-Hall, 1990.

[Ull82] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982. 2nd. Edition.

[Ull87] J.D. Ullman. Database Theory: Past and Future. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 1–10, March 1987.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988. Volume 1.

[Zan88] C. Zaniolo. The Database Language GEM. In M. Stonebraker, editor, *Readings in Database Systems*, pages 423–434. Morgan Kaufmann Publishers, 1988.

[ZM90] S. Zdonik and D. Maier. Fundamentals of Object-Oriented Databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1–36. Morgan Kaufmann Publishers, 1990.

A

Language Grammar

< *session* >

::= **quit**

| < *statement list* > **quit**

< *statement list* >

::= < *statement* >

| < *statement* > , < *statement list* >

< *statement* >

::= < *tdl statement* >

| < *tql statement* >

| < *tcl statement* >

< *tdl statement* >

::= < *type declaration* >

| < *collection declaration* >

| < *class declaration* >

| < *behavior manipulation* >

| < *function declaration* >

| < *association* >

< *type declaration* >

::= **create type** < *new reference* >

under < *type list* >

< *behavior specification* >

< *collection declaration* >

::= **create collection** < *new reference* >

type < *type reference* >

[**with** < *obejct variable list* >

< *class declaration* >

::= **create class** [< *new reference* >]

on < *type reference* >

< *behavior manipulation* >

::= **add to** < *type reference* > < *behavior specification* >

| **remove from** < *type reference* >

behaviors: < *behavior name list* >

< *function declaration* >

::= < *language* > **function** < *function signature* >

begin

< *function code* >

end

| **external function** < *function signature* >

< *association* >

::= **associate in** < *type reference* >

[< *computed list* >]

[< *stored list* >]

< *computed list* >

::= < *comp elem* >

| < *computed list* > , < *comp elem* >

< *comp_get list* >

::= < *comp_get elem* >

| < *comp_get list* > , < *comp_get elem* >

< *comp_get_set list* >

::= < *comp_get_set elem* >

| $< comp\_get\_set\ list >$, $< comp\_get\_set\ elem >$

$< stored\ list >$

::= $< get\ elem >$ [$< copm\_get\ list >$] $< set\ elem >$
[$< copm\_get\_set\ list >$]

| $< get\ elem >$ [$< copm\_set\ list >$] $< get\ elem >$
[$< copm\_get\_set\ list >$]

$< comp\_get\ elem >$

::= $< comp\ elem >$

| $< get\ elem >$

$< comp\_set\ elem >$

::= $< comp\ elem >$

| $< set\ elem >$

$< comp\_get\_set\ elem >$

::= $< comp\ elem >$

| $< get\ elem >$

| $< set\ elem >$

$< comp\ elem >$

::= $< behavior\ reference\ list >$ **with** $< function\ reference >$

$< get\ elem >$

::= $< behavior\ reference\ list >$ **with** GET

$< set\ elem >$

::= $< behavior\ reference\ list >$ **with** SET

$< association\ reference >$

::= $< funtion\ reference >$

| GET

| SET

$< function\ code >$

::= $< TQL\ Statement >$

| **C++_String**

< *language* >
::= **TQL**
| **C++**

< *new reference* >
::= **identifier**

< *type reference* >
::= < *term* >

< *class reference* >
::= < *term* >

< *function reference* >
::= < *term* >

< *behavior reference* >
::= < *term* >

< *collection reference* >
::= < *term* >
| < *subquery* >

< *behavior name* >
::= **identifier**

< *function name* >
::= **identifier**

< *type list* >
::= < *type reference* >
| < *type list* > , < *type reference* >

< *behavior name list* >
::= < *behavior name* >

| < *behavior name list* > , < *behavior name* >

< *behavior specification* >
::= < *public behaviors* > < *private behaviors* >

< *public behaviors* >
::= /* empty */
| **public** < *signature list* >

< *private behaviors* >
::= /* empty */
| **private** < *signature list* >

< *signature list* >
::= < *behavior signature* >
| < *signature list* > , < *behavior signature* >

< *behavior signature* >
::= < *behavior name* > [(< *type list* >)]
: < *type reference* >

< *function signature* >
::= < *function name* > [(< *formal parameter list* >)]
: < *type reference* >

< *formal parameter list* >
::= < *formal parameter list* >

< *formal parameter list* >
::= < *formal parameter* >
| < *formal parameter list* > , < *formal parameter* >

< *formal parameter* >
::= **identifier** : < *type reference* >

< *TQL Statements* >
::= < *select statement* >

| < *union statement* >
| < *minus statement* >
| < *intersect statement* >

< *select statement* >
::= **select** < *object variable list* >
[**into** [**persistent** [**all**]] < *collection reference* >]
from < *range variable list* >
[**where** < *boolean formula* >]

< *union statement* >
::= < *collection reference* > **union** < *collection reference* >

< *minus statement* >
::= < *collection reference* > **minus** < *collection reference* >

< *intersect statement* >
::= < *collection reference* > **intersect** < *collection reference* >

< *object variable list* >
::= < *object variable* >
| < *object list* > , < *object variable* >

< *object variable* >
::= [(< *cast type* >)] < *term* >
| < *index variable* >

< *term* >
::= < *variable reference* >
| < *constant reference* >
| < *path expression* >

< *index variable* >
::= **identifier** [< *behavior name list* >]

< *variable reference* >
::= **identifier**

< *constant reference* >

::= **#identifier**

< *path expression* >

::= < *term* > . < *function expression* >

< *function expr* >

::= < *behavior name* > ( )

| < *behavior name* > (< *term list* >)

< *term list* >

::= < *term* >

| < *term list* > , < *term* >

< *variable list* >

::= < *variable* >

| < *variable list* > , < *variable* >

< *range variable list* >

::= < *range variable* >

| < *range variable list* > , < *range variable* >

< *range variable* >

::= < *variable list* > **in** < *collection reference* > [+]

< *boolean formula* >

::= < *atom* >

| **not** < *boolean formula* >

| < *boolean formula* > **and** < *boolean formula* >

| < *boolean formula* > **or** < *boolean formula* >

| (< *boolean formula* >)

| < *exists predicate* >

| < *forAll predicate* >

| < *boolean function expression* >

< *atom* >

::= < *term* > **=** < *term* >

| < *term list* > **in** < *collection reference* > [+]

< *exists predicate* >

::= **exists** < *collection reference* >

< *forAll predicate* >

::= **forAll** < *range variable list* > < *boolean formula* >

< *subquery* >

::= (< *query specification* >)

< *tcl statement* >

::= < *open session* >

| < *save session* >

| < *close session* >

| < *make persistent* >

| < *quit objectbase* >

| < *assignment* >

< *open session* >

::= **open** < *session reference* >

< *session reference* >

::= < *term* >

< *save session* >

::= **save** [< *session reference* >]

< *close session* >

::= **close** [< *session reference* >]

< *make persistent* >

::= **persistent** < *object reference* >

| **persistent all** < *collection reference* >

< *quit objectbase* >

::= **quit**

< *assignment* >

::= **let** < *right side* > **be** < *right side* >

< *left side* >

::= < *object refernce* >

< *right side* >

::= < *TQL Statement* >

| < *term* >

B

Type Specifications

T_session

Supertypes: `T_object`

Subtypes:

Native Behaviors:

openSession *B_openSession* : `T_session`

Example: *B_openSession*(*o*)

Symbol:

It opens a session and establishes a connection with an objectbase. The receiver of the message is opened, and it becomes the current session in an objectbase.

saveSession *B_saveSession* : `T_session`

Example: *B_saveSession*(*o*)

Symbol:

It saves the session environment. All transient objects are saved, and their references are stored in the session symbol table. Next time that session object is opened, the environment is restored, and the user can continue a previously closed session. Otherwise, all transient objects are lost.

closeSession *B_closeSession* : `T_session`

Example: *B_closeSession*(*o*)

Symbol:

It closes the session (workspace). If the session environment has not been saved, all transient object are lost. If the session object has not been made persistent before this statement was issued, it is lost as well. If, on the other hand, the session environment has been saved, next time this session is opened, the entire environment is restored.

quitObjectbase *B_quitObjectbase* : `T_null`

Example: *B_quitObjectbase*(*o*)

Symbol:

It exists the whole TIGUKAT objectbase. The request to closed all sessions which are currently open is sent. The objects which haven't been made persistent or saved in any opened session are lost.

environment *B_environment* : `T_object`

Example: *B_environment*(*o*)

Symbol:

It returns a session symbol table in which the transient objects are stored (session environment). This is a private behavior.

T_query

Supertypes: `T_function`

Subtypes: `T_adhocQuery`, `T_productionQuery`

Refined Behaviors:

source *B_source* : `T_string`

Example: *B_source*(*o*)

Symbol:

It returns the source code for a query *o* which is a TIGUKAT Query Language (TQL) statement.

executable *B_executable* : `T_object`

Example: *B_executable*(*o*)

Symbol:

It returns the executable code which is in form of the Execution Plan of the optimized query object *o*.

execute *B_execute* : `T_list`⟨`T_object`⟩ → `T_object`

Example: *B_execute*(*p*)(*o*)

Symbol:

It submits a list *p* of the execution plans of the query object *o* to the Object Manager for processing.

compile *B_compile* : `T_object`

Example: *B_compile*(*o*)

Symbol:

It compiles the source code for a query. The compilation process involves the following steps: translating the query statement *o* written in TQL language into an equivalent calculus expression; translating the calculus expression into an equivalent algebra expression and checking it for type consistency; optimizing by applying equivalence preserving rewrite rules to the algebra expression; and generating an Execution Plan by replacing each individual algebra operator from the transformed object algebra query with a "best" subtree of object manager calls.

argTypes *B_argTypes* : `T_list`⟨`T_object`⟩

Example: *B_argTypes*(*o*)

Symbol:

	It returns a list of types and the ordering of the argument objects of the query o. The type of each element is either `T_collection`, or any subtype of that type(`T_class`,`T_bag`, etc.).
resultType	*B_resultType* : `T_type`
	Example: *B_resultType*(o)
	Symbol:
	It returns the result type of the query execution. That type is either the `T_collection` type, or any subtype of this type (`T_class`, `T_bag`, etc.)

Native Behaviors:

initialOAPT	*B_initialOAPT* : `T_algOp`
	Example: *B_initialOAPT*(o)
	Symbol:
	It returns the initial Object Algebra Processing Tree (OAPT) resulting from the calculus to algebra translation. This initial OAPT(s) constitutes the initial state(s) of the search space used for the algebraic optimization of the query object o.
optimizedOAPT	*B_optimizedOAPT* : `T_set`⟨`T_algOp`⟩
	Example: *B_optimizedOAPT*(o)
	Symbol:
	It accesses the optimized OAPT (or set of optimized OAPTs) resulting from the optimization process for the query object o.
searchStrat	*B_searchStrat* : `T_searchStrat`
	Example: *B_searchStrat*(o)
	Symbol:
	It accesses the search strategy which is used by the optimizer to control the optimization of the query object o.
transformations	*B_transformations* : `T_list`⟨`T_algEqRule`⟩
	Example: *B_transformations*(o)
	Symbol:
	It accesses the list of transformation rule objects used for the algebraic optimization of the query object o.
argMbrTypes	*B_argMbrTypes* : `T_list`⟨`T_type`⟩
	Example: *B_argMbrTypes*(o)
	Symbol:

	It returns a list of types corresponding to the member types of target collections of the query object o.
resultMbrType	*B_resultMbrType* : `T_type`
	Example: *B_resultMbrType*(o)
	Symbol:
	It returns the membership type of the resulting collection from executing the query object o.
optimize	*B_optimize* : `T_set`⟨`T_algOp`⟩
	Example: *B_optimize*(o)
	Symbol:
	It starts the execution of the algebraic query optimizer over the query object o, using its search strategy, and taking its initial OAPT.
genExecPlan	*B_genExecPlan* : `T_algOp` → `T_omOp`
	Example: *B_genExecPlan*(o)(p)
	Symbol:
	It generates the ""best" Execution Plan from the optimized OAPT object p for the query object o.
budgetOpt	*B_budgetOpt* : `T_integer`
	Example: *B_budgetOpt*(o)
	Symbol:
	It accesses the optimization budget for the optimization process to the query object o. It is an upper bound for the optimization cost which is used by the search strategy to control the optimization.
lastOpt	*B_lastOpt* : `T_date`
	Example: *B_lastOpt*(o)
	Symbol:
	It accesses the last date in that the query object o was optimized.
lastExec	*B_lastExec* : `T_date`
	Example: *B_lastExec*(o)
	Symbol:
	It accesses the last date in that the query object o was executed. It can be useful for consistency checks between changes in statistics of the query and the last optimization.
materialization	*B_materialization* : `T_object`
	Example: *B_materialization*(o)
	Symbol:

It is a reference to the materialized query result (i.e., the actual result collection itself).