University of Alberta

The Query Model and Query Language of TIGUKAT

by

Randal J. Peters
Anna Lipka
M. Tamer Özsu
Duane Szafron

Technical Report TR 93–01
January 1993

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

The Query Model and Query Language of TIGUKAT

Randal J. Peters
Anna Lipka
M. Tamer Özsu
Duane Szafron

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
{randal,anna,ozsu,duane}@cs.ualberta.ca

Technical Report TR 93–01
January 1993

Abstract

The establishment of a formal object model provides a theoretical foundation to investigate other objectbase features such as query processing. In this report, we present an extensible uniform behavioral object query model for the TIGUKAT object management system. The TIGUKAT object model [PÖS92] is purely *behavioral* in nature, supports full encapsulation of objects, defines a clear separation between primitive components such as *types, classes, collections, behaviors* and *functions*, and incorporates a *uniform* semantics over objects which makes it a favorable basis for an extensible query model. Every concept that can be modeled in TIGUKAT has the uniform semantics of a *first class object* with well-defined behavior. Following this semantics, queries are modeled as type and behavioral extensions to the base object model, thus incorporating queries as an extensible part of the model itself.

The complete query model definition presented in this report includes: the type and behavior extensions to the base model; a formal object calculus with a logical foundation that introduces a function symbol to incorporate the behavioral paradigm of the object model into the calculus; a behavioral/functional object algebra with a comprehensive set of object-preserving and object-creating operators; an SQL-like *ad hoc* query language (TQL) for user-level retrieval of objects; user-level definition and control languages (TDL and TCL) for defining new types, classes, behaviors, functions, etc., and for controlling an interactive session with the query processor; a rigorous definition of safety based on the *evaluable* class of queries which is arguably the largest decidable subclass of the *domain independent* class; a notion of completeness that includes reductions between the algebra and calculus that prove their equivalence and a reduction from the user-level language to the calculus; and in addition to the formal aspects, we give a complete algorithmic translation from the calculus into the object algebra. At this point, the algebraic expressions can be optimized and an execution plan can be generated and passed to the storage manager for processing.

A prototype implementation of the object model on top of the EXODUS storage manager is ongoing. We are implementing a compiler for the user language and an extensible query optimizer for the algebra. Furthermore, we are developing a view manager, with update semantics, for the model.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

To meet data and information management requirements of new complex applications, object management systems (OMSs)[1] are emerging as the most likely candidate. The general acceptance of this new technology is dependent on the increased functionality it can provide, and two important measures lie in the power of its query model and its languages. User requirements of these systems demand a declarative language to formulate queries by focusing on "what" information is needed and leaving it up to the system to determine "how" to efficiently retrieve the information. Therefore, the formal query model of these systems define an object calculus as a theoretical framework for supporting formulation of declarative queries and a procedural or functional algebra, equivalent in expressive power to the calculus, to execute them efficiently.

In this report, we present an extensible uniform behavioral object query model and its languages. Our work is conducted within the framework of the TIGUKAT[2] project. TIGUKAT is an extensible uniform behavioral object management system. Its object model is characterized by a purely behavioral abstraction of types and a uniform approach to objects [PÖS92]. The high-level behavioral abstraction of the object model may be mapped to a variety of structural models. The uniformity of the model abstracts everything, including types, classes, behaviors, functions, meta-information and so on, as a first class object with well-defined behavior. The query model is a direct extension of the object model in that queries are defined as type and behavior extensions to the base model, meaning they inherit all behaviors of objects including their semantics. This makes for an extensible query model that is uniformly integrated with the base object model. Some advantages of this approach are that we have a behavioral-theoretic definition of a query model that is consistent with the base object model and since the query model is integrated with the object model, it itself is queryable. For example, we may query a collection of queries to gather some statistical information about them or may query the types and behaviors of the query model to examine its definition. These types and behaviors can be easily extended using object-oriented techniques to evolve the query model as more advanced features are demanded of it. This illustrates an advantage of our approach in designing a uniform extensible query model since advanced information processing features can be added as they are required.

[1] We prefer the terms "objectbase" and "object management system" over the more popular terms "object-oriented database" and "object-oriented database management system" since not only data in the traditional sense is managed, but objects in general which include things such as code and complex information in addition to data.

[2] TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning "objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

Other identifying characteristics of the TIGUKAT query model that differentiates it from other object query model proposals are the following:

1. It incorporates a formal and powerful object calculus and object algebra with a proven equivalence in expressive power and a complete algorithmic translation from calculus to algebra.

2. Its safety criterion is based on the *evaluable* class of queries [GT91] which is arguably the largest decidable subclass of domain independent queries [Mak81].

3. It exploits object-oriented features to extend the *evaluable* class by introducing notions of *object generation* on equality and membership atoms which relaxes *range* specification requirements. The result is that a broader class of safe queries are recognized by our approach.

4. It incorporates a complete SQL-like user language called TQL (TIGUKAT Query Language), an object definition language called TDL (TIGUKAT Definition Language) and a control language called TCL (TIGUKAT Control Language). TQL is proven equivalent to the formal languages making it easy to perform logical transformations and argue about its safety.

5. It uniformly models queries as first class objects by directly defining them as type and behavior extensions to the TIGUKAT object model. This makes for an extensible query model that has a consistent uniform underlying semantics commensurate with the object model.

6. The extensible algebra specification forms a uniform basis for processing queries and is exploited by our extensible algebraic query optimizer and execution plan generator which are topics of a forthcoming paper.

7. To the best of our knowledge, it is the first extensible uniform behaviorally-oriented query model to formally bring together the components of an object-oriented user language, complete object calculus and object algebra definitions, proofs of completeness between the languages, and an algorithmic translation from the calculus to the algebra. A parser for the user language is being developed and integrated with the calculus to algebra translation.

Even though our work is within the context of the TIGUKAT project, the results reported here extend to any system based on a uniform behavioral object model where behaviors define the semantics of types and are implemented in a functional paradigm.

The remainder of the report is organized as follows. In Chapter 2, we discuss some of the earlier work on object query models. In Chapter 3, we give an overview of the TIGUKAT object model. This outlines the fundamental features of the model and gives the specification of the primitive type lattice. In Chapter 4, an overview of the TIGUKAT query model as an extension to the object model is presented, and the concept of queries as objects is introduced. In Chapter 5 the formal object calculus is defined that builds on the behaviors of the object model to form atoms and the well-formed formulas which make up calculus expressions. The class of *safe* calculus expressions is also defined. In Chapter 6, the syntax and semantics of the TIGUKAT Query Language (TQL), TIGUKAT Definition Language (TDL) and TIGUKAT Control Language are given. In Chapter 7, the operators of the formal object algebra are presented along with a brief overview of the type creation and inferencing mechanisms used by our algebra. In Chapter 8, we define a Geographic Information System (GIS) as an example objectbase application and present several example queries expressed in their equivalent TQL, object calculus and object algebra forms. The reader may want to refer to the examples in this chapter while reading the earlier parts of the report. In Chapter 9, we

give the theorems and proofs showing the equivalence between the languages of the query model, along with an algorithm that translates safe object calculus expressions into equivalent object algebra expressions. Finally, Chapter 10 contains our concluding remarks and a brief discussion of the ongoing work.

Chapter 2

Related Work

One reason for the broad acceptance of relational database management systems (DBMSs) is their implementation of a high level, declarative query facility which provides an elegant and simple interface to the underlying model. One of the most popular query languages in those systems is SQL which has become an international standard for the definition and management of relational data [Dat87].

In order to consistently extend the functionality of relational systems, next generation DBMSs must extend the power of the relational query model and SQL. Therefore, one of the problems facing object-oriented systems designers is the definition of an object query model and language for these systems.

The power and expressiveness of a query model is characterized by its calculus, its algebra, its notion of safety and its completeness. The usability of a particular query model is measured by the user languages developed for it. In this chapter, we examine some of the recent literature on these topics including:

- framework papers that discuss the qualities of query models and serve as guidelines for query model development;
- complex object query models that, like our behavioral query model, have complete definitions of an algebra, a calculus and a link between them;
- specific complex algebras that introduce object-oriented operators and semantics that are exploited and expanded on in our object algebra;
- user languages that are similar to the SQL-like syntax and semantics of our language.

2.1 Query Model Frameworks

Although there is not one single universally accepted object-oriented model, a core set of features has been identified and presented in a number of manifestos [ABD+89, SRL+90]. Similar guidelines for the design of an object query model and user language have recently appeared as well. They are summarized below.

Yu and Osborn [YO91] define a framework for evaluating the power and expressibility of object algebras. A set of categories is proposed for measuring the object-orientedness, expressiveness,

formalness, performance and database issues support of an algebra. The framework is not meant to be all inclusive. In fact, some of the recommendations are contradictory requiring compromise in a design. To illustrate the practicality of the framework, four object algebras are compared within its dimensions. The framework serves as a useful guideline for developing object algebras.

The object query module specification [Bla91] of the DARPA open object-oriented database [FKMT91] offers a structured discussion of language features that an object query language should provide. Some of the more general properties which distinguish object query models from others are classified into "essential" and "non-essential" categories. This is supplemented by a more detailed discussion of specific features which are organized into a framework that defines an overall design space for object query languages. This framework is intended to serve as a reference model and is expected to accommodate a broad spectrum of existing and future object query model definitions. The reference model is similar to that of Yu and Osborn [YO91] and assists in understanding the dimensions of object query model design by providing a common foundation for comparing and reasoning about existing object query language definitions. This in turn helps to identify common areas of agreement which may lead to an eventual standardization of object query model features.

In [ÖSP93] we examined several issues relating to design alternatives for an object query model in the context of knowledge base systems. This work focused on presenting a general discussion of the key issues concerning query model design, how a particular set of choices are carried through to an object query model definition, and the ramifications of the choices made. Several of the alternatives outlined in that report were readdressed during the development of this query model.

2.2 Complete Object Query Models

Several object query models have been proposed. Many focus on a particular language aspect such as a calculus, an algebra or a user language. Others define a complete model, but in order to deal with safety they restrict their languages in certain ways. Many query models are built on the nested *set-and-tuple* style structural model. Ours differs in that it is a purely behavior-theoretic approach, defining the query model as an extensible part of the base object model. Some complete query models influencing our design are examined below.

The emphasis of Straube and Özsu's [SÖ90a, Str91] work is to illustrate the viability of developing a query processor for an object-oriented database system with comparable power and expressibility available in relational systems. A formal methodology for object-oriented query processing is developed in line with the relational paradigm. That is, a high-level declarative calculus is defined, optimization techniques on the calculus are developed, an object-oriented algebra is defined, translation of conjunctive calculus formulas with limited negation into the algebra is defined, algebraic type-checking and optimization strategies based on traditional and object-oriented transformation rules are developed, and an execution plan generation mechanism is designed that translates optimized algebraic expressions into an execution plan that consists of a series of packaged object manager calls. This increases efficiency of query processing by reducing the number of times the query processor must cross the bridge to the object manager.

One contribution of [SÖ90a, Str91] is the definition of both an object algebra, an object calculus and the linking of the two with translations between them. The algebra-to-calculus translation is complete while the calculus-to-algebra transformation is not. The algebra defines a comprehensive set of *object-preserving* operators, but lacks *object-creating* operators such as *project* and *join*[1].

[1] *Object-preserving* operators are limited to returning existing objects from an objectbase while *object-creating*

Furthermore, the classification of "safe" queries is limited to conjunctive queries without existential negation, meaning there is no allowance for universal quantification in the translation.

Abiteboul and Beeri [AB93] have defined a query model for complex objects that is based on a *set-and-tuple* data model. Their model includes set and tuple type constructors that relax the common restriction of alternating set and tuple structuring. This allows for arbitrary structures with the only restriction being that the last constructor used is a set. The calculus and algebra have complete definitions and include extended set operations such *set-collapse* for collapsing sets of sets, *powerset* for forming the powerset of a given set and a higher-order restructuring operator called *replace* that generalizes relational projection and provides set-and-tuple restructuring capabilities. Safety in their model is defined constructively similar to the (*range*) *restricted* formulas in [Ull88]. They assume that a partial order on the variables has been defined, and based on this ordering, form range terms for variables. The range terms *restrict* the domains of the variables. Constructions are defined that build safe formulas from range terms using conjunction, disjunction, quantification and negation. With this approach, safety is dependent on how the formula is constructed from the ground up and does not take advantage of the structure of the formula to recognize a broader class of queries. The class of safe queries recognizable by this approach is a strict subset of the *evaluable* class of queries that we use as a basis for our safety criteria. Although the formal work of [AB93] is sound, an algorithmic definition of safety and a calculus-to-algebra translation algorithm are not given. Furthermore, an efficient solution for their transformation is not apparent since it requires the formation of *DOM* sets for each variable appearing the formula. These sets consists of all possible values from the database (and the constants in the query) that the variable can take on. With complex valued variables allowed in their calculus, these sets can become quite large.

2.3 Complex Object Algebras

An algebra is usually one of the first components developed for a query model. Its design determines the ease with which data can be retrieved. Several complex object algebras have appeared in recent years evolving from the nested relational models and functional approaches. A select number of proposals related to our algebra are discussed below.

PDM (PROBE Data Model) [MD86] builds on the functional model and language of DAPLEX [Shi81]. It defines an algebra-based query model that is an extension of the relational algebra. It is a functional algebra that defines traditional relational operators, plus an "apply-and-append" operator that provides a functional notion of the join operator. Apply-and-append accepts a relation (essentially a function) and a function over this relation as arguments. It returns a relation containing the columns of the original relation, plus an additional column holding the result of applying the function to each tuple of the original relation. Thus, the relation acts as the first operand of a join and the function defines the second operand, plus the join term. A nearly identical approach is described by the OOAlgebra of OODAPLEX [Day89]. We define a variant of these approaches in our algebra since their uniform functional approach fit in naturally with the behavioral nature of our query model.

The object algebra of Shaw and Zdonik [SZ89, SZ90] is a *set-and-tuple* model that consistently extends the relational algebra with both object-preserving and object-creating operators. The algebraic operators work on collections of objects which have parameterized set types. The algebra defines traditional set operations along with a *flatten* operator for collapsing sets of sets. For tuples,

operators may create new objects during their execution [SS90].

nest and *unnest* operators are defined to restructure the representation of tuples as flat or nested relations. In addition to these, they define a traditional *select* operator, an *image* operator that applies a function to each object of a collection and returns the results as another collection, a *project* operator as an extension of *image* that returns a newly constructed tuple object for each object of a queried collection, and an *ojoin* operator to serve as a *cartesian product* between two collections of objects. The result of an *ojoin* is a set of object pairs with the elements of each pair containing objects from the original collections that satisfy the join condition.

Osborn [Osb88] defines an algebra for an object-oriented model based on atomic objects, strongly typed aggregates (tuples) and both homogeneous and heterogeneous sets. A fairly comprehensive set of algebraic operators is defined. The algebra is multi-sorted since the operators are defined over multiple types (sorts) of objects and are undefined for certain combinations of these types. Operators include traditional set operations, a *combine* operator that is equivalent to cartesian product for sets and has a similar semantics for aggregates, a *partition* operator for carving up aggregate objects only, and a *choose* operator which is a generalization of the relational *select*. The objects created by *partition*, and the types to which they belong, are all grouped under a "CreatedAggregates" class. There is no relationship between CreatedAggregates and the classes from which the new objects are derived. The integration of the results of *combine* with the existing lattice is not specified.

Kim [Kim89] defines the query model for the ORION OODBMS. The simple form of a query in this model is restricted to a single target class. Queries always return a new class with new object instances created from the objects in the target class. Thus, the algebra is strictly object-creating. The integration of new classes into the existing lattice is achieved by hanging them off the root. Reasoning about the type of the result class to better integrate it with the existing lattice is not defined. Single operand queries are too restrictive as they do not allow explicit joins. Therefore, the model extends queries over multiple target classes. However, there is a restriction on the domains of the "join attributes" of a query in that they must be identical or in a sub/supertype relation with one another. The result of a multiple-operand query, as with single-operand ones, is a new class with new object instances that hang off the root of the lattice.

Davis [Dav90] defines a formal object algebra that includes a full set of algebraic operators. Classical object-preserving set operators and a *select* are defined which are closed on sets (they use classes). The relative position of the resulting classes in the class lattice is derived by manipulating class properties which are in membership normal form (MNF). A *property restriction* operator, similar to select, is used to extract objects with particular properties and form a class of these objects which is a subclass of the operand. The algebra also defines *project* and *cross product* operators for "taking apart" and "putting together" objects, respectively. These two operators create new objects and form new classes that are not integrated with the classes from which they were formed. Thus, the results of these operators are not classified like they are with the object-preserving operators. We have defined product and a form of behavioral projection that allow us to integrate their results with the existing lattice. Moreover, our research involves the definition of procedures for deriving new types and classes of all operators and investigating how to integrate new schema objects into the existing type lattice.

2.4 User Languages

A number of object query languages have been designed. One reason for this variety is due to the inseparability of the object model and the query language. Presently, there is no universally

accepted object query language which is not surprising since there is no accepted object model either. Our approach in designing an *extensible* query model gives us an advantage by ensuring that our model will be able to meet future information processing requirements.

SQL3 [Gal92] is expected to incorporate numerous features for an object query language. It is intended to be a complete language for managing, creating and querying persistent objects. It provides facilities for defining new abstract data types (ADT), creating new functions and accessing objects. Moreover, non-traditional language statements are defined (while-loop, if-statement, branch statement) that add to the computational power of the language. However, the language is not based on any underlying object model and as a result it contains some artificial constructs (objects are mapped to relational tables) and several object-oriented features are missing (definition of sets, classes or other container objects). We note that SQL3 is still being designed and the standard specification is not expected to be released until 1995. A number of these problems may be solved by then.

Blakeley [Bla91, Bla93] addresses the query-programming language integration problem in the context of an object-oriented database that 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 mixed with 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 impedance mismatch problem is a nonissue. 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 be inputs to other queries and can be used in the *from* and *where* clauses of queries (i.e., nested queries). However, the formal semantics of the language is not defined which raises questions regarding the safety, completeness and optimization possibilities of the language. Furthermore, aggregate functions are not addressed and ZQL[C++] does not seem to support a syntax for them.

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 an object algebra that is closed on sets, and is well integrated with O++ which is the host language in Ode. Finally, queries are orthogonal to persistence, since 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 of objects and values. It is a functional language and is a subset of a host programming language. Thus, the problem of impedance mismatch does not exist. A *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. Furthermore, the semantics of the language is not based on any formal calculus.

EXCESS [CDV88] is the query language for EXODUS [CDF+88] and is different from ZQL[C++], CQL++, and O_2 languages in that it is based on QUEL rather then SQL. Its main features include the uniform treatment of sets and arrays (meaning 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 or arrays of objects, values or tuples and return sets as results. 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 was largely influenced by the SQL standard. Consequently, OSQL serves as an object description, manipulation and query language that 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 results of queries can be inputs to other queries. The syntax for nested queries in the *from* and *where* clauses is not supported.

Quite different design ideology is presented in the object query language for the ObjectStore [LLOW91, OHMS92]. 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 know paradigm 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 boolean.

OQL [ASL89] is a somewhat unorthodox object query language for object-oriented databases. 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. The syntax of OQL is not based on any common languages (neither SQL, nor QUEL) and therefore, as many argue, it is not very intuitive and rather unclear.

Chapter 3

TIGUKAT Object Model

The TIGUKAT object model [PÖS92] is defined *behaviorally* with a *uniform* object semantics. The model is *behavioral* in the sense that all access to, and manipulation of, objects is based on the application of behaviors (operations) on objects. The model is *uniform* in that every component of information, including its semantics, is uniformly modeled by objects and has the status of a *first class object*.

Uniformity in TIGUKAT is similar to the approaches of DAPLEX [Shi81], its object-oriented counterpart OODAPLEX [Day89] and FROOM [MB90]. However, our definition of uniformity is complete in the sense that it extends over all forms of information including schema, meta-information, query model, query optimizer and so on. We adopt another significant aspect of these models: their functional approach to defining behaviors. TIGUKAT enhances this approach by extending functionality, providing a full set of precise specifications and defining an integrated structural counterpart. In this chapter, we give only a general overview of the TIGUKAT object model in order to introduce its fundamental concepts. We provide depth on only those aspects of the model that are important in regards to object query processing. For the complete model specification, including the structural counterpart, we refer the reader to [PÖS92].

An *object* is a fundamental concept in TIGUKAT. Every component of information, including its semantics, is uniformly represented by objects. This means that at the most basic level, every expressible element incorporates at least the semantics of our primitive notion for "object."

The model defines a number of primitive objects that include: *atomic entities* (such as *reals, integers, naturals, strings, characters and booleans*); *types* for defining the features of common objects; *behaviors* for specifying the semantics of operations that may be performed on objects; *functions* for specifying the implementations of behaviors over various types[1]; *classes* for the automatic classification of objects based on type[2]; and *collections, bags, posets* and *lists* for supporting general heterogeneous user-definable groupings of objects.

The primitive type system of TIGUKAT is shown in Figure 3.1 with the type `T_object` as the root of the lattice and type `T_null` as the base. `T_null` is introduced to provide, among other things, error handling and null semantics for the model. For example, there is an object `null` that is an instance of `T_null` and can be returned by behaviors that have no other result. In a similar way, we could define `undefined`, `dontknow` and error objects of this type. We can even subtype `T_null` to specialize its semantics. In the remainder of the report, the prefix `T_` refers to a type,

[1] Behaviors and functions form the support mechanism for *overloading* and *late binding* of behaviors

[2] Types and their extents are separate constructs in TIGUKAT.

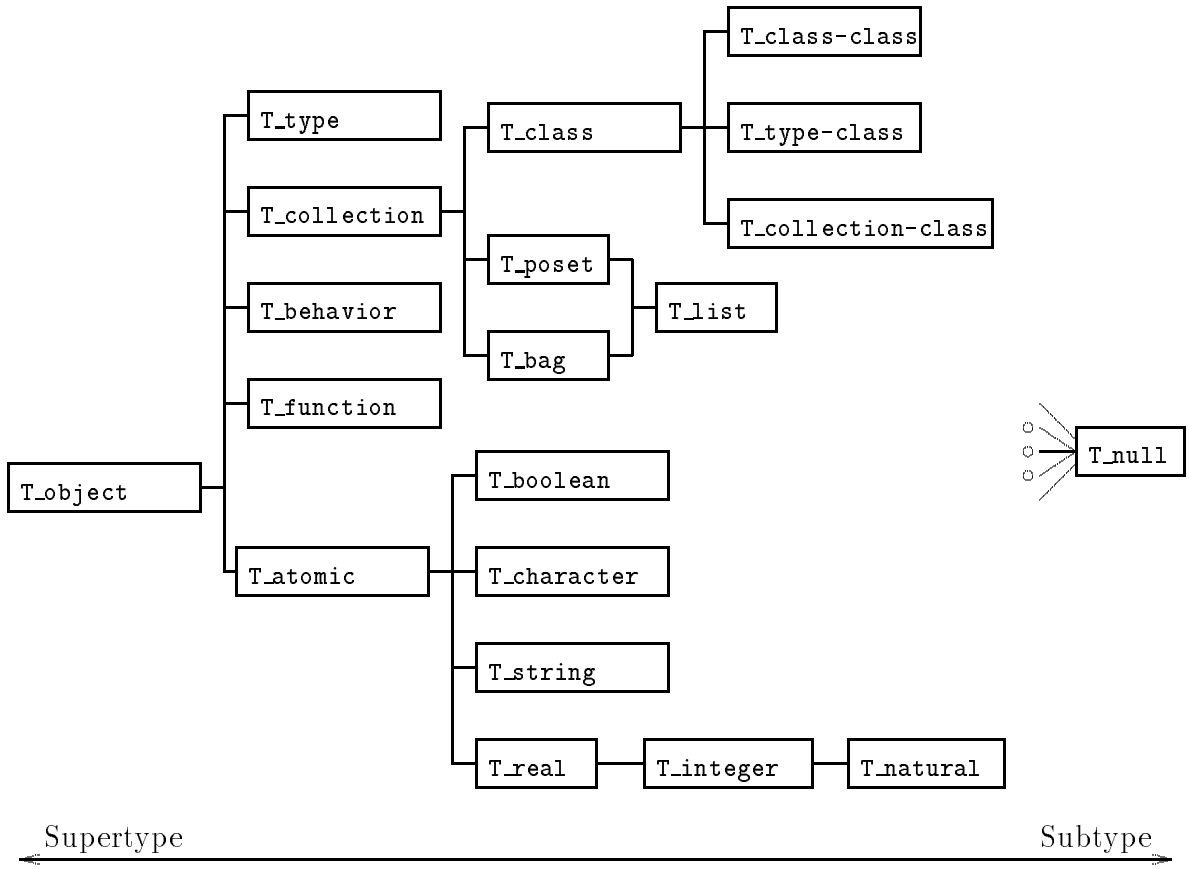

Figure 3.1: Primitive type system of TIGUKAT.

C_ refers to a class, **L_** refers to a collection, and *B_* refers to a behavior. For example, `T_person` is a type reference, **C_person** is a class reference, **L_seniors** is a collection reference, *B_age* is a behavior reference, and a reference such as sherry without any specific prefix represents some other application specific reference. In Appendix A we give tables showing the signatures for the native behaviors defined by the types in our model. Some behaviors are elaborated on in this paper, but for a complete discussion of the semantics of these and other behaviors we refer the reader to [PÖS92].

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, our model supports *strong object identity* as described in [KC86], meaning every object has a unique existence within the model and is distinguishable from every other object. However, this does not preclude application environments such as object programming languages from having many *references* (or *denotations*) to objects which need not be necessarily unique and may even change depending on the scoping rules of the application. On the other hand, the *state* of an object *encapsulates* the information carried by that object, along with its representation. In other words, the state encapsulates the *denotations* of objects and hides the structure and implementation. Conceptually, every object in TIGUKAT is a *composite* object, meaning every object has references (not necessarily implemented as pointers) to other objects. For example, even integers have behaviors that return objects, but obviously it would be inefficient to implement them as a series of pointers. This illustrates a strength in the model's separation of behaviors from their implementations.

The access and manipulation of an object's state occurs exclusively through the application of behaviors, similar to the message-based approach of Smalltalk [GR85], FROOM [MB90] and OODAPLEX [Day89]. An important primitive behavior defined on objects is that of *identity equality* (denoted *B_equal* and defined on `T_object`) that compares two object references based on their identities. In this case, two object references are *identity equal* if and only if they refer to the same object. Other notions of equality (such as those based on the structural components of objects like *deep-equality* and *i-equality* [SZ90]) can be defined in terms of the primitive identity equality and can vary over different types.

We separate the means for defining the characteristics of objects (i.e., a *type*) from the mechanism for grouping instances of a particular type (i.e., a *class*). A *type* specifies behaviors and encapsulates the implementation and representation for objects created using the type as a template. Thus, a type serves as an information repository of common characteristics for all objects of that particular type. The behaviors defined by a type describe the *interface* (denoted *B_interface* defined on `T_type`) to the objects of that type. Types are organized into a lattice structure using the notion of *subtyping* which promotes software reuse and incremental type development. Since TIGUKAT supports *multiple subtyping* (i.e., a type can be a subtype of several other types), the type structure is potentially a directed acyclic graph (DAG). However, this DAG is converted to a lattice by *lifting* with the base type `T_null`.

A *class* ties together the notions of *type* and *object instance*. A *class* is a supplemental, but distinct, construct responsible for managing all instances created using a specific type as a template. The entire group of objects of a particular type is known as the *extent* of the type. This is separated into the notion of *deep extent* which refers to all objects created from the given type, or one of its subtypes, and the notion of *shallow extent* which refers only to those objects created from the given type without considering its subtypes. Thus, shallow extent is a subset of deep extent. In general, we use *extent* in place of *deep extent* and explicitly mention *shallow extent* when required.

Objects of a particular type cannot exist without an associated class and every class is uniquely associated with a single type. Thus, a fundamental notion of TIGUKAT is that *objects* imply *classes* which imply *types*. Another unique feature of classes is that object creation occurs only through a class using its associated type as a template for the creation. Defining object, type and class in this manner introduces a clear separation of these concepts. This separation is important in schema evolution which manipulates type objects into new subtype relationships and need not be concerned with the overhead of classes. Furthermore, more general grouping constructs, called collections, use types to define common characteristics of their member objects.

Classes represent objects that are part of the objectbase. For example, the class **C_person** represents all person objects in the objectbase and **C_object** represents all the objects in the objectbase. We assume a finite objectbase and therefore all classes are finite. There are two kinds of classes provided by the model. The one kind is called an *explicit class* because it explicitly manages its shallow extent and computes its deep extent by recursing over the shallow extents of its subclasses. The second kind is called an *implicit class* because the shallow extent is not explicitly stored, but rather is implied from the contents of the objectbase. In other words, the shallow extent of an implicit class is the (finite) collection of objects in the objectbase that belong to the class. The shallow extent of an implicit class can be computed by scanning the objectbase and returning the objects that belong to the class.

Most classes are explicit classes. However, the classes for the atomic types `T_real`, `T_integer`, `T_natural` and `T_string` are implicit. Moreover, they are special in the sense that there is a built-in mechanism for creating the constants of the these classes. The act of writing down a constant of

one of these classes (in a query for example) can be thought of as a request to return an object representing the constant, creating a new one if necessary. For example, the class **C_integer** is initialized with the object zero and by using the *B_succ* and *B_pred* behaviors on this object, any integer object can be theoretically created and returned. The act of writing down the integer constant 2, can be thought as a request to apply *B_succ* to the object zero and then to apply *B_succ* to the result. This either returns the existing object representing the integer 2, or creates a new one. This is an assurance that there is only one integer 2 in the objectbase. Any intermediate objects created along the way that are not stored in the objectbase are deleted. The reals and naturals have a similar semantics. The *B_succ* and *B_pred* behaviors on reals are limited to the precision of reals on a particular system. The class **C_string** is initialized with the empty string and string representations of all the characters of which there are a finite number. With these initial strings and the concat function any string can be created and returned. The act of writing down the string "joe" can be thought of as a request to apply *B_concat* to the string objects "j" and "o" and then to apply *B_concat* to the result and the string object "e". Of course, in our implementation of TIGUKAT we don't actually do it in this way, but instead use the "native" domains of the implementation language. The above is just a formal model that is consistent with the uniformity aspects of the object model.

We define a *collection* as a general user-definable grouping construct. A *collection* is synonymous with a set and we use the terms interchangeably. Collections are similar to *classes* in that they both represent an extent of objects, but they differ in the following respects. First, object creation cannot occur through a collection; object creation occurs only through classes. This means that collections are derived from existing objects and therefore collections are finite. Second, an object may exist in any number of collections, but it is a member of the shallow extent of only one class. Third, classes are automatically managed by the system based on the subtype lattice whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class consists of the extension of a single type (shallow extent) along with the extensions of its subtypes (deep extent). Therefore, the elements of a class are homogeneous up to inclusion polymorphism while a collection may be heterogeneous in the sense that it may contain objects of different types that are not related by subtyping. There is no distinction of shallow and deep extent for collections in the sense that a collection represents its entire extent. The subtypes `T_bag`, `T_poset` and `T_list` are specialized collections that add duplication and ordering; bags maintain duplicates. posets maintain ordering, and lists maintain both.

Classes in TIGUKAT are similar to the grouping constructs in Iris [FBC+87], ODE [AG89], ObjectStore [LLOW91] and Orion [BCG+87], while collections resemble those in EXODUS [CDV88], ENCORE [SZ90], GEMSTONE [MS87] and O_2 [LRV88]. Having both defined within the same model is beneficial in that type extents are automatically maintained through classes and users have the flexibility to define their own object groupings by means of collections. Beeri [Bee90] also identifies these two forms of grouping constructs, but does not separate them as we do here. He regards a *type* as both the specification of object structure, plus the entire extent of objects created using that type as a template. This is similar to a merger of our notions of *type* and *class*. Furthermore, Beeri does not separate type extents (classes) into shallow and deep extent as we do. He introduces the notion of *class* as a general grouping construct which we call a *collection* in our model.

In TIGUKAT, we define *class* (type `T_class`) as a specialization (subtype) of *collection* (type `T_collection`). This introduces a clean semantics between the two and allows the model to utilize both constructs in an effective manner. For example, the targets and results of queries are typed collections of objects. This means targets also include classes because of the specialization of

classes on collections. This approach provides great flexibility and expressiveness in formulating queries and gives *closure* to the query model which is often regarded as an important feature [Bla91, YO91]. The remaining subtypes of `T_class`, namely, `T_class-class`, `T_type-class` and `T_collection-class`, make up the *meta* type system. Their placement within the type system itself directly supports the uniformity definition of the model. For a full discussion on how these types support uniformity in TIGUKAT, we refer the reader to [PÖS92].

Two other fundamental notions of TIGUKAT are *behaviors* and the *functions* (known as *methods* in other models) that implement them. In the same way as object specifications (types) are separated from the groupings of their instances (classes and collections), we separate the definition of a behavior from its possible implementations (functions/methods). The advantage of this approach is that common behaviors over different types can have a different implementation for each of the types. This is referred to as *overloading* the behavior, meaning that the implementation of the behavior may vary depending on the type of the object to which it is applied. This gives the model the ability to *dynamically bind* implementations to behaviors at run time (known as *late-binding*) which is recognized as a major advantage of object-oriented computing.

The semantics of every operation on an object is specified by a behavior defined on its type. A function implements the semantics of each behavior. Alternatively, we say it provides the *operational* semantics of its corresponding behavior. The implementation of a particular behavior may vary over the types that support it. Nonetheless, the semantics of the behavior remains the same over the types supporting that behavior. We define two kinds of implementations for behaviors. One is a *computed function* that consists of runtime calls to executable code and the other is a *stored function* that is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavioral application as the invocation of a function, regardless of whether the function is stored or computed. Functions are discussed more in the query model where it is shown that queries are specialized functions. That is, queries are really objects and are uniformly described in terms of the model itself. This example of uniform specification illustrates the expressive power of the TIGUKAT object model.

Chapter 4

TIGUKAT Query Model

The design of a complete uniform behavioral object model forms a basis for an extensible object query model. The query model reported here is a uniform extension of the base object model, it defines a logical object calculus, an equivalent behavioral/functional algebra, equivalence preserving reductions between the two and an SQL-like user language. The model's extensibility can be used to describe extensible query optimization and execution plan generation, but these results are not reported here.

4.1 Query Model Overview

An identifying characteristic of the TIGUKAT query model is that it is defined as type and behavioral extensions to the base object model. The uniform behavioral paradigm of the object model is carried through into the query model. Queries are defined as a specialization of functions and the algebraic operators are defined as behaviors on the type `T_collection`. Thus, the query model is a collection of objects (types, behaviors, functions, etc.) uniformly integrated with the base model. This approach has many advantages. For example, the query model is itself queryable, meaning a query may be posed on a collection of query objects or on the types and behaviors making up the query model definition (i.e., schema). Another advantage is that we have a single underlying semantics for both the object and query model resulting in a clean integration of the two. The mechanics of this integration is explained in Section 4.2.

A distinction is commonly made [SS90] between *object preserving* and *object creating* operations in object query models. An *object preserving* operator is one whose result contains only existing objects. That is, it does not create or modify objects in any way, either explicitly or by side effects. The query formalism of Straube and Özsu [SÖ90a] considered only operations of the object preserving kind. On the other hand, *object creating* operators allow for the "taking apart" and "putting together" of objects into various new structures, with new identity, which are distinct from any existing objects in the objectbase. The objects created (especially persistent objects) must be integrated into the underlying type system, including any derived types or classes necessary for the consistent existence of these new objects.

The debate over object preserving *vs.* object creating operators has strong arguments on both sides. On the one hand, object preserving operators are important because a query language must support these kinds of queries independent of its support for object creating operators. On the other hand, object creating operators allow otherwise unrelated objects to be combined in new ways which

is important for composing new relationships among objects and reorganizing information; this has application in knowledge base systems where knowledge is acquired by forming new relationships from the existing facts. However, object creating operators introduce several problems that need to be resolved. First, new objects require a type that may not exist and must be integrated with the existing type lattice. Questions on how this type fits into the existing lattice and what behaviors it supports must be addressed. Second, the issue of query safety becomes more complex due to the introduction of new objects during query processing. For example, consider a query that creates new objects in one of its argument collections with every iteration of its evaluation. If the semantics were such that the query would continue to process these new objects, then more objects would be created and the query could go on indefinitely.

The terms *object-preserving* and *object-creating* require further clarification in the context of a uniform object model like TIGUKAT in which everything is an object. Queries in our model (at minimum) always create and return a new collection object that represents the result. Furthermore, a query may also create a new type object to go along with the collection if a proper type does not already exist. Thus, in TIGUKAT *all* queries are object-creating in one sense. If the result collection of a query contains objects created during the execution of the query, it is called a *target-creating* query; otherwise it is called a *target-preserving* query.

The user query language (TQL) has a syntax based on the SQL *select-from-where* structure, and formal semantics defined by the object calculus. Thus, it extends the relational query languages with object-oriented features. The definition language (TDL) provides functionality to create new types, classes, collections and behaviors; to define new functions in the query language or an external language; to add and remove behavior definitions to and from types; and to associate functions with behaviors on types. The control language (TCL) consists of a few simple commands for controlling a session with the query processor.

The object calculus has a logical foundation and its expressive power is outlined by the following characteristics. It defines predicates on collections (essentially sets) of objects and returns a collection of objects as a result. This property makes the language *closed* which is important for uniformity. It incorporates the behavioral paradigm of the object model and allows the retrieval of objects using nested behavioral applications, sometimes referred to as *path expressions* or *implicit joins*. It supports both *existential* and *universal* quantification over collections. It has a rigorous definition of safety based on the evaluable class of queries that is compile time checkable. Finally, it supports controlled creation and integration of new collections, types and objects into the existing schema.

The algebra has a behavioral (or functional) basis as opposed to the logical foundation of the calculus. Like the calculus, the algebra is *closed* on collections. The algebraic operators are modeled as behaviors on the primitive type `T_collection`. Thus, any subtype of `T_collection` (such as classes) may be used as an operand of an algebra operator.

A desirable property of an object query model is that the algebra and calculus be equivalent in expressive power, meaning that all queries expressed in one language can also be expressed in the other. In Chapter 9 we state our theorems and proofs which show the equivalence of our algebra and calculus, along with the reductions of the user language to the formal languages. Safety of our languages is addressed in Section 5.2.

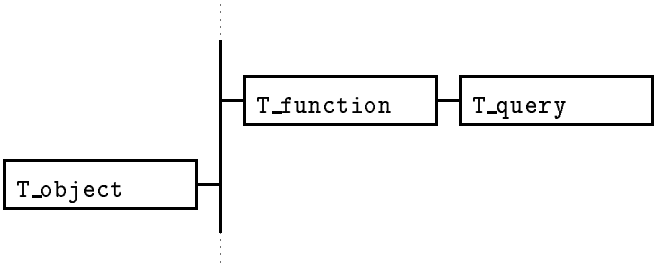

Figure 4.1: Query type extension to primitive type system.

B_argTypes:	T_list⟨T_type⟩
B_resultType:	T_type
B_source:	T_string
B_execute:	T_list → T_object
B_compile:	T_object
B_executable:	T_object
B_initialOAPT:	T_algOp
B_optimizedOAPT:	T_collection⟨T_algOp⟩
B_searchStrategy:	T_searchStrategy
B_transformations:	T_list⟨T_algEqRule⟩
B_argMbrTypes:	T_list⟨T_type⟩
B_resultMbrType:	T_type
B_optimize:	T_searchStrategy → T_algOp → T_collection⟨T_algOp⟩
B_genExecPlan:	T_algOp → T_function
B_budgetOpt:	T_integer
B_lastOpt:	T_date
B_lastExec:	T_date
B_materialization:	T_object

Table 4.1: Behavior signatures for type T_query. Upper half are inherited from T_function. Lower half are native to this type.

4.2 Queries as Objects

Modeling queries as objects is a natural extension to the TIGUKAT object model. We define a type T_query as a subtype of T_function in the primitive type system as illustrated in Figure 4.1. This means that queries have the status of *first class objects* and that they inherit all the behaviors and semantics of objects. Moreover, queries are a specialized kind of function object. This means they can be used as implementations of behaviors, they can be compiled, they can be executed and so on.

Table 4.1 lists the signatures of behaviors defined on type T_query. The upper half of the table are the behaviors inherited from T_function and the lower half are the native behaviors defined by this type.

Functions have source code associated with them and the source code for a query is a TIGUKAT query language statement as defined in Chapter 6. The behavior *B_source* retrieves this language statement from the query. Functions have a behavior *B_compile* that compiles the code. For a query, this involves translating the query statement into an algebra expression, optimizing it and

generating an execution plan. Functions have a behavior *B_execute* that executes the compiled code. In general, for a query this means submitting the execution plan to the object manager for processing. Furthermore, queries have specialized behaviors such as *B_materialization* which is a reference to the materialized query result (i.e., the actual result collection itself). If this result is made persistent, then the query is said to be *stored* and doesn't need to be re-evaluated the next time it is called upon to *B_execute* itself. Other behaviors include *B_initialOAPT* and *B_optimizedOAPT* for accessing the initial and optimized Object Algebra Processing Trees; *B_searchStrategy* for accessing the search strategy used for optimization; *B_transformations* for accessing the list of transformation rules used during optimization; *B_argMbrTypes* for accessing the membership types of the argument collections as opposed to *B_argTypes* which are the types of the collection objects themselves; *B_resultMbrType* for accessing the membership type of the result collection as opposed to *B_resultType* which is the type of the collection; and several other behaviors for keeping various statistics about queries. These behaviors relate to the extensible query optimizer and will be discussed in a forthcoming paper.

Incorporating queries as a specialization of functions is a very natural and uniform way of extending the object model to include declarative query capabilities. The major benefits of this approach are as follows:

1. Queries are *first class objects*, meaning they support the uniform semantics of objects and are maintained within the objectbase as just another kind of object.

2. Since queries are objects, they can be queried and can be operated upon by other behaviors. This is useful for retrieving information about queries, generating statistics about the performance of queries and in defining extensible optimization techniques on query objects.

3. Queries 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. The type `T_query` can be further specialized by subtyping. This can be used to dichotomize the class of queries into additional subclasses, each with its own unique characteristics, and to incrementally develop the characteristics of new kinds of queries as they are discovered. For example, we can subtype `T_query` into `T_adhocQuery` and `T_productionQuery` and then define different evaluation strategies for both. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies while, on the other hand, production queries are usually compiled once and then executed many times. Thus, more time is usually spent on optimizing production queries.

Chapter 5

The Object Calculus

It is well recognized that a declarative query facility is an essential component of any database management system; object-oriented systems are no exception. In this chapter, we present a high-level behavioral object calculus with first-order semantics.

In order to maintain the uniformity of the behavioral object model within the query model, the behavioral abstraction paradigm is carried through into the calculus. The logical foundation of the calculus includes a function symbol to incorporate the behavioral nature of the object model. This allows the use of general path expressions in the calculus. The safety of our calculus is based on the *evaluable* class of queries [GT91] which is arguably the largest decidable subclass of the domain independent class [Mak81]. We extend this class by making use of *object generators* for equality and membership atoms and this relaxes the requirement of specifying explicit *range* expressions for each variable.

We begin by presenting the *first-order theory* of our object calculus that defines the *well-formed formulae* of the language. We then describe augmentations to the theory which form *object calculus expressions* (OCEs) that represent the declarative queries posed on an objectbase.

The alphabet of the object calculus consists of the following symbols:

Object constants:	a, b, c, d
Object variables:	o, p, q, u, v, x, y, z
Predicate symbols	
monadic:	C, P, Q, R, S, T
dyadic:	$=, \neq, \in, \notin$
n-ary:	$Eval$
Function symbols:	β
Logical connectives:	$\exists, \forall, \wedge, \vee, \neg$
Delimiters:	( ) ,

Note that the object constants, object variables, monadic predicates and function symbols may be subscripted (e.g., a_3, o_i, C_n, β_1,etc.). In addition, we adopt a vector notation $\vec{s}$ to denote a countably infinite list of symbols $s_1, s_2, \ldots, s_n$ where $n \geq 0$.

From object constants and object variables we develop the syntax and semantics of the function symbol β called a *behavioral specification* (Bspec). A *term* is an object constant, an object variable or a Bspec. A Bspec is an n+2-ary function $\beta(s, b, \vec{t})$ where s and each t_i denote terms and where b is an object constant. For $n = 0$, we use $\beta(s, b)$ without loss of generality.

The ordered list of terms $s, b, \vec{t}$ is considered to be *behaviorally consistent* if and only if the following properties hold:

1. b is an object constant denoting a behavior, meaning b is not allowed to range over behaviors (functions) which ensures a first-order semantics when incorporated into a language with quantification;
2. the type of the object denoted by s defines behavior b as part of its interface, meaning b is applicable to s because it is defined on the type of s;
3. $\vec{t}$ is compatible with the arity of the argument list for behavior b, meaning the number of arguments expected by b is equivalent to the number of terms in $\vec{t}$; and
4. the types of the objects denoted by $\vec{t}$ are compatible with the argument types of behavior b, meaning the types of the terms are compatible with the argument types of b.

A Bspec $\beta(s, b, \vec{t})$ is *consistent* if and only if $s, b, \vec{t}$ are *behaviorally consistent*. In TIGUKAT, every object knows its type and therefore, we can determine the consistency of a Bspec at compile time.

The "evaluation" of a consistent Bspec involves applying the behavior b to the object denoted by term s using objects denoted by terms $\vec{t}$ as arguments. The "result" of Bspec evaluation denotes an object in the objectbase. Since Bspecs denote objects, they have a type (and a class) that are in the objectbase as well.

The "evaluation" of Bspecs has the following logical formation. We introduce the n+3-ary predicate $Eval(R, s, b, \vec{t})$ as an axiom in the language such that $Eval(R, s, b, \vec{t})$ is true if and only if R denotes the "result" of applying behavior b to the object denoted by term s using terms $\vec{t}$ as arguments. The function symbol $\beta(s, b, \vec{t})$ is a logical representation of R. The $Eval$ predicate also serves as an enforcement of the consistency property of Bspecs. From now on we consider only those Bspecs that are consistent.

Bspecs may be composed. This provides the capability of building *path expressions* in queries. For example, given the object constants emp, *B_department* and *B_budget* with the obvious semantics, we can compose the Bspec $\beta(\beta(\mathsf{emp}, B\_department), B\_budget)$ that denotes the object representing the annual budget of the department that employee emp works in. Also note that the example Bspec has the properties of a *ground term* (see Definition 5.1 below).

For brevity, we recast the syntax of Bspecs into the dot notation as $s.b(\vec{t})$ which we intend as being semantically equivalent to the original specification. If behavior b does not require any arguments, then the notation simplifies to $s.b$. The previous example can then be represented as emp.*B_department*.*B_budget* assuming left-associativity of behavioral applications. Parenthesis may be used to change the order of precedence. Some other equivalent syntax, such as function application $b(s, \vec{t})$ which is popular in other languages, could have been chosen instead.

As shown by the above example, many path expression formations often include a series of behaviors with the semantics that the result of the first behavior be used as the input to the second and so on. We call such a sequence of **m**ultiple **op**erations a **mop** [SÖ90a] which is equivalent to a Bspec. We introduce the multi-operation dot notation $<\vec{s}>.b_1.b_2...b_m$ to denote a multi-operation resulting in the application of behavior object constants $b_1.b_2...b_m$ using objects denoted by terms $\vec{s}$ as arguments. Furthermore, $<\vec{s}>.\vec{b}$ is used as a shorthand to denote a multi-operation where the number and ordering of the behaviors are immaterial.

To illustrate the processing of a **mop**, consider the multi-operation $<s_1, s_2, \ldots, s_n>.b_1.b_2\ldots b_m$. Let k_i denote the number of parameters[1] defined by behavior b_i, let r_i designate the intermediate object denoted by the Bspec formation of behavior b_i and let r denote the final result of the *mop*. Procedurely, a **mop** is processed as follows where "$\leftarrow$" denotes assignment:

$$\begin{array}{rcl} r_1 & \leftarrow & s_1.b_1(s_2, \ldots, s_{k_1+1}) \\ r_2 & \leftarrow & r_1.b_2(s_{k_1+2}, \ldots, s_{(k_1+k_2+1)}) \\ \vdots & & \vdots \\ r_i & \leftarrow & r_{i-1}.b_i(s_{(\sum_{j=1}^{i-1} k_j)+2}, \ldots, s_{(\sum_{j=1}^{i} k_j)+1} \\ \vdots & & \vdots \\ r = r_m & \leftarrow & r_{m-1}.b_m(s_{(\sum_{j=1}^{m-1} k_j)+2}, \ldots, s_n) \end{array}$$

The above sequence of behavioral application making up the **mop** is illustrated in Figure 5.1.

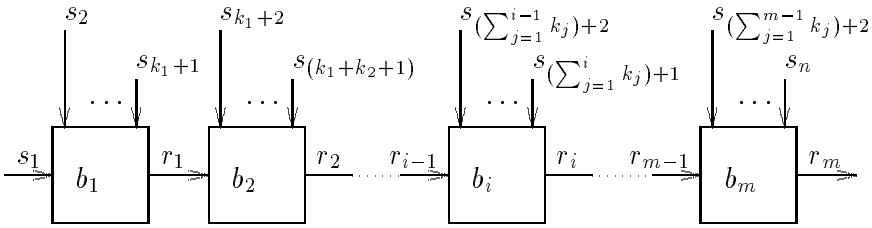

Figure 5.1: Sequence of behavioral applications making up a *mop*.

Bspecs and mops are equivalent forms of representation. One form can be freely transformed into the other and results established using one form also hold for the other. This result is important since we can transform between the formal calculus and "simplier" language notations. The equivalence is formalized by the following lemma.

Lemma 5.1 Bspecs and mops are equivalent representations.

Proof: Trivial. Due to the following equivalence mappings between Bspecs and mops where s and t represent terms and b represents behavior constants:

$$\beta(s, b, \vec{t}) \equiv <s, \vec{t}>.b \quad (5.1)$$

$$<\vec{t}>.b.\vec{b} \equiv \ll\vec{t}>.b>.\vec{b} \quad (5.2)$$

The first mapping shows that every Bspec can be replaced by an equivalent mop over a single behavior and vice versa. The second mapping shows the unnesting of mops over multiple behaviors into an equivalent series of single behavior mops which is handled by the first mapping. □

We generalize the notions of constants and variables to include Bspecs by defining *ground terms* and *variable terms* as follows:

Definition 5.1 *Ground Term:* A *ground term* is recursively defined as follows:

[1] Here the *parameters* refer to the objects supplied to the behavior, not including the initial object to which the behavior is being applied.

1. every *object constant* is a *ground term*;
2. if $\beta(s, b, \vec{t})$ is a Bspec and all of $s, \vec{t}$ are *ground terms* (note that b must be a ground term by the definition of Bspec), then $\beta(s, b, \vec{t})$ is a *ground term*;
3. nothing else is a *ground term*.

From now on, symbols defined as denoting an object constant, including symbols a, b, c, d, are extended to include ground terms as well. Any term that is not a ground term is called a *variable term* since it must contain at least one object variable. If $\vec{o}$ are the object variables appearing in some term r, then r is called a *variable term over* $\vec{o}$. The variables can be thought of as the parameters of the term. If r is the object variable o, then r is a variable term over o. If r is a term defined by Bspec $s.b(\vec{t})$ and $\vec{o}$ represents the object variables appearing in the Bspec, then r is a variable term over $\vec{o}$. We use the notation $r\{\vec{o}\}$ to denote that r is a variable term over $\vec{o}$. We generalize this notation to $\beta\{\vec{o}\}$ when the form of the term is immaterial. If $\vec{o}$ is empty, then $\beta\{\}$ denotes a generic ground term.

The *atomic formulas* or *atoms* are the building blocks of calculus expressions. They represent the fundamental predicates of the calculus. The atoms of the calculus consist of the following:

Range Atom: $C(o)$ is called a *range atom* for o where C corresponds to a unary predicate representing a collection and o denotes an object variable. We say C is the *range* of o. A range atom is true if and only if o denotes an object in collection C. When C is defined for a class, it denotes the deep extent of the class and we extend the notation to include $C^+(o)$ which is true if and only if o denotes an object in the *shallow extent* of the class. One may think of C^+ as a separate monadic predicate for specifying the shallow range of o. Range atom specifications of the form $C(s)$ where s is a term denoting an object constant or Bspec (i.e., not an object variable) are represented by membership atoms defined below.

Equality Atom: $s = t$ is a built-in predicate called an *equality atom* where s and t are terms. The predicate is true if and only if the object denoted by term s is object identity equal to the object denoted by term t. This atom is type consistent for all objects since all objects must support an object identity equality behavior. Note, as a syntactical convenience, an equality atom where both terms are boolean and where one of the terms is the object constant true, say $s =$ true where s is boolean, is simplified to s. If one of the terms is the object constant false, we simply the atom specification to $\neg s$. The built-in predicate $s \neq t$ is the complement of equality.

Membership Atom: $s \in t$ is a built-in predicate called a *membership atom* where s and t are terms and t is a term denoting a collection. The predicate is true if and only if the object denoted by s is an element of the collection denoted by t. Note that a range specification of the form $C(s)$ where s is an object constant or Bspec (i.e., not an object variable) is represented as a membership atom $s \in C'$ where C' is a constant denoting the collection represented by predicate C. The built-in predicate $s \notin t$ is the complement of membership.

Generating Atom: An equality atom of the form $o = t$ or a membership atom $o \in t$, where o is an object variable, t is an appropriate term for the atom, and o does not appear in t, are called *generating atoms* for o. They are so named because the object denotations for o can be *generated* from t. We call o the *generated variable* and t the *generator*. Any atom that is not a generating atom is called a *restriction atom* and any variable that is not generated is called a *restriction variable*.

A *ground atom* is an atom that contains only ground terms. A *literal* is either an atom or a negated atom. A *ground literal* is a literal whose atom is a ground atom.

The choice of atoms may seem restrictive when compared to other calculi such as the tuple relational calculus which defines a greater variety of comparison predicates including $=, <, \leq, >$, and $\geq$. An identifying characteristic of our calculus is that it is strictly behavioral and does not define explicit value-based comparisons of objects or their subcomponents. Thus, operations such as $<, >, \geq, \leq$ must be defined as behaviors on the respective types of objects that are to be compared. The only comparison predicates defined are object identity equality and membership. However, type implementors can specialize the behaviors for these comparison predicates in their own types (e.g., value based comparisons) that are of most utility to them. For example, a form of "structural equality" on cartesian product types that compares two product objects based on the pairwise equality of their respective component objects can be defined.

From atoms, we build the definition of a *first-order well-formed-formula* or simply *formula* (abbreviated WFF) of the object calculus. WFFs are defined in terms of *free* and *bound* object variables. An object variable is *bound* in a formula if it has been previously introduced by the quantifier $\exists$ or $\forall$. If the variable has not been introduced with a quantifier it is *free* in the formula. WFFs are defined recursively as follows:

1. Every atom is a formula. All object variables in the atom are free in the formula.
2. If ψ is a formula, then $\neg\psi$ is a formula. Object variables are free or bound in $\neg\psi$ as they are free or bound in ψ.
3. If ψ_1 and ψ_2 are formulas, then $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ are formulas. Object variables are free or bound in $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ as they are free or bound in ψ_1 or ψ_2.
4. If ψ is a formula, then $\exists o(\psi)$ is a formula. Free occurrences of o in ψ are bound to $\exists o$ in $\exists o(\psi)$.
5. If ψ is a formula, then $\forall o(\psi)$ is a formula. Free occurrences of o in ψ are bound to $\forall o$ in $\forall o(\psi)$.
6. Nothing else is a formula.

We use A, B, F, G and ψ, ω to denote formulas and subformulas. We use the relation "$A \stackrel{\text{def}}{=} F$" as meaning symbol A "is defined by" the expression F. This is used to associate formula symbols with formulas. Furthermore, we use $A(x)$ to denote that variable x is free in formula A. Formulas may be enclosed in parenthesis to indicate order of precedence. In the absence of parenthesis, we adopt the following precedence hierarchy, with the highest precedence at the top:

$$\neg, \exists, \forall$$
$$\wedge$$
$$\vee$$

5.1 Calculus Queries

Several classifications of object-oriented queries have been made. One class of queries deals only with behaviors that are *side-effect free*. A behavior is said to be *side-effect free* if it does not

modify the state of any object or create new objects during its execution. This property is too restrictive in the context of our model since all operations (including the algebraic operators) are uniformly managed as behaviors. At minimum, a query always returns a new collection as a result and in certain cases generates a new type for the collection as well. Thus, there is a small set of predefined behaviors that manage the controlled creation of collections (and possibly types) as their side effects. These behaviors include the algebraic operators and the primitive behaviors for collection creation and construction. We use the notation $newcoll(o_1, \ldots, o_n)$ as a shorthand to represent the creation of a collection containing objects $o_1, \ldots, o_n$. The primitive sequence of behavioral applications corresponding to this notation is as follows:

$$\textbf{C\_collection}.B\_new.B\_insert(o_1)\ldots B\_insert(o_n)$$

A new empty collection is created and then each object o_i is added to the collection in turn. The result is a collection containing objects $o_1, \ldots, o_n$. A compiler could optimize this series of $n+1$ behavioral applications into a single internal primitive collection creation operation since collections are part of the primitive model.

We assume that all user-defined behaviors appearing in calculus expressions are side-effect free. In other words, all user-defined behaviors appearing in calculus expressions must be retrieval oriented.

A target-preserving query is an *object calculus expression* (OCE) of the form $\{t \mid \psi\}$ where t is a target term consisting of a single variable, say o, possibly indexed by a set of behaviors, ψ is a WFF with o as the only free variable, and *all* behaviors in the expression are side-effect free or retrieval oriented.

Indexed variables are of the form $o[\mathcal{B}]$ where $\mathcal{B}$ represents a subset of behaviors defined on the type of variable o, unioned with the behaviors defined on type `T_object`. The union with `T_object` is necessary since every object must support the behaviors of `T_object`. The semantics of indexed terms is to *project* over the behaviors in $\mathcal{B}$ for variable o creating a new type for the result. Following a projection, the membership type of the result collection will be a type that only defines the behaviors in $\mathcal{B}$. This resticts the behaviors that can, in general, be applied to the members of the result collection.

Target-preserving queries may seem to be somewhat simplistic and too restrictive, but this form supports a wide variety of useful queries. For example, assume finite classes **C_dept** and **C_emp** where **C_emp** objects have behaviors *B_dept* and *B_age* defined on them. The following target-preserving query returns a collection of department objects that have senior citizens working for them:

$$\{\ o \ \mid\ \textbf{C\_dept}(o)\ \wedge\ \exists p(\textbf{C\_emp}(p) \\ \wedge\ o = p.B\_dept\ \wedge\ <p, 65>.B\_age.B\_greaterThan)\ \}$$

All queries that are not target-preserving are target-creating. We extend the notation of OCEs for target-creating queries to include the form $\{t_1, \ldots, t_k \mid \psi\}$ where the set of variables appearing in (possibly indexed) target terms $t_1, \ldots, t_k$ is precisely the set of free variables, say $\vec{o}$, in the WFF ψ. This form is a generalization of the target-preserving kind by allowing $k \geq 2$ target terms over $\vec{o}$ distinct object variables. The result of this second form of query is a collection of product objects created by joining permutations of t_1 through t_k that satisfy ψ. Assume that in the previous example we wanted to return (department, employee) pairs instead of just departments and that the employee objects are projected over behavior *B_age*. The target-creating query that produces

this result is as follows:

$$\{\ o, p[B\_age] \quad | \quad \mathbf{C\_dept}(o)\ \wedge\ \mathbf{C\_emp}(p) \\ \wedge\ o = p.B\_dept\ \wedge\ <p, 65>.B\_age.B\_greaterThan\ \}$$

Additional examples of both target-preserving and target-creating queries are given in Chapter 8.

5.2 Safety of Object Calculus Expressions

A traditional notion in relational database systems is that "reasonable" queries are ones whose correct answers contain values that are limited to the constants that appear in the query or the database relations that appear in the query. A corresponding notion in an object model is that reasonable queries produce correct answers that contain objects which are limited to the objects appearing the query or in the collections that appear in the query. Unary predicates $C(o)$ are defined for the finite collections and classes appearing in the objectbase. These are used to range over the elements of a collection. The collection represented by the complement of a predicate is assumed to be infinite (i.e., $\neg C(o)$ is infinite for all predicates C).

The object calculus is very expressive and allows for the formation of queries that have no "reasonable" interpretation. For example, the complement of a predicate $\neg C(o)$ holds for arbitrary objects o that are not in the collection C. Another problematic query is the one that adds objects to collections over which it is ranging. This has the effect of updating the predicate on each iteration. These kinds of queries are considered "unreasonable" and in an implementation we wish to strictly avoid processing such constructs. Therefore, we define a criterion of *safety* and some tests, based on the structure of the formula (i.e., its syntax), to check if a formula is safe. We only process queries that are safe and reject those that do not pass the tests. The general notion of safety is defined as follows.

Definition 5.2 *Safety:* An expression is considered *safe* if it can be evaluated in finite time and produces finite output [OW89].

The above definition is a semantic one which raises the problem of finding an efficient solution for determining whether an arbitrary expression is safe or not. In other words, we would like to define a syntactic check that could be performed on any arbitrary formula and could tell us, in polynomial time, whether the given formula is safe or not. The safe formulas are the ones translated to an algebra, optimized and executed. Since the implementations of behaviors can be arbitrary code, we can only guarantee safety up to Bspec evaluation. That is, we have no mechanisms to guarantee the termination of a function that may be called as part of a behavior being applied to an object.

The first safety check is on the calculus formula and determines the *domain independence* of the formula. The second check is based on the operators of an equivalent algebra expression for the formula and determines the *operand finiteness* of a query, meaning it checks that objects aren't being added to operand collections or classes of the operator. If the query fails either test, it is rejected. We first talk about the domain independence form of "safety" and then switch our discussion to the operand finiteness of queries.

The class of *domain independent* formulas [Mak81, Fag82] is recognized as being the largest class of "reasonable" queries. However, the undecidability of this class is well known; Nicolas and

Demolombre [ND82] have shown domain independence to be equivalent to the class of *definite* formulas defined by Kuhns [Kuh67] which has been shown to be not recursive by DiPaola [DiP69].

Many decidable subclasses of the domain independent class have been proposed. The class of conjunctive queries are those which include only $\exists$ and $\wedge$ connectives and is one of the simplest "reasonable" subclasses shown to be decidable [Ull82]. Larger decidable subclasses augment conjunctive queries with negation and disjunction. Several object calculi proposals have defined safety in the context of conjunctive queries with disjunction and restricted forms of negation [SÖ90a, Cha92]. These proposals define a broader range of safe queries, however, more general classes have been identified. The class of *evaluable* queries as first proposed by Demolombre [Dem81] and later examined by Gelder and Topor [GT87, GT91] is argued as being the largest decidable subclass of domain independent queries. In our query model, we use the evaluable class as the base set of safe queries that can be translated into the object algebra. The class of *range restricted* queries [Dem82] has been shown to be equivalent to the evaluable class [GT91]. A strict subclass of the range restricted class (hence the evaluable class) is essentially the basis of safety in the structural query model of Abiteboul and Beeri [AB93]. Furthermore, their definition assumes the existence of a partial order on the variables in a calculus formula such that all variables are *restricted*. An indication of how to construct a proper partial ordering from a given formula is not presented. Our safety model also defines a partial order and the first part of the translation from calculus to algebra (see Section 9.2) constructs this ordering.

The class of evaluable queries can be defined in terms of the two relations *gen* and *con* (see Figure 5.2) between variables and (sub)formulas. These relations were introduced by Gelder and Topor [GT87, GT91] in the form of logical rules.

Intuitively, $gen(x, A)$ means that 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. In other words, if $gen(x, A(x, \vec{y}))$ holds and $A(c, \vec{d})$ is true for some variable assignment $x = c$ and $\vec{y} = \vec{d}$, then we can conclude that c is an element of a finite collection of objects derivable from the formula A itself. If $con(x, A(x, \vec{y}))$ holds, then the variable x is said to be *constrained* in A, meaning that x is generated in every disjunct of A in which x appears. The *con* rules subsume the *gen* rules. Thus, it is clear that $gen(x, A)$ implies $con(x, A)$, but $con(x, A)$ does not imply $gen(x, A)$.

We extend these rules by adding a *gdb* relation that makes use of generating atoms in formulas. The *gdb* relation relies on a globally accessed partial order denoted $<_F$. This partial order consists of pairs (x, N) where x is a variable and N is a positive integer or the symbol ∞. We also use $<_F$ in the *gdb* rules as an infix dyadic predicate on the variables appearing in the partial order $<_F$. This predicate is defined as follows:

Definition 5.3 *Ordering Predicate* ($<_F$)*:* For any two elements (x, N_x) and (y, N_y) appearing in the partial order $<_F$, the predicate $x <_F y$ is defined by the following table where n and m denote positive integers and m is greater than zero:

N_x	N_y	$x <_F y$
∞	∞	false
∞	n	false
n	∞	true
n	$n + m$	true
$n + m$	n	false
n	n	false

Figure 5.2 shows the rules for the *gdb* relation and the extended *gen* and *con* relations. The partial order used by the *gdb* relation is built from the atoms in a calculus formula F during the first step in the translation from the calculus to the algebra. The partial order is constructed to produce a representation of the generating atom dependencies between variables in a formula F. If predicate $x <_F y$ holds for the partial order $<_F$, this means that variable x is not dependent on variable y and that x potentially generates values for y in formula F. For example, the partial order for the formula:

$$F \stackrel{\text{def}}{=} \exists x(\mathbf{C\_emp}(x) \wedge y = x.B\_name)$$

is $<_F \stackrel{\text{def}}{=} \{(x,0),(y,1)\}$ since x is generated independently of y from **C_emp** and y is generated using x in $y = x.B\_name$. The reason we say x "potentially" generates y is clear from the following example. Consider the formula:

$$F' \stackrel{\text{def}}{=} \exists x \exists w(\mathbf{C\_emp}(x) \wedge y = x.B\_name \wedge \mathbf{C\_emp}(w) \wedge z = w.B\_age)$$

The partial order for this formula is $<_{F'} \stackrel{\text{def}}{=} \{(x,0),(w,0),(y,1),(z,1)\}$. Now, $x <_{F'} z$ holds and x is not dependent on z, but x does not generate objects for z in F'. Thus, x is only a potential generator for z.

The additional predicates and functions that appear within the rules of Figure 5.2 are defined as follows:

- Predicate *edb*(A) holds if one of the following conditions is met:
  1. formula A is a range atom of the form $C(x)$ where predicate symbol C represents a finite collection;
  2. formula A is an equality atom of the form $x = c$ where c is a ground term; or
  3. formula A is a membership atom of the form $x \in c$ where c is a ground term representing a finite collection.
- Predicate *free*(x, A) holds if variable x appears as a free variable in formula A.
- Predicate *notfree*(x, A) holds if variable x is bound in formula A or if x does not appear in A.
- Predicate *distinct*(x, y) holds if x and y are different variables.
- Function *pushnot*$(\neg A)$ represents a formula B (provided *edb*(A) does not hold) that is evaluated as follows:

$\neg A$	B
$\neg(A_1 \wedge A_2)$	$(\neg A_1) \wedge (\neg A_2)$
$\neg(A_1 \vee A_2)$	$(\neg A_1) \vee (\neg A_2)$
$\neg\exists x\, A_1$	$\forall x \neg A_1$
$\neg\forall x\, A_1$	$\exists x \neg A_1$
$\neg\neg A_1$	A_1
$\neg(s = t)$	$s \neq t$
$\neg(s \neq t)$	$s = t$
$\neg(s \in t)$	$s \notin t$
$\neg(s \notin t)$	$s \in t$

  If *edb*(A) holds, then *pushnot*$(\neg A)$ represents a formula, say $\perp$, that causes the corresponding *gen* or *con* predicate to fail.

$gdb(x, x = y)$	**if**	$y <_F x$
$gdb(x, x = \beta\{\vec{y}\})$	**if**	$\vec{y} <_F x$
$gdb(x, x \in y)$	**if**	$y <_F x$
$gdb(x, x \in \beta\{\vec{y}\})$	**if**	$\vec{y} <_F x$

$gen(x, A)$	**if**	$edb(A)$ **and** $free(x, A)$
$gen(x, A)$	**if**	$gdb(x, A)$
$gen(x, \neg A)$	**if**	$gen(x, pushnot(\neg A))$
$gen(x, \exists y A)$	**if**	$distinct(x, y)$ **and** $gen(x, A)$
$gen(x, \forall y A)$	**if**	$distinct(x, y)$ **and** $gen(x, A)$
$gen(x, A \vee B)$	**if**	$gen(x, A)$ **and** $gen(x, B)$
$gen(x, A \wedge B)$	**if**	$gen(x, A)$
$gen(x, A \wedge B)$	**if**	$gen(x, B)$

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

Figure 5.2: Logical rules that define the *gen* and *con* relations.

From the relations of *gen* and *con*, the class of *evaluable* [GT91] formulas is defined below. The class of formulas satisfying this definition (or which can be rewritten to satisfy the definition) is exactly the class of "safe" formulas of our calculus.

Definition 5.4 *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.

This definition provides an efficient, syntactic approach for determining whether a given formula is evaluable or not: simply apply the appropriate *gen* and *con* rules to the formula and subformulas. We extend this definition to object calculus expressions by stating that an OCE $\{\vec{t} \mid \psi\}$ where $\vec{t}$ contains at least one target term, is evaluable if the formula ψ is evaluable in the sense of

Definition 5.4. This establishes the decision mechanism for accepting or rejecting any arbitrary query posed as an OCE. For example, assuming all range predicates represent finite collections, the following OCE is evaluable:

$$\{o \mid C(o) \wedge \exists p(P(p) \vee \neg Q(o))\}$$

while:

$$\{o \mid C(o) \wedge \exists p(\neg P(p) \wedge p.B\_something = o.B\_something)\}$$

is not because $con(p, \neg P(p) \wedge p.B\_something = o.B\_something)$ does not hold. Note that the evaluable OCE above as given is an example of a formula that is safe in the evaluable class, but is unsafe in the (range) restricted class as defined by [AB93].

Without a partial order defined (i.e., we cannot make use of the *gdb* predicate), formulas satisfying Definition 5.4 are known as *strict-sense evaluable* [GT91] because of the conservative approach taken towards the built-in equality and membership predicates: $gen(x, x\theta y)$ and $con(x, x\theta y)$ where θ is one of $=, \in$ never hold. The strict-sense evaluable queries are the class considered in [GT91]. However, they realized that many formulas are evaluable despite this conservative approach. They presented transformations that remove some instances of equality ($=$) and yield an "equality reduced" form. However, a more general solution was needed for our model to deal with Bspecs and generating atoms that were not part of their work. Our introduction of the *gdb* predicate and the formation of the partial order $<_F$ consistently extends the class of evaluable queries to a larger class recognized in [GT91]. Formulas that fail strict-sense evaluability, but can be made evaluable through transformations or rule extensions are known as *wide-sense evaluable*.

This concludes the definition of our syntactic based check for recognizing the domain independence of a formula. We check if the formula is in the evaluable class of queries which is arguably the largest decidable subclass of the domain independent class. Once it is known that an OCE is evaluable, there are a finite number of steps (described in Chapter 9 by the calculus to algebra reduction Theorem 9.3) that translates any evaluable OCE into an equivalent object algebra expression (OAE).

The second test for "safety" determines whether a query adds objects to the collections and classes that it is ranging over and to reject it if it does. We call this form of safety the check for *operand finiteness*. An example calculus expression that exhibits this problematic operation is as follows:

$$\{o \mid \exists p(\mathbf{C\_collection}(p) \wedge o = newcoll(p))\}$$

This query ranges over the class of collections and for each collection p in this class it creates a new collection containing the collection p. The problem is that the new collections are created as instances of class **C_collection**, thereby increasing the cardinality of **C_collection** for every object in **C_collection**. The semantics of the query is to range over all members of **C_collection**, meaning the newly created collections should be included in the range of p which results in the creation of more collections and so on. We defer the check for operand finiteness until after the generation of an equivalent algebraic expression and perform this check on algebraic operators (see Section 7.3). This is done because the algebraic expression defines the structure of the query and we define a recursive process that goes through and tests each operator in turn.

Chapter 6

The TIGUKAT User 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 TIGUKAT language as well as in other programming language like for example C++. 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 allows the retrieval of objects in a TIGUKAT objectbase. Its syntax is 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, making objects persistent, and so on. The description of every language is given in the subsequent sections, while the full syntax of the TIGUKAT language is described in Appendix B.

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 bar '|' separates alternatives. The square brackets '[', ']' enclose optional material which consists of one or more items separated by vertical bars.

6.1 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 declaration, collection declaration, class declaration, behavior manipulation, function declaration, and association. Statements in TIGUKAT language are separated by a semicolon.

The type declaration statement is used to create new TYPE Objects in a TIGUKAT objectbase. The general syntax of this statement is as follows:

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

The *create type clause* declares a reference to a new type. The *under clause* contains a type list which declares all direct supertypes of a new type. This list is composed of comma-separated constant references to existing types. It cannot be empty, as every type in TIGUKAT is at least a subtype of `T_Object` type. The last part of type declaration statement is the behavior list which is made up of zero or more behavior signatures separated by commas. The behavior signature has the following syntax:

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

A behavior signature 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. Thus, the whole *behavior list* is of the form:

< *behavior list* >: [< *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 can be declared either as a **public** behavior, or as 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. 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 the native behaviors can be removed from them. The general syntax of these statements is as follows:

< *behavior statements* >: **add to** < *type reference* > < *behavior list* >
| **remove from** < *type reference* > **behaviors:** < *name list* >

The first statement adds new behaviors to an existing type. It consists of a behavior list which has the same format as the behavior list in the type declaration statement. Thus, behaviors must be declared public or private, and all behavior names must be unique within a given type and all its supertypes. The second component of the *add* statement is a type reference which declares the type with which new behaviors are to be associated . The *remove* statement deletes behaviors from a given type. However, only the native behaviors can be removed from a given type. The behaviors are automatically removed from all the subtypes of this type. Again, the behavior list has the same syntax as in the type declaration statement, and the type reference is a reference to an existing type from which the behaviors are to be removed. 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 TIGUKAT objectbase has to 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 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 the 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 functions which already exist in the objectbase. The function signature specifies the semantics of the function, and the function object with this semantics is bound to the local reference. Thus, in both function declaration statements, a function signature must be declared. A function signature has the following format:

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

A *function name* in the function signature specifies a unique name of the function, and it also becomes a reference to the function object. The formal parameter list has the following syntax:

< *formal parameter list* >: [< *first parameter* >] < *parameter list* >
< *first parameter* >: **self :** < *type reference* > [,]

The first parameter, which is optional, declares a type of the receiver object. In other words, it defines a type with which the function can be associated. If it is not specified, `T_Object` primitive type is assumed by default. The second part of the formal parameter list declares parameters and their types in the function. The syntax of a parameter declaration in this list is as follows:

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

The parameter declarations in this list are separated by a comma. In TQL functions, parameters play the role of constants in TQL statements. The last part of the function signature is a type reference specified after a colon. It defines a result type of a new function. In the following example, two computed functions *age* and *spouse* are declared:

```
C++ function f_age(self:T_person): T_integer
   begin
       T_date today();
       today.initDate();
       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 `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 bounds 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 who must resolve it.

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

< *association* >: **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* > **with** < *function reference* >

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

< *get association* >: < *behavior reference list* > **with** *GET*

< *set association* >: < *behavior reference list* > **with** *SET*

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

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 the given type. Every element of this list consists of the *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 two different semantics of behaviors which can be associated with stored functions. The semantics of behaviors can either be 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, within the same association statement all *set* and *get* association correspond to the same stored function. Thus, at most one *get, set* pair is created within a single association statement. In order to associate behaviors in the 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. It is incorrect, as behaviors have different semantics and should not be associated with the same stored function.

associate in `T_person`
getName **with** *GET*, *setName* **with** *SET*,
getBrtday **with** *GET*, *setBrtday* **with** *SET*;

The example below illustrate 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`
getName **with** *GET*, *setName* **with** *SET*;

associate in `T_person`
getBrtday **with** *GET*, *setBrtday* **with** *SET*,
age **with** *f_age*, *spouse* **with** *f_spouse*;

The first association statement creates a pair of stored functions. 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 type `T_person`. The second statement creates a new pair of stored functions, and associate for behaviors *B_getBrtday* and *B_setBrtday* with *GET* and *SET* respectively. The 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 existing types. The association is done automatically during class creation. When the 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 and a request to create a class for this type is posted. The general syntax of the class declaration statement is as follows:

< *class declaration* >: **create class** [< *new reference* >] **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 reference of the corresponding type becomes a reference to a new class object. Although, a direct reference to a given type object is lost, it can still be accessed through the *B_Mapsto* behavior defined on the primitive `T_Class` type. Consider the following example:

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 `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. The reference of the type object becomes a reference to the class object. Although, the direct reference to the type object is lost, it can still be accessed through the *B_Mapsto* behavior defined on `T_Class`.

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* >: **create collection** < *new reference* >
type < *type reference* >
[**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.

Summarizing, 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 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 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. Association between behaviors and functions is defined by the association statement. 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. The example in Chapter 8 illustrates the whole process of creating new type, class, behavior and function objects and defining associations among them.

6.2 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.

6.2.1 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, in the *from clause* if the result of the application of behaviors is a finite collection, and in the *where clause* as predicates. The object equality is defined on the primitive type `T_object`, thus, 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 `T_collection` primitive type. They can be applied to any collection including those returned as a result of a query.

6.2.2 The Syntax of TIGUKAT Query Language

There are four basic TQL operations: **select, insert, delete**, and **update**, and 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 the 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 as follows:

< *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 form of simple variables, path expressions (which are equivalent to Bspecs defined in Chapter 3, Chapter 5), 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 a 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 TIGUKAT language supports the assignment statement, the result of a query can be directly assigned to a variable reference, and 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 the 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. We 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:

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

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 not specified a deep extend is assumed by default. In case of collections, a 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:

$$
\begin{aligned}
&< \textit{boolean formula} >: < \textit{atom} > \\
&\qquad\qquad\qquad \mid \ \textbf{not} < \textit{boolean formula} > \\
&\qquad\qquad\qquad \mid \ < \textit{boolean formula} > \textbf{ and } < \textit{boolean formula} > \\
&\qquad\qquad\qquad \mid \ < \textit{boolean formula} > \textbf{ or } < \textit{boolean formula} > \\
&\qquad\qquad\qquad \mid (< \textit{boolean formula} >) \\
&\qquad\qquad\qquad \mid \ < \textit{exists predicate} > \\
&\qquad\qquad\qquad \mid \ < \textit{forAll predicate} > \\
&\qquad\qquad\qquad \mid \ < \textit{boolean path expression} >
\end{aligned}
$$

Atom in the TQL boolean formula is one of the following:

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

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 the existential and universal quantifications. The existential quantifier is expressed by the *exists predicate* which is of the following format:

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

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 the 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 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 6.1

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

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 in given in Section 6.2.3). $\diamond$

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:

$$< object\ variable > \quad : \quad [(\ < cast\ type >\ )]\ < term > \\ \quad | \quad < index\ variable >$$

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 it proceeds. It acts as a behavior 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 is is not given, the type of the result collection is inferred from types of collections defined as ranges. The second definition of the object variable is the index variable. It has the following format:

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

The role of the index variables 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 the 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**. Similar to a select statement, they operate on the collection of objects and always return new collections as result. The syntax of these statements is as follows:

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

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

6.2.3 The Formal Semantics of TQL

The semantics of TQL is defined in terms of the object calculus. It is shown in this section that every TQL statement corresponds to the 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 statements of the form:

select $p_1,\ p_2,\ ...\ p_k$
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 as $S(p_1, ..., p_k)$. In other words, queries can be modeled as functions $S(p_1, ..., p_k)$ which operate upon one or more collections, and return collections as results. Lists which are returned in the result collections are made up of objects referenced by $p_1, ..., p_k$, and they are 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 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 collection, or implicitly in form of subqueries. If the range variable is defined over a constant collection reference, then it corresponds to the **range atom** (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 into an equivalent flat query.

Theorem 6.1 Every TQL query $S_p(p_1, ..., p_k)$ with nested queries in the *from clause* can be rewritten into 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:

$$\begin{aligned} \exists p_1 ... \exists p_k\ \ & (P_1(p_1) \wedge ... \wedge P_i(p_i) \wedge \\ & p_{i+1}\ \textbf{in}\ S_{i+1}(q_{i+1}) \wedge ... \wedge p_k\ \textbf{in}\ S_k(q_k) \wedge \exists r(R(r) \wedge F(p_1, ..., p_k, r))) \end{aligned} \tag{6.1}$$

$P_1, ..., P_i$ 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:

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

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:

$$\begin{aligned} & p_{i+j}\ \textbf{in}\ S_{i+j}(q_{i+j}) \equiv \\ & \quad \exists q_{i+j}(Q_{i+j}(q_{i+j}) \wedge \exists r_{i+j}(R_{i+j}(r_{i+j}) \wedge F_{i+j}(q_{i+j}, r_{i+j})) \wedge\ p_{i+j} = q_{i+j}) \end{aligned} \tag{6.2}$$

[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.

In 6.2, 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 6.1 the following equivalent formula is obtain:

$$\begin{aligned} &\exists p_1 ... \exists p_k (P_1(p_1) \wedge ... \wedge P_i(p_i) \wedge \\ &\quad (Q_{i+1}(p_{i+1}) \wedge \exists r_{i+1}(R_{i+1}(r_{i+1}) \wedge F_{i+1}(p_{i+1}, r_{i+1})) \wedge ... \wedge \\ &\quad (Q_k(p_k) \wedge \exists r_k(R_k(r_k) \wedge F_k(p_k, r_k))) \wedge \exists r(R(r) \wedge F(p_1, ..., p_k, r))) \end{aligned} \tag{6.3}$$

The formula 6.3 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:

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

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

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

□

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 6.2

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 of the following 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 object calculus, and build the second part of the calculus expression (b). 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 a range of the variable p is a constant reference, while the range of q is given in form of a subquery (S_p) which corresponds to the following 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 the following:

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

The query can be rewritten in TQL as:

$$\begin{array}{l}\textbf{select } \underbrace{p}_{a} \\ \textbf{from } \underbrace{p \textbf{ in } \#P, v \textbf{ in } \#V, w \textbf{ in } \#W}_{b} \\ \textbf{where } \underbrace{F_1(p,v,w) \textbf{ and } F_2(p,v)}_{c}\end{array}$$

$\diamond$

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

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

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

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

$< atom >$: $< term > = < term >$
| $< term\ list >$ **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 the **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*:

$< exists\ predicate >$: **exists** $< collection\ reference >$

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 a 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 the equivalent flat query. We decided to include it in the language, so users are not forced write queries in the prenex normal form. Consider the example:

Example 6.3

$S(p) \equiv$ **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 to:

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

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 a general case, this predicate is as follows:

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

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 the boolean formula is the *boolean path expression*. In general, path expressions in TIGUKAT language correspond to Bspecs defined in Chapter 5. Boolean path expressions are Bspecs which evaluate to objects of `T_boolean` type. A boolean formula which is given in 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 of the 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 formulas in object calculus. Thus, every TQL boolean formula is equivalent to an object calculus well-formed formula. $\square$

As shown in Section 6.2.2, 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, object calculus allows constant, 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, while path expression in TQL corresponds to 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 proceeded by a cast type which extracts (generalizes) behaviors from an object type. Thus, TQL cast type and the following term correspond to **index variable** in the object calculus as well.

Example 6.4

`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_person**(p) $\wedge$ $F(p)$)

which corresponds to the following calculus expression:

$\{p[B\_name, B\_age] \mid$ **C_person**$(p) \wedge F(p)\}$

$\diamond$

Theorem 6.3 Every select statement in TQL has an equivalent object calculus expression.

Proof: It follows directly from Theorem 6.1 and Theorem 6.2. Every select statement can be expressed as:

select $p_1,\ p_2,\ ...\ p_k$
from p_1 **in** $\#P_1,\ ..., p_k$ **in** $\#P_k, q_1$ **in** $\#Q_1, ..., q_n$ **in** $\#Q_n$
where $F(p_1, ..., p_k, q_1, ..., q_n)$

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

$$\{\ p_1, ..., p_k \mid P_1(p_1) \wedge\ ...\ \wedge P_k(p_k) \wedge \exists q_1, ..., \exists q_n(Q_1(q_1) \wedge\ ...\ \wedge Q_n(q_n) \wedge F(p_1, ..., p_k, q_1, ..., q_n))\ \}.$$

$\square$

Summarizing, the *select clause* of the select statement defines free variables of an object calculus formula, which correspond to variables of the target list in the object calculus expression. The *from clause* declares 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 is well defined.

Theorem 6.4 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 object calculus they are expressed by simple calculus expressions of the following form respectively: $\{ o \mid P(o) \vee Q(o)\}$, $\{ o \mid P(o) \wedge \neg Q(o)\}$, $\{ o \mid P(o) \wedge Q(o)\}$, where P is a reference to the first collection in the binary statement, and Q is a reference to the second collection. □

Theorem 6.5 The reduction from TQL to the object calculus is complete.

Proof: It follows directly from Theorems 6.1, 6.2, 6.3 and 6.4. □

6.3 TIGUKAT Control Language

The last part of 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, closed and so on. 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_environment*. Every TIGUKAT objectbase, has at least one instance of the **C_session** class which is referred to as a *root session*. When TIGUKAT is called, a *root session* is opened. All other sessions can be opened from this session. 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 as follows:

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

The session reference is a reference to a session object in the objectbase. If the session referenced by the *session reference* does not exists, a new session object is created, and it is opened.

The *save session* statement is used to save the session environment. The general syntax of this statement is as follows:

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

All transient objects are saved meaning that their references are stored in the session symbol table (they do not became persistent however). Next time that session object is opened, the environment is restored. Otherwise, once 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 as follows:

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

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 was saved, next time this session object is open, 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 as follows:

$$\begin{array}{lll} < \textit{make persistent} > : & \textbf{persistent} < \textit{object reference list} > \\ & | & \textbf{persistent all} < \textit{collection reference} > \end{array}$$

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 a *quit statement* which is used to quit the session without saving, and leave the TIGUKAT objectbase. The syntax of this statement is as follows:

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

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.

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

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

where the left side is:

$$< \textit{left side} >: < \textit{object reference} >$$

and the right side can be one of two things:

$$\begin{array}{ll} < \textit{right side} >: & < \textit{TQL Statement} > \\ & | < \textit{path expression} > \end{array}$$

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

Chapter 7

The Object Algebra

An algebraic expression represents a typed collection of objects. The operands and result of algebraic operators are typed collections. Collections can be heterogeneous. When combining collections with certain algebra operators (e.g., product, union, intersection), a collection with a different type from those of the operand collections (or any type in the lattice) may be created. Thus, in order to integrate these new types into the existing lattice a type inferencing mechanism is introduced and used by the algebra.

There are two types to consider here: the type of the container (i.e., the type of the collection object) and the type of the objects in the container (i.e., the membership type of the collection). The types we are referring to in our inferencing mechanism are the membership types of collections.

7.1 Semantics of Type Inferencing

Type creation and type inferencing are topics related to schema evolution. In this section, we do not give an exhaustive discussion of schema evolution. However, in order to fully appreciate the semantics of the target-creating algebra we present the intuition of the type creation and inferencing semantics developed for this model. The full description of our schema evolution semantics will appear in a forthcoming paper.

Let T_i $(1 \leq i \leq n)$ denote types. Then the behavioral application $T_i.B\_interface$ denotes the collection of behaviors applicable to objects of type T_i. The type inferencing mechanism is based on type construction operations that are modeled as behaviors on the primitive type `T_type`. They are defined as follows:

$T_1 \sqcap T_2$ (*B_tmeet*) produces the *meet* type of the argument types. The result type, say T, defines the behaviors that are common to types T_1 and T_2. The interface set of T is defined as $T_1.B\_interface \cap T_2.B\_interface$. If T_2 is a subtype of T_1, then $T_1 \sqcap T_2$ is T_1. The converse is true if T_1 is a subtype of T_2. The *B_tmeet* behavior produces a result type that is integrated into the type lattice as a supertype of the argument types.

$T_1 \sqcup T_2$ (*B_tjoin*) produces the *join* type of the argument types. The result type, say T, defines all the behaviors of T_1 together with all the behaviors of T_2. The interface set of T is defined as $T_1.B\_interface \cup T_2.B\_interface$. If T_2 is a subtype of T_1, then $T_1 \sqcup T_2$ is T_2. The converse is true if T_1 is a subtype of T_2. The *B_tjoin* behavior produces a result type that is integrated as a subtype of the argument types.

$T_1 \otimes T_2$ (*B_tproduct*) produces the *product* type of the two argument types. The result type, say T, defines product behaviors (see below) and is integrated as a subtype of other product types according to the product behaviors defined. That is, the name and result type of product behaviors determines subtyping on product types. Objects of type T are pairs with the first component being an object of type T_1 and the second component an object of type T_2. The *B_tproduct* behavior produces a product of types that does not have a sub/supertype relationship with the argument types, but is integrated with other product types. Instances of a product type are called *product objects*. They are created from objects in the extents of the types that contributed to the product type. The components of a product object are the original objects from which it was created.

The binary $\sqcap, \sqcup, \otimes$ behaviors can be naturally extended by defining them over multiple types in the following way (where $n \geq 2$):

$$\begin{aligned} \sqcap_{i=1}^{n} T_i &\equiv T_1 \sqcap T_2 \sqcap \cdots \sqcap T_n \\ \sqcup_{i=1}^{n} T_i &\equiv T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n \\ \otimes_{i=1}^{n} T_i &\equiv T_1 \otimes T_2 \otimes \cdots \otimes T_n \end{aligned}$$

Parentheses may be used with the above operators. Each parenthesized subexpression represents the creation of a new type. With respect to the behaviors defined on the final type created, operators $\sqcap$ and $\sqcup$ are commutative and associative while $\otimes$ is neither. Parentheses affect the semantics of the product operator in the following way. Product types define inject behaviors (ρ_i) that return the i^{th} component of a product object. With this in mind, the following product types are all different types that define different inject behaviors with different result types:

$$\begin{gathered} (T_1 \otimes T_2) \otimes T_3 \\ T_1 \otimes (T_2 \otimes T_3) \\ T_1 \otimes T_2 \otimes T_3 \end{gathered}$$

The first type defines two inject behaviors; ρ_1 that returns a product object of type $T_1 \otimes T_2$ and ρ_2 that returns an object of type T_3. The second one defines two inject behaviors that differ from the first; ρ_1 that returns an object of type T_1 and ρ_2 that returns a product object of type $T_2 \otimes T_3$. The third type defines three inject behaviors; ρ_1 that returns an object of type T_1, ρ_2 that returns an object of type T_2 and ρ_3 that returns an object of type T_3.

The definition and integration of product types into the existing lattice and the creation of product objects is designed to be an automated process. A request is made through the application of a behavior to create a product object from a given list of objects. This may spawn the creation of a new product type and a class for the object if they don't already exist. In order to support these semantics, the following extensions are made to the primitive type system:

- We define `T_product` as a subtype of `T_type`. `T_product` defines the native behavior

  $$B\_compTypes : \texttt{T\_list}\langle\texttt{T\_type}\rangle$$

  which returns the list of component types that make up a product type. Intuitively, `T_product` is the type that describes the semantics of product types. The class **C_product** for this type is created as an instance of `T_type-class` so that the primitive type creation behavior (defined as *new* on this type) can be applied to it and passed a list of component types. The semantics of applying this creation behavior to **C_product** with a list of argument types is to create a

product type (if one doesn't already exist) whose component types are the argument types passed and to integrate the new type with existing product types. The behavior *B_tproduct* ($\otimes$) applies the type creation behavior to **C_product** passing along its arguments types. This defines the creation of new product types as instances of **C_product**.

- We define type `T_product-class` as a subtype of `T_class`. A product object creation behavior

$$B\_new : \texttt{T\_list}\langle\texttt{T\_object}\rangle \rightarrow \texttt{T\_object}$$

is defined on `T_product-class`. Intuitively, this type defines the semantics for the classes of product types. The class **C_product-type** is created as an instance of **C_class-class**. The type `T_class-class` defines a class creation behavior (*new*) that accepts a type (the type to associate a class to) as an argument. By applying this behavior to **C_product-class** and passing a product type, a class for the product type is created (if one does not already exist). Now, product objects can be created through the resulting class by applying the *B_new* behavior defined on `T_product-class` to the class and passing a list of objects.

For example, the following series of behavioral applications create a new product type called `T_person-dwelling`, a product class called **C_person-dwelling** and a product object *o* as an instance of this class. The first component of *o* is the person object `joe` and second component is the dwelling object `apt204`. The "$\leftarrow$" symbol denotes assignment and <> denotes a list of objects.

`T_person-dwelling`	$\leftarrow$	**C_product**.*B_new*(<`T_person`, `T_dwelling`>)
C_person-dwelling	$\leftarrow$	**C_product-class**.*B_new*(`T_person-dwelling`)
o	$\leftarrow$	**C_person-dwelling**.*B_new*(<`joe`, `apt204`>)

Finally, we define a behavior *B_newprod* on `T_object` that accepts as arguments a list of objects and a list of corresponding behavioral projection sets. The result of applying this behavior with these arguments is as follows:

1. A product type is created (if one does not already exist) using the type of the receiver object and the types of the objects in the first argument list. The types are projected over the behavioral projections in the second argument list before the product type is formed.
2. A class for the product type is created (if one does not already exist).
3. A product object formed from the receiver and the objects in the first argument list is created as an instance of the (possibly new) product type and a reference to this object is returned.

For a given list of objects $o_1, o_2, \ldots, o_n$ and list of behavioral projection sets $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n$, we use the notation $newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n])$ to denote a Bspec that represents the application of the product creating behavior with the given argument lists as:

$$o_1.B\_newprod(<o_2, \ldots, o_n>, <\mathcal{B}_1, \ldots, \mathcal{B}_n>)$$

The result is a product object $(o_1, \ldots, o_n)$ whose i^{th} component is the original object o_i from which it was formed. The type of each o_i component object in the product type is the type of the original o_i object projected over the behaviors in $\mathcal{B}_i$. When the behavioral projection list is immaterial, we simplify the notation to $newprod(o_1, \ldots, o_n)$.

In order to extract and operate on the original component objects of a product object, every product type defines an inject behavior for each of its component types. Product types are integrated into the type lattice according to the names and return types of these behaviors (more generally, their semantics). The behaviors defined on product types are the following:

Inject: For every product type $T_1 \otimes \cdots \otimes T_n$, there are n inject behaviors defined ρ_i, $1 \leq i \leq n$ such that for a given object of this type, say o, the behavioral application $o.\rho_i$ returns the object of type T_i that represents the i^{th} component of o.

A product type $T_1 \otimes \cdots \otimes T_n$ is integrated as a subtype of a product type $T'_1 \otimes \cdots \otimes T'_m$ if $m \leq n$ and T_i is a subtype of T'_i for $1 \leq i \leq m$. It is integrated as a supertype of $T''_1 \otimes \cdots \otimes T''_k$ if $n \leq k$ and T_i is a supertype of T''_i for $1 \leq i \leq n$. If the product type cannot be integrated as a subtype of some other type, it is defined as a subtype of `T_object`.

Equality: The object equality behavior for `T_product` is refined to be based on pairwise identity equality of the component objects. That is, for two product objects o and o' of types $T_1 \otimes \cdots \otimes T_n$ and $T'_1 \otimes \cdots \otimes T'_n$, $o = o'$ is true if and only if $o.\rho_i = o'.\rho_i$ for $1 \leq i \leq n$.

7.2 Algebra Expressions

The underlying framework of the object algebra and calculus are essentially the same. However, an important difference is that the algebra can be viewed as having a functional basis as opposed to the logical foundation of the calculus. This perspective was described by Backus [Bac78] and has been exploited by several complex object models [MD86, Day89, AB93]. In our algebra, names are used as placeholders for collections with the appropriate types. The predicates $=, \neq, \in, \notin$ and connectives $\wedge, \vee, \neg$ are handled as boolean-valued functions. The object creating behaviors *newcoll*( ) and *newprod*( ) are variadic functions. There is a small set of well-defined algebraic operators (viewed as functions) that provide meaningful iterations over collections and can be composed to form more complicated queries (existential and universal quantification are handled by composing these operators). Thus, an algebraic query is a functional expression to be evaluated and the algebra is a functional language.

The basic algebra expression consists of a single collection specification. In our algebra, a base algebra expression is either a collection name or the function application $newcoll(c_1, \ldots, c_n)$ where each c_i denotes a constant (i.e., a ground term). We call the latter a collection constant. Other algebra expressions can be constructed from the base expressions using the algebraic operators.

The basic constructs of the calculus (object constants, object variables and Bspecs) have a functional interpretation that abstracts over the free variables in the constructs. We call this interpretation of the construct a *functional expression*.

Definition 7.1 *Functional Expression:* A *functional expression* is a functional abstraction of an object constant, an object variable or a Bspec defined as follows:

1. For every constant c, there is a unary functional expression $\lambda x.c$ that returns the constant c.
2. For every variable x, there is a unary functional expression $\lambda x.x$ that is the identity function.
3. For every Bpsec $\beta\{\vec{x}\}$, there is a functional expression $\lambda\vec{x}.\beta\{\vec{x}\}$ that represents a functional abstraction of the Bspec. If the Bspec is a ground term (i.e., is not free over any variables), then its functional expression is $\lambda x.\beta\{\}$ with the same semantics as for constants.

The variables appearing after the λ symbol and before the first dot are called the *parameters* of the functional expression.

Since Bspecs can be abstracted into functional expressions, all behaviors have this abstraction. This means that predicates $=, \neq, \in, \notin$ and connectives $\wedge, \vee, \neg$ are boolean-valued functional expressions. The object creating behaviors *newcoll*( ) and *newprod*( ) are variadic functional expressions that produce the appropriate collection or product object. The algebraic operators (defined below) are functional expressions that operate on collections and produce collections as results.

In general, we use *mop* to denote a functional expression and call it a *mop function*. Given a mop function (*mop*) with parameters $\vec{x}$ and given objects $\vec{o}$ that are type compatible with $\vec{x}$, we use $mop(\vec{o})$ to denote the application of the mop function to the objects. That is, each o_i is substituted for an x_i to form a context, the context is evaluated and the result object is produced.

Operands and results of the object algebra operators are typed collections of objects. Thus, the algebra is *closed* since the result of any operator may be used as the operand of another. Let Φ represent an operator in the algebra. The notation $P\ \Phi\ \langle Q_1, \ldots, Q_n \rangle$ is used for expressions where P and each Q_j are names for typed collections of objects. They represent the arguments to Φ. When $n = 1$ we use $P\ \Phi\ Q$ and when $n = 0$ we use $P\ \Phi$ without loss of generality. The collections represented by P and Q_j may be names for base collections, a collection constant creation request or the result of an algebraic subexpression. Since the model supports substitutability, any specialization of collection, including classes, may be used as the operand. Similar to the range predicates of the calculus, we define P^+ to denote the shallow extent when P is the name for a class.

Certain algebraic operators require a functional expression (mop function) as an argument. The operator applies the mop function to permutations of elements from its operand collections and takes appropriate action on the result. Some operators require a boolean-valued functional expression (a predicate) denoted F. Evaluating F for particular permutation of arguments produces a boolean result upon which the operator takes an appropriate action. The membership types of the operand collections must be consistent with the types expected by the mop function. Mop function qualified operators are written as $P\ \Phi_{mop}\ \langle Q_1, \ldots, Q_n \rangle$ where *mop* is a mop function (or predicate) with parameters, say $p, q_1, \ldots, q_n$, that range over the elements of collections $P, Q_1, \ldots, Q_n$, respectively. To make the identification of arguments with parameters simplier and more explicit in algebraic operators, we sometimes drop the $\lambda\vec{x}$ specification from mop functions and subscript operand collections with the parameters of the mop function as P_p. This explicitly indicates that the range of variable p (in the mop function) are the elements of the operand collection P. For example, $P_p\ \Phi_{mop(p,q)}\ Q_q$ is used instead of the abstract notation $P\ \Phi_{\lambda p,q.mop(p,q)}\ Q$. For operands consisting of product objects with components $\vec{x}$, we subscript the operands with all the components as $P_{\vec{x}}$. This means that some combination of inject behaviors on the elements of P will retrieve the original x_i components. This is only a notational convenience to identify the ranges of variables and the components of product objects in algebra expressions.

For a collection P, the notation Λ_P denotes the membership type of the objects in P. Furthermore, the behavioral application $\Lambda_P.B\_interface$ denotes the behaviors applicable to objects of this type. We use this notation and the results of Section 7.1 to infer a new membership type for the result collection produced by the operators.

The object algebra defines both *target-preserving* and *target-creating* operators. The target-preserving operators are as follows:

Difference (denoted $P - Q$): Difference is a binary operator that produces a collection containing objects that are in P and not in Q. The membership type of the result collection is exactly the type of P (i.e. Λ_P).

Union (denoted $P \cup Q$): Union is a binary operator that produces a collection containing objects that are in P, in Q or in both. The membership type of the result collection is $\Lambda_P \sqcap \Lambda_Q$. This type defines behaviors common to both Λ_P and Λ_Q.

Intersection (denoted $P \cap Q$): Intersection is a binary operator that produces a collection containing objects that are both in P and in Q. The membership type of the result collection is $\Lambda_P \sqcup \Lambda_Q$. This type defines all behaviors of both Λ_P and Λ_Q. Note that $P \cap Q$ is derivable from difference as $P - (P - Q)$ or $Q - (Q - P)$. Even though these three operations produce result collections with identical extents, the membership type of each result may differ.

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

$$P \Downarrow \equiv \bigcup \{x \mid x \in P\}$$

The membership type of the result collection is the extended meet over the membership types of the collections in P.

$$\sqcap \{\Lambda_x \mid x \in P\}$$

Select (denoted $P\ \sigma_F\ \langle Q_1, \ldots, Q_n \rangle$): where F is a predicate over the elements of collections $P, Q_1, \ldots, Q_n$, meaning F expects arguments $p, q_1, \ldots, q_n$ and that they are type consistent with the membership types of the collections. Select is a higher order operation accepting a mop function, the predicate F, and the $n+1$ collections $P, Q_1, \ldots, Q_n$ as arguments. The select operation produces a collection containing objects from P corresponding to the p component of each permutation $<p, q_1, \ldots, q_n>$ that satisfies $F(p, q_1, \ldots, q_n)$. The membership type of the result collection is exactly the type of P (i.e. Λ_P).

Example 7.1 Return the persons that are senior citizens:

$$\textbf{C\_person}_p\ \sigma_{p.B\_age \geq 65}$$

Example 7.2 Return the maps that contain water zones:

$$\textbf{C\_map}_p\ \sigma_{q \in p.B\_zones}\ \textbf{C\_water}_q$$

Project (denoted $P\ \Pi_{\mathcal{B}}$): where $\mathcal{B}$ is a behavioral projection set with the restriction that it be a subset of the behaviors defined by the membership type of P. (i.e., a subset of $\Lambda_P.B\_interface$). The $\mathcal{B}$ collection is automatically unioned with the behaviors of type `T_object` before the project is performed in order to ensure consistency with the object model (i.e., everything is an object and therefore must support the behaviors of `T_object`). Project produces a collection containing the objects of P, but with a membership type coinciding with the behaviors in $\mathcal{B}$.

The new type is integrated into the sublattice rooted at `T_object` and with the base Λ_P. An abstract type definition is created that has all the behaviors defined by $\mathcal{B}$. The implementations of these behaviors are undefined, but this doesn't cause problems because no class is created and therefore no objects of this type exist. This new type has no special properties, meaning it can be subtyped, implementations for its behaviors can be defined, a class can be associated with it and objects of this type can be created.

The $\mathcal{B}$ projection set has no impact on which objects appear in the result collection of the query. It is only important during the final type assignment that occurs at type inferencing time after the extent of the query has been produced. This form of project differs from the traditional one in that it does not project over the structure of objects, but rather over their behavioral specification. Our project is a behavioral-theoretic notion of projection that has no structural implications.

Example 7.3 Project over behaviors *B_name* and *B_age* for class **C_person**:

$$\textbf{C\_person}\ \Pi_{B\_name,B\_age}$$

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

Product (denoted $Q_1 \times \cdots \times Q_n$): where $n \geq 2$. Product produces a collection containing product objects of the form $(q_1, q_2, \ldots, q_n)$ created from each permutation $<q_1, q_2, \ldots, q_n>$ such that component q_i is an object from Q_i. Product may initiate the creation of a new type along with a new class to maintain the product objects. The membership type of the result collection is $\Lambda_{Q_1} \otimes \cdots \otimes \Lambda_{Q_n}$. Although this operator seems structural in nature, Section 7.1 defines a behavioral-theoretic notion of product that is commensurate with the uniformity of the object model.

There is an additional operator that fits into both the target-preserving and target-creating classification. The **map** operator produces a collection of new or existing objects depending on the mop function argument passed to it. That is, if the mop function is target-creating, the operator is target-creating, otherwise it is target-preserving. Map is defined as follows:

Map (denoted $Q_1 \gg_{mop} \langle Q_2, \ldots, Q_n \rangle$): where *mop* is a mop function over the elements of collections $Q_1, Q_2, \ldots, Q_n$, meaning it expects arguments $q_1, q_2, \ldots, q_n$ and that they are type consistent with the membership types of the collections. Map is a higher order operation accepting the mop function *mop* and the n collections $Q_1, Q_2, \ldots, Q_n$ as arguments. For each permutation of objects $<q_1, q_2, \ldots, q_n>$ formed from the elements of the argument collections, $mop(q_1, q_2, \ldots, q_n)$ is applied and the resulting object is included in the result collection. The membership type of the result collection is the type of the mop function. Map is a generalized version of the same operator defined in [SÖ90a] and is similar to the *replace* restructuring operator in [AB93]. However, *replace* operates over a single set-valued relation in contrast to ours which is variadic over the number of argument collections. Map is also similar to the **image** operator of [SZ90] except that theirs is restricted to the application of single behaviors while the *mop* in our approach is a general functional expression.

Example 7.4 Return the zones that have people living in them:

$$\textbf{C\_person}_p \gg_{p.B\_residence.B\_inZone}$$

Example 7.5 Return the proximities of water zones to the City of Edmonton:

$$\textbf{C\_water}_p \gg_{p.B\_proximity(\mathsf{edmonton})}$$

Example 7.6 Return (person, person, children) triples for all combinations of people:

$$\mathbf{C\_person}_p \gg_{newprod(p,q,p.B\_children(q))} \mathbf{C\_person}_q$$

The operators defined above form the *primitive* algebra (some refer to this as a *physical* algebra). They are fundamental in supporting the expressive power of the calculus and the subsequent operators can be defined in terms of them. We add the following operators to the primitive algebra and call this the *extended* algebra (some call this a *logical* algebra). These operators are derived from the primitive algebra, they support a useful functionality, they generalize the expressive power of the algebra and some are important for higher-level optimizations [SÖ90a]. Note that the following operators are target-creating.

Join (denoted $P \bowtie_F \langle Q_1, \ldots, Q_n \rangle$): where $n \geq 1$ and F is a predicate over the elements of collections $P, Q_1, \ldots, Q_n$. Join produces a collection containing product objects of the form $(p, q_1, \ldots, q_n)$ created from each permutation $< p, q_1, \ldots, q_n >$ that satisfies $F(p, q_1, \ldots, q_n)$. The membership type of the result collection is $\Lambda_P \otimes \Lambda_{Q_1} \otimes \cdots \otimes \Lambda_{Q_n}$. This type and its associated class may be created if they don't already exist.

The join operator can be expressed in terms of product and selection as follows:

$$E_{x_1} \bowtie_F \langle E_{x_2}, \ldots, E_{x_n} \rangle \quad \equiv \quad (E_{x_1} \times E_{x_2} \times \cdots \times E_{x_n})_o \; \sigma_{F'}$$

where F is a predicate over variables $\vec{x}$ and F' is F except that every occurrence of x_i is replaced with $o.\rho_i$, the inject of component x_i from product object o.

Example 7.7 Return married couples that don't live together:

$$\mathbf{C\_person}_p \bowtie_{p.B\_spouse=q \;\wedge\; q.B\_residence \neq p.B\_residence} \mathbf{C\_person}_q$$

Example 7.8 Return (map, water zone, water zone) triples where the given map contains two different water zone that are within 100 units from each other:

$$\mathbf{C\_map}_m \bowtie_{x \in m.B\_zones \;\wedge\; y \in m.B\_zones \;\wedge\; x \neq y \;\wedge\; x.B\_proximity(y) \leq 100} \langle \mathbf{C\_water}_x, \mathbf{C\_water}_y \rangle$$

Generate Join (denoted $Q_1 \; \gamma_g^o \; \langle Q_2, \ldots, Q_n \rangle$): g is a generating atom of the form $o \; \theta \; mop$ where θ is either $=$ or $\in$ and mop is a mop function over the elements of collections $Q_1, Q_2, \ldots, Q_n$. 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 $(q_1, q_2, \ldots, q_n, mop(q_1, q_2, \ldots, q_n))$ for each permutation of the q_i's (i.e., each product object is a permutation of the q_i's extended by the result of applying the mop function to that permutation). If θ is $\in$, the result contains product objects of the form $(q_1, q_2, \ldots, q_n, o)$ for each permutation of the q_i's and each $o \in mop(q_1, q_2, \ldots, q_n)$ (i.e., for a permutation of the q_i's and for each member o of the collection resulting from the application $mop(q_1, q_2, \ldots, q_n)$, a product object with components $(q_1, q_2, \ldots, q_n, o)$ is created as a member of the result collection). Generate Join is similar to PDM's *apply-append* operator except theirs works on a single tuple while ours is over an arbitrary number of collections.

The equality atom based generate join can be expressed by map as follows:

$$E_{x_1} \; \gamma^o_{o=mop} \; \langle E_{x_2}, \ldots, E_{x_n} \rangle \quad \equiv \quad E_{x_1} \gg_{newprod(x_1, x_2, \ldots, x_n, mop(\vec{x}))} \; \langle E_{x_2}, \ldots, E_{x_n} \rangle$$

The membership atom based generate join can be expressed by the following series of algebraic operations:

$$\begin{aligned} A &\stackrel{\text{def}}{=} (E_{x_1} \times E_{x_2} \times \cdots \times E_{x_n})_x \gg_{newprod(x,mop(x.\rho_1,x.\rho_2,\ldots,x.\rho_n))} \\ B &\stackrel{\text{def}}{=} (A_x \gg_{newcoll(x.\rho_1)\times x.\rho_2}) \Downarrow \\ E_{x_1}\ \gamma^o_{o\in mop}\ \langle E_{x_2},\ldots,E_{x_n}\rangle &\equiv B_x \gg_{newprod(x.\rho_1.\rho_1,x.\rho_1.\rho_2,\ldots,x.\rho_1.\rho_n,x.\rho_2)} \end{aligned}$$

Example 7.9 Return (zone, proximity) pairs of each zone extended with its proximity to all water zones:

$$\mathbf{C\_zone}_p\ \gamma^o_{o=p.B\_proximity(q)}\ \mathbf{C\_water}_q$$

Example 7.10 Return (map, zone) pairs of each map extended with the zones contained in that map:

$$\mathbf{C\_map}_p\ \gamma^o_{o\in p.B\_zones}$$

Reduce (denoted $P\triangle_{\rho_i}$): where P is a collection of product objects and ρ_i is an inject behavior defined on the membership type of P. The reduce operator has the effect of discarding the i^{th} component of the product objects in P. That is, product objects of the form $(p_1,\ldots,p_{i-1},p_i,p_{i+1},\ldots,p_n)$ with inject behaviors $\rho_1,\ldots,\rho_{i-1},\rho_i,\rho_{i+1},\ldots,\rho_n$ are mapped to product objects $(p_1,\ldots,p_{i-1},p_{i+1},\ldots,p_n)$ with inject behaviors $\rho_1,\ldots,\rho_i,\rho_{i+1},\ldots,\rho_{n-1}$. This is similar to the relational projection operator except that the specified components are removed. If P is not a product object, the empty collection is returned.

The reduce operator can be expressed by map as follows:

$$E\triangle_{\rho_i} \equiv E_o \gg_{newprod(o.\rho_1,\ldots,o.\rho_{i-1},o.\rho_{i+1},\ldots,o.\rho_n)}$$

The effect of the map is to produce product objects that contain all the original components of o, minus the i^{th} component. Map, together with the product object creation behavior, is a generalization of the relational projection on product objects.

As a notational convenience, a series of reduce operators is coalesced into a single one and the ρ symbol is dropped from the specification. The equivalence is defined as follows:

$$P\triangle_{\rho_{x_1}}\ldots\triangle_{\rho_{x_n}} \equiv P\triangle_{x_1,\ldots,x_n}$$

Example 7.11 Let E be the result of Example 7.8 above. Reduce E by excluding the first water zone of the result:

$$E\triangle_x$$

The functional nature of queries is twofold. On the one hand, a query may be thought of as a function where collection names serve as variables representing the arguments. By associating these names with collections in an instantiation of an objectbase we get a substitution that can be evaluated. On the other hand, for a given objectbase a query denotes a constant. That is, a query is a function only when all possible objectbases are considered. For a given objectbase, a query is an expression resembling a 0-ary function. In contrast, behavioral compositions such as Bspecs (mops) are functions even within the instantiation of a objectbase. When they are composed with algebraic operators *select, map, join* and *generate join*, they denote functions that are applied to permutations of the elements from the operand collections.

We have chosen not to include a *powerset* operator in our algebra because one of our primary interests is to produce an efficient implementation of our query model. Use of powerset causes exponential growth of collections and the costs that this could incur is unacceptable for our implementation.

The foundations of powerset and recursive query capability are present in our model. One extension is the addition of a primitive *powerset* algebraic operator that accepts a collection and produces the powerset of the collection as output. Using this, we could derive a form of generate join that creates a collection of product objects, one for each element in the powerset of the *mop* function evaluation, whose components are the operand collections appended with the element from the powerset. Since we already have a *B_containedBy* behavior (analogous to $\subseteq$) defined on `T_collection`, we only need to define a predicate $s \subseteq t$ in the calculus for this behavior. If the term s is a variable, then this becomes another kind of generating atom in the calculus.

A clean definition of safety with respect to powerset that complies with the efficient translation of evaluable formulas (i.e., without forming a DOM domain) is not apparent. The powerset property has a logical derivation as follows:

$$\begin{aligned} s \subseteq t &\equiv \forall x(x \in s \rightarrow x \in t) \\ &\equiv \forall x(x \notin s \vee x \in t) \\ &\equiv \neg\exists x(x \in s \wedge x \notin t) \end{aligned}$$

This derivation does not satisfy the evaluable property unless s and t are further restricted outside the formula. This means that $s \subseteq t$ can not in general be used to generate objects for s from t and its only consistent use would be as a restriction atom. However, our language already handles this because the derivation is a valid formula of the calculus and is safe if s and t are restricted outside the formula. Thus, without being able to generate values for s from the derivation, no additional power is added by including a $\subseteq$ predicate and a powerset operator. On the contrary, it would make the algebra more expressive than the calculus since the translation of the powerset operator to the calculus (i.e., the derivation above) would result in an unsafe calculus formula.

A clean incorporation of powerset capability that complies with the feasible translation properties of the evaluable class is part of our future research. If a compatible derivation can be found, extending the proofs of completeness will be straightforward. From algebra to calculus it is simply a matter of stating the derivation of the powerset operator and from calculus to algebra it involves carrying the $\subseteq$ predicate through the translation.

7.3 Safety of Algebraic Expressions

Recall from the discussion in Section 5.2 that there are two forms of safety to consider. The first form checks the domain independence of the query and was defined in that section. The second form checks the safety of a query with respect to *operand finiteness*, meaning it checks that the query does not add objects to any collections or classes that it is ranging over. We define this check on algebraic expressions that checks the operand finiteness of each operator in the expression.

Since object creation and insertion occurs through the application of behaviors, the check for operand finiteness could be combined with an algebraic type checking mechanism such as the one defined in [SÖ90b] that goes through an algebraic expression and examines the behaviors being applied in algebraic operators for type consistency.

The "problematic" operators of the algebra that can violate operand finiteness by adding objects

to their operands are *select*, *map*, *join* and *generate join* because they contain mop functions that are general behavioral applications. The only side effect behaviors allowed in mop functions are insertion into a collection (i.e., *B_insert* on a collection) and creation of a new object (i.e., *B_new* on a class). We further restrict this in that the insertion or creation behavior must be applied to a constant reference of a collection or a class (i.e., not to a variable or the result of a behavioral application) or must not occur at all.

We assume that all other behaviors in a mop function are side-effect free (i.e., they do not create new objects or modify existing objects in any way). The reason for this assumption is that we don't examine the implementations of behaviors to determine their safety with respect to operand finiteness. The exceptions to this assumption are the primitive defined *newcoll*() and *newprod*() behaviors and the algebraic operators. They can occur in mop functions, but their use is restricted as defined below.

An algebraic expression is rejected if it contains an algebraic operator that is *unsafe* with respect to operand finiteness. An algebraic operator Φ is *unsafe* with respect to operand finiteness if it is a *select*, *map*, *join* or *generate join* operator which has a mop function that contains one of the following:

- an application of *B_new* on a class that is an operand of Φ,
- an application of *B_new* on a class that is a subclass of an operand of Φ and the operand is a class ranging over its deep extent,
- an application of *B_insert* on a collection that is an operand of Φ,
- an application of *newcoll*() and one of the operands of Φ is the class **C_collection**,
- an application of *newprod*() that creates an object in a class that is an operand of Φ,
- an application of *newprod*() that creates an object in a subclass of an operand of Φ and the operand is a class ranging over its deep extent,
- an algebraic operator and one of the operands of Φ is **C_collection**,
- an algebraic operator and this algebraic operator is *unsafe* with respect to operand finiteness.

Chapter 8

Example Objectbase

Object-orientation is intended to serve many application areas requiring advanced data representation and manipulation. A geographic information system (GIS) [Aro89, Tom90] is 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 applications 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 GIS can be defined as an application "designed for the collection, storage and analysis of objects and phenomena where geographic location is an important characteristic or critical for analysis... In each case, what it is and where it is must be taken into account." [Aro89]. Some examples include displaying the effective range of a police force, illustrating how logging activities affect wildlife populations, and depicting the severity of soil erosion.

GIS technology is being applied to many areas. Some common ones include agriculture and land use planning, forestry and wildlife management, geology, archaeology, municipal facilities management, and more global scale applications such as ecology. Each of these areas rely on statistical data, historical information, aerial photographs, and satellite images for analyzing and presenting empirical data, for drawing conclusions about certain phenomena, or for predicting future events through sophisticated computer simulations using the information at hand. GISs require advanced information management and analysis features in order to be effective. Object-oriented databases have the potential to provide this advanced functionality.

A type lattice for a simplified GIS is given in Figure 8.1. The example is sufficiently complex to illustrate the advanced functionality of the query model we present, yet simple enough to be

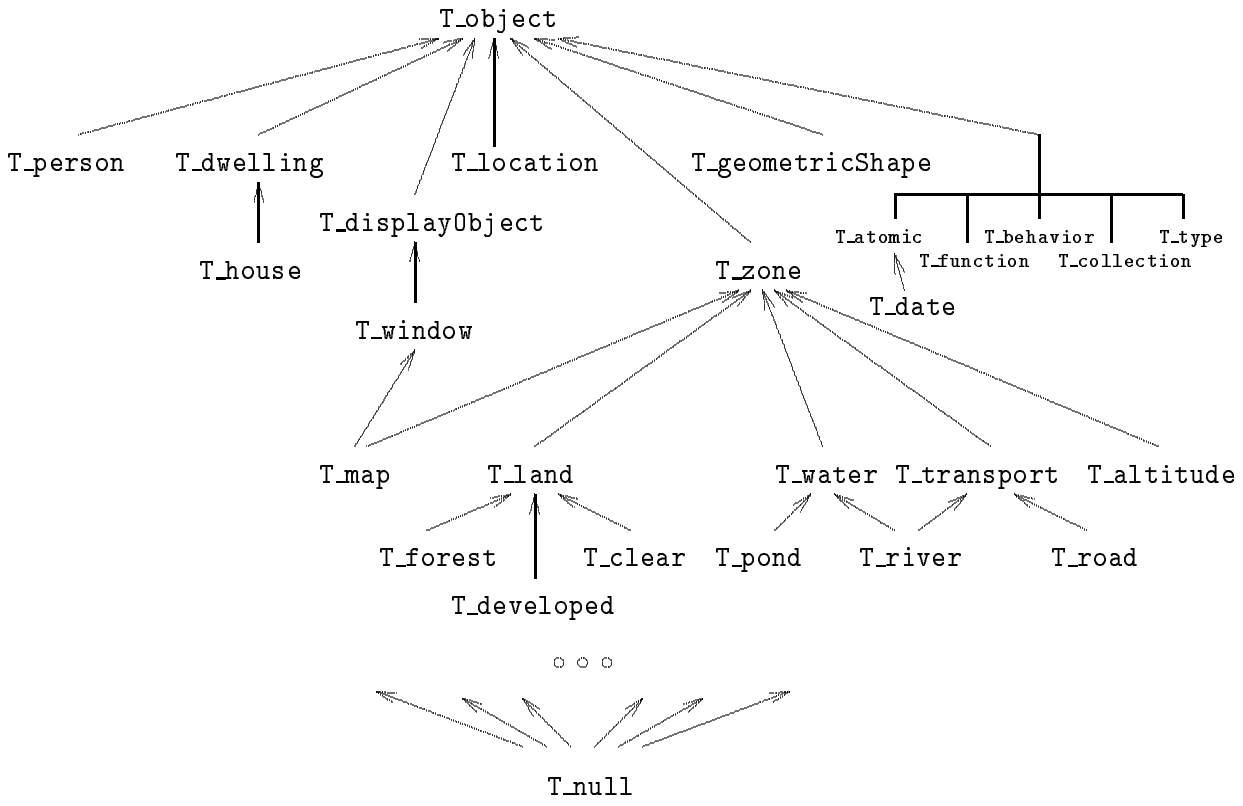

Figure 8.1: Type lattice for a simple geographic information system.

understandable without an elaborate discussion. The example includes the root types of the various sub-lattices from the primitive type system in Figure 3.1 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_date`, `T_dwelling` and `T_house`. Note that `T_date` is a new atomic type introduced by the application which is used to represent dates in a form acceptable to the application.

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.

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.

Table 8.1 lists the signatures of the behaviors defined on GIS specific types in the lattice of Figure 8.1.

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 8.1: Behavior signatures pertaining to example specific types of Figure 8.1.

8.1 Example Definitions

We define the types T_dwelling and T_house for the GIS. The type T_dwelling is a subtype of T_object and it defines two behaviors: *B_address* and *B_inZone*. The type T_house is a subtype of T_dwelling, it specializes behavior *B_inZone* inherited from T_dwelling and defines native behavior *B_mortgage*. The definition of these two types in TDL is as follows:

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`;

Behavior *B_address* is to be associated with a stored function in `T_dwelling`. In the absence of a language with assignment, we opt to define two behaviors, *B_setAddr* and *B_getAddr* for the single behavior *B_address*. Although these two behaviors have different semantics, they will be associated with the same stored function. Behavior *B_mortgage* is handled the same way. In the definition of type `T_house`, we do not specify *B_inZone*, since it is inherited from `T_dwelling`. However, as the implementation for this behavior in type `T_house` changes, an association with a correct function will be performed later. The next step is to create function objects, and to define local references to them.

external function *dw_inZone* : `T_land`;
external function *hs_inZone* : `T_developed`;

We assume that functions *dw_inZone* : `T_land` and *hs_inZone* : `T_developed` already exist somewhere in the system. The above statements declare local references to them. There is a pair of stored functions (one to *set*, one to *get*) for the *address* behavior in `T_dwelling` and a pair of stored functions for the *mortgage* behavior in `T_house`. They are created as part of the association of functions to behaviors which is specified next.

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`;

This completes the definition of `T_dwelling` and `T_house`.

8.2 Example Queries

The following examples illustrate possible queries on the GIS. They are first expressed in TQL, followed by the corresponding object calculus expression and then the equivalent algebraic expression. In the algebraic expressions, we subscript operand collections by the variable that ranges over it. If the operand consists of product objects, we list the variables that make up the components of these objects. The indexed variables are used as a symbolic reference to the elements of the collection as described in Section 7.2. Furthermore, we use the arithmetic notation for operations like *o.greaterthan(p)*, *o.elementof(p)*, etc., instead of their boolean Bspec equivalents. The execution of the algebraic expression is from left-to-right, except that parenthesized expressions have higher priority and are executed first.

Example 8.1 Return land zones valued over $100,000 or that cover an area over 1000 units.

TQL: **select** o
from o **in C_land**
where $(o.B\_value() > 100000)$ **or** $(o.B\_area() > 1000)$

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

TQL: **select** o
from q **in C_person**
where $(o = q.B\_residence().B\_inzone())$

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

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

Example 8.3 Return the maps with areas where senior citizens live.

TQL: **select** o
from o **in C_map**
where exists (**select** p
from p **in C_person**, q **in C_dwelling**
where $(p.B\_age() \geq 65$ **and** $q = p.B\_residence()$
and $q.B\_inzone() \in o.B\_zones()))$

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: $\left(\mathbf{C\_map}_o \bowtie_F \langle \mathbf{C\_dwelling}_q,\ \left(\mathbf{C\_person}_p\ \sigma_{p.B\_age\geq 65}\ \right)_p \rangle\right)_{o,q,p} \triangle_{p,q}$

where F is the predicate $(q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones)$

Example 8.4 Return all maps that describe areas strictly above 5000 feet.

TQL: **select** o
from o **in C_map**
where forAll p **in** (**select** q
from q **in C_altitude**
where $q \in o.B\_zones())$
$p.B\_low() > 5000$

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: $\mathbf{C\_map} - \left(\left(\mathbf{C\_map}_o \bowtie_{p \in o.B\_zones} \left(\mathbf{C\_altitude}_p\ \sigma_{\neg(p.B\_low>5000)}\ \right)_p\right)_{o,p} \triangle_p\right)$

Example 8.5 Return the dollar values of the zones that people live in.

TQL: **select** $p.B\_residence().B\_inzone().B\_value()$
from p **in C_person**

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 simplier form using the map operator as follows:
$\mathbf{C\_person}_p \gg_{p.B\_residence.B\_inzone.B\_value}$

Example 8.6 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*.

TQL: **select** $o[B\_title, B\_area]$
from p **in C_map**, o **in** $p.B\_zones$, q **in C_water**
where $o.B\_proximity(q) < 10$

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: $\Big(\Big(\mathbf{C\_map}_p\ \gamma^o_{o \in p.B\_zones}\ \Big)_{p,o}\ \bowtie_{o.B\_proximity(q)<10}\ \mathbf{C\_water}_q\Big)_{p,o,q}\ \triangle_{q,p}\ \Pi_{B\_title,B\_name}$

Example 8.7 Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

TQL: **select** $p,\ q.B\_title()$
from p **in C_person**, q **in C_map**
where $p.B\_residence().B\_inZone() \in q.B\_zones()$

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: $\Big(\mathbf{C\_person}_p \bowtie_{p.B\_residence.B\_inZone \in q.B\_zones} \Big(\mathbf{C\_map}_q\ \gamma^o_{o=q.B\_title}\ \Big)_{q,o}\Big)_{p,q,o}\ \triangle_q$

Example 8.8 Return (person, spouse, child) triples of all couples and their children where the first parent is homeless. The children set of a couple is "flattened" by grouping each child with their parents.

TQL: **select** $p,\ s,\ c$
from p, s **in C_person**, c **in** $p.B\_children(s)$
where $s = p.B\_spouse()$ **and**
not $p.B\_residence()$ **in** (**select** h
from h **in C_house**)

Calculus: $\{p, s, c \mid \mathbf{C\_person}(p) \wedge \mathbf{C\_person}(s) \wedge c \in p.B\_children(s)$
$\wedge\ s = p.B\_spouse \wedge p.B\_residence \notin \mathbf{C\_house}\}$

Algebra: $\Big(\Big(\mathbf{C\_person}_p\ \sigma_{p.B\_residence \notin \mathbf{C\_house}}\ \Big)_p\ \bowtie_{s=p.B\_spouse}\ \mathbf{C\_person}_s\Big)_{p,s}\ \gamma^c_{c \in p.B\_children(s)}$

Chapter 9

Completeness of Languages

A desired property of the languages of a query model is that they be equivalent in expressive power. That is, any expression formed in one language has an equivalent formation in the other. In the calculus it was shown that certain queries are not "reasonable" because there is no efficient way to process them. Thus, in defining the completeness of our languages we only concern ourselves with the "reasonable" or "safe" expressions.

In this chapter, we show the completeness of the reduction from the algebra to the calculus, the calculus to the algebra, and from TQL to the calculus and algebra. This proves the equivalence of the formal languages (algebra and calculus) and shows that the calculus supports the expressive power of the user language TQL.

9.1 Theorems and Proofs

Theorem 9.1 The reduction from TQL to the object calculus is complete.

Proof: Every statement of TQL presented in Chapter 6 was shown in Section 6.2.3 to have a translation into an equivalent calculus expression. □

Theorem 9.2 The reduction from the object algebra to the object calculus is complete.

Proof: We need to show that if E is an expression in the object algebra, then there is an object calculus expression (OCE) equivalent to E. The proof is by structural induction on the number of operators in E.

Basis. Zero Operators: Then E consists of a single collection name C or a collection creating behavior application $newcoll(c_1, \ldots, c_n)$ where each c_i is a constant. An equivalent OCE for E in the first case is $\{o \mid C'(o)\}$ where C' is the predicate for collection C. In the second case an equivalent OCE for E is $\{o \mid o \in newcoll(c_1, \ldots, c_n)\}$.

Induction: Assume E has at least one operator and that the theorem is true for expressions with fewer operators than E.

Case 1: $E \stackrel{\text{def}}{=} E_1\ \Pi_{\mathcal{B}}$. Since E_1 is an object algebra expression with fewer operators than E, we can find an OCE $\{o \mid \psi_1(o)\}$ equivalent to E_1. Then E is equivalent to $\{o[\mathcal{B}] \mid \psi_1(o)\}$.

Case 2: $E \stackrel{\text{def}}{=} E_1 - E_2$. By renaming of variables if necessary, we can find OCEs $\{o[\mathcal{B}_1] \mid \psi_1(o)\}$ and $\{o[\mathcal{B}_2] \mid \psi_2(o)\}$ equivalent to E_1 and E_2, respectively (the behavioral projections $\mathcal{B}_1$ and $\mathcal{B}_2$ may be empty). Then E is equivalent to $\{o[\mathcal{B}_1] \mid \psi_1(o) \wedge \neg\psi_2(o)\}$.

Case 3: $E \stackrel{\text{def}}{=} E_1 \cup E_2$. We can find OCEs for E_1 and E_2 as in Case 2. Then E is equivalent to $\{o[\mathcal{B}_1 \cap \mathcal{B}_2] \mid \psi_1(o) \vee \psi_2(o)\}$. Note that $\mathcal{B}_1 \cap \mathcal{B}_2$ denotes the intersection of the two component behavioral projections. This intersection represents the proper behavioral projection of the result collection.

Case 4: $E \stackrel{\text{def}}{=} E_1 \cap E_2$. E_1 and E_2 have equivalent OCEs as in Case 2. Then E is equivalent to $\{o[\mathcal{B}_1 \cup \mathcal{B}_2] \mid \psi_1(o) \wedge \psi_2(o)\}$. Here $\mathcal{B}_1 \cup \mathcal{B}_2$ denotes the union of the two component behavioral projections.

Case 5: $E \stackrel{\text{def}}{=} E_1 \Downarrow$. There is an equivalent OCE for E_1 as in Case 2. Then E is equivalent to $\{o \mid \exists o_1(\psi_1(o_1) \wedge o \in o_1)\}$.

Case 6: $E \stackrel{\text{def}}{=} E_1\ \sigma_F\ \langle E_2, \ldots, E_n\rangle$. There are n OCEs equivalent to $E_1, E_2, \ldots, E_n$. Then E is equivalent to $\{o[\mathcal{B}_1] \mid \psi_1(o) \wedge \exists o_2 \ldots \exists o_n(\psi_2(o_2) \wedge \ldots \wedge \psi_n(o_n) \wedge F(o, o_2, \ldots, o_n))\}$.

Case 7: $E \stackrel{\text{def}}{=} E_1 \times \cdots \times E_n$. There are n OCEs equivalent to $E_1, \ldots, E_n$. Then E is equivalent to $\{o \mid \exists o_1 \ldots \exists o_n(\psi_1(o_1) \wedge \cdots \wedge \psi_n(o_n) \wedge o = newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n]))\}$. Here $newprod(o_1[\mathcal{B}_1], \ldots, o_n[\mathcal{B}_n])$ denotes the behavioral application that creates a product object constant whose i^{th} component is the object denoted by o_i that is typed according to the behavioral projection set $\mathcal{B}_i$.

Case 8: $E \stackrel{\text{def}}{=} E_1 \gg_{mop} \langle E_2, \ldots, E_n\rangle$. There are n OCEs equivalent to $E_1, E_2, \ldots, E_n$. Then E is equivalent to $\{o \mid \exists o_1 \exists o_2 \ldots \exists o_n(\psi_1(o_1) \wedge \psi_2(o_2) \wedge \ldots \wedge \psi_n(o_n) \wedge o = mop(o_1, o_2, \ldots, o_n))\}$.

The other algebraic operators can be written in terms of the primitive ones above and this completes the proof. □

Theorem 9.3 The reduction from the the object calculus to the object algebra is complete.

Proof: The reduction from the calculus to the algebra is proven by a translation algorithm that follows the steps illustrated in Figure 9.1. The first step, called *evalify*, determines the *evaluability* (Definition 5.4) of a given object calculus formula. Recall from Section 5.2 that evaluability is enough for safety; this is proved by the translation algorithm in Section 9.2. Moreover, the class of evaluable queries we are translating are *wide-sense evaluable* with respect to equality and membership, meaning a broader class of safe queries are recognized by our approach. If the input formula is not evaluable, it is rejected. From a database perspective, we are only interested in the reduction of those queries which are considered safe. For evaluable formulas, the rest of the translation is similar to that presented in [GT91], except that our extended definitions are used. The *genify* step converts an evaluable formula into an *allowed* form (Definition 9.1) that rewrites the formula to include range "generators" for variables in each subformula. The *ANFify* step places an allowed formula into *Allowed Normal Form* (ANF) (Definition 9.9) that makes each constructive subformula independent of atoms outside the quantifier for the subformula. The *ANFify* step makes use of *Existential Normal Form* (ENF) (Definition 9.7) and *simplified* form (Definition 9.4). The advantage of ANF is that the transformation from this form to the algebra is straightforward. The final step of the translation involves simple pattern matching to *transform* the ANF formula into

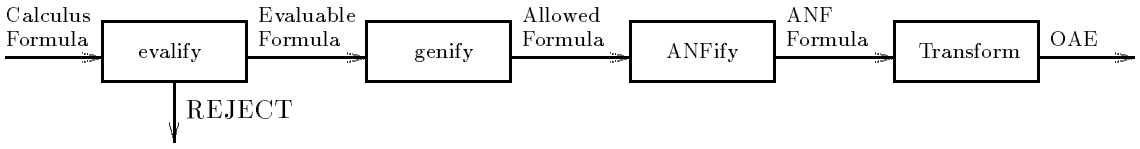

Figure 9.1: Translation steps from object calculus to object algebra.

a (safe) *object algebra expression* (OAE) that is equivalent to the original formula. The complete translation algorithm is presented in Section 9.2. □

Corollary 9.1 The reduction from TQL to the object algebra is complete.

Proof: This follows directly from Theorem 9.1 and Theorem 9.3. □

9.2 Calculus to Algebra Translation

In this section, we describe a complete translation algorithm that converts safe object calculus expressions into equivalent algebraic expressions and rejects expressions that are unsafe. The algebra expressions should be checked for type consistency before they are optimized and prior to an execution plan being generated. Since every object knows its type, this step may be performed during compilation of the query. Query optimization and execution plan generation are not addressed in this report.

To help understand the translation process, we use the following query as a running example. The calculus expression in Example 9.1 will be translated into an equivalent algebra expression with the intermediate steps shown along the way.

Example 9.1 Return zones that are transport zones or that have people living in them.

Consider the query expressed in the following way:

$$\{\ o\ \mid\ \exists p((\mathbf{C\_person}(p)\ \wedge\ o = p.B\_residence.B\_inZone)\ \vee\ \mathbf{C\_transport}(o))\ \}$$

For brevity, we map predicate **C_person** to P, **C_transport** to T and the behavioral application $p.B\_residence.B\_inZone$ to $p.\alpha$. The query can then be written as:

$$\{\ o\ \mid\ \exists p((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o))\ \}$$

Let the formula part of the query be $F \stackrel{\text{def}}{=} \exists p((P(p)\ \wedge\ o = p.\alpha)\ \vee\ T(o))$ □

We first extend the *gen* and *con* rules of Figure 5.2 by adding the notion of "generators" as described in [GT91]. The extended rules are shown in Figure 9.2. The technique adds a third argument $G(x)$ that serves as a "generator" of sorts for the variable x. A $G(x)$ "generator" is a disjunction of *edb* and *gdb* atoms (possibly including a placeholder $\perp$) that generates all the needed objects for x in the given formula and possibly more (i.e., $G(x)$ is a range for x that is at least as large as the values that x can take on in the formula). Moreover, the atoms in $G(x)$ were the ones used to prove that the *gen* or *con* relation holds for variable x in some formula $A(x)$. The placeholder "$\perp$" is used when x is not free in the formula A: it may be thought of as a 0-ary predicate that always fails.

$gdb(x, x = y)$	**if**	$y <_F x$
$gdb(x, x = \beta\{\vec{y}\})$	**if**	$\vec{y} <_F x$
$gdb(x, x \in y)$	**if**	$y <_F x$
$gdb(x, x \in \beta\{\vec{y}\})$	**if**	$\vec{y} <_F x$

$gen(x, A, A)$	**if**	$edb(A)$ **and** $free(x, A)$
$gen(x, A, A)$	**if**	$gdb(x, A)$
$gen(x, \neg A, G)$	**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**	$gdb(x, A)$
$con(x, A, \bot)$	**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** $con(x, B, G_2)$

Figure 9.2: Extended rules of *gen* and *con* that produce "generators".

9.2.1 Evalify: Syntactic Safety Check

The *evalify* algorithm (Algorithm 9.1) syntactically determines whether a given input formula F is evaluable or not and returns an indicator SAFE or REJECT, respectively. Recall from the discussion in Section 5.2 that the *evaluable property* (Definition 5.4) is sufficient for safety. A side-effect of the algorithm is that the partial order $<_F$ for formula F is defined. When *evalify* is first called, the partial order is initialized as being undefined. The algorithm incrementally builds the partial order on each pass through the repeat loop; the first pass orders variables that are generated from *edb* atoms, the second pass orders variables that are generated from variables in the first pass and so on. The *gdb* predicate for the *gen* and *con* rules uses the "partially defined" partial order in each intermediate pass through the repeat loop. Thus, the results of the previous pass are used to update the partial order on the current pass. The temporary set V is used to temporarily store undefined elements of the partial order that are updated after the *gen* and *con* application. This is done to avoid misorderings since the partial order is incrementally built and always used by the *gdb* predicate. If all variables in $<_F$ become ordered, the input formula is evaluable and therefore

SAFE. A fixpoint of the algorithm is reached when no changes are made to the partial order. At this point we REJECT the formula since there are variables in $<_F$ that cannot be ordered, meaning they have no "reasonable" range defined and they cannot be generated from the other variables.

The result of applying *evalify* to the formula F from Example 9.1 is the indicator SAFE and the instantiation of the partial order $\{(p,0),(o,1)\}$ for $<_F$. Two passes are made through the repeat loop. The first pass updates element $(p,0)$ of the partial order and the second pass updates $(o,1)$.

9.2.2 Genify: Adding Range Expressions to Subformulas

The next step of the translation process converts an evaluable formula into an *allowed* form. The definition of *allowed* is as follows:

Definition 9.1 *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.

The allowed property is stronger than evaluable since every formula satisfying the allowed property satisfies the evaluable property (because $gen(x,F)$ implies $con(x,F)$), but the converse does not hold. However, every evaluable formula can be translated into an equivalent allowed formula. The desired properties of allowed formulas are that all variables, free and bound, are generated from the formula and allowed formulas are more robust under certain transformations than evaluable ones. Gelder and Topor [GT91] define *conservative transformations* which include $\vee$ and $\wedge$ distribution that do not always preserve the evaluable property, but do preserve the allowed property. These transformations are used in subsequent steps of the translation to algebra and therefore we convert evaluable formulas into an equivalent allowed form.

As an example of allowed vs. evaluable, consider the formula:

$$P(p) \ \wedge \ \exists q(Q(q) \vee (R(q) \wedge p.\alpha = q.\alpha))$$

which is allowed and the formula:

$$P(p) \ \wedge \ \exists q(Q(q) \vee (\neg R(p) \wedge p.\alpha = p.\beta))$$

which is evaluable, but not allowed because $gen(q, Q(q) \vee (\neg R(p) \wedge p.\alpha = p.\beta))$ does not hold.

Algorithm 9.2 (*genify*) follows the *con-to-gen* algorithm presented in [GT91] and translates an evaluable formula into one that's allowed. The basic procedure of the algorithm is to identify the subformulas $\exists x A$ such that $con(x,A)$ holds, but $gen(x,A)$ fails and then to rewrite these formulas as an equivalent formula, say A', so that $gen(x,A')$ holds. From this point on, unless otherwise noted, we assume that all occurrences of $\forall x A$ in a formula have been replaced with the logical equivalent $\neg\exists x \neg A$. The *genify* algorithm is general in the sense that if the input formula is not evaluable, it can identify this and returns an error. It is necessary before applying the *genify* algorithm that we check separately that $gen(x_i,F)$ holds for all free variables x_i in F. The algorithm relies on the following definitions paraphrased from [GT91].

Algorithm 9.1 *evalify:*

Input: An object calculus formula F

Output: SAFE indicating that F is evaluable or REJECT

Comments: The algorithm incrementally builds the global partial order $<_F$ with each pass through the **repeat** loop. A temporary set V is used to store elements of $<_F$ that need to be updated after each pass.

Initialization

1. For every variable x_i appearing in F, initialize a pair (x_i, ∞) in $<_F$. This indicates that the order for x_i is undefined.
2. $order = 0$

Procedure:

repeat

$V = \{\ \}$
foreach undefined element (x_i, ∞) in $<_F$ **do**
if $free(x_i, F)$ **then**
apply $gen(x_i, F)$
else if x_i is $\exists$ bound as $\exists x A$ **then**
apply $con(x_i, A)$
else x_i must be $\forall$ bound as $\forall x A$
apply $con(x_i, \neg A)$
if *gen* or *con* application succeeded **then**
$V = V \cup \{(x_i, \infty)\}$
endfor
foreach element (x_i, ∞) in V **do**
update element (x_i, ∞) in $<_F$ to $(x_i, order)$ which defines its order
endfor
if no more undefined elements (x_i, ∞) in $<_F$ **then** return SAFE
increment *order*

until no changes made to $<_F$

return REJECT

Algorithm 9.2 *genify:*

Input: An evaluable formula F with universal quantifiers replaced.

Output: An allowed formula equivalent to F.

Procedure:

1. **if** F is an atom **then** return F
2. **if** F has the form $\neg A$ **then** return $\neg genify(F)$
3. **if** F has the form $A \wedge B$ **then** return $genify(A) \wedge genify(B)$
4. **if** F has the form $A \vee B$ **then** return $genify(A) \vee genify(B)$
5. **if** F has the form $\exists x A$ **then**
   (a) **if** $gen(x, A(x), G(x))$ holds **then** return $\exists x\ genify(A(x))$
   (b) **if** $con(x, A, G)$ holds **then**
      i. **if** $notfree(x, A)$ and hence $G = \bot$ **then** return $genify(A)$
      ii. **else** $free(x, A)$ holds and $G = P_1(x) \vee \cdots \vee P_m(x)$ where $m \geq 1$ and some of the disjuncts may be $\bot$. Let $\mathcal{R}$ be the truth value simplification of $A[G/\mathsf{false}]$. Define:
      $$\hat{F} \stackrel{\text{def}}{=} \exists x\ (G(x) \wedge A(x)) \vee \mathcal{R}$$
      and return $genify(\hat{F})$
   (c) Note that if $con(x, A, G)$ does not hold then F is not evaluable so we return an error.

Definition 9.2 *Truth Value Simplification:* The operation of *truth value simplification* consists of applying the following simplifications to a formula for as long as possible.

$$\begin{array}{rcl@{\qquad}rcl} \neg\mathsf{false} & \Longrightarrow & \mathsf{true} & \neg\mathsf{true} & \Longrightarrow & \mathsf{false} \\ A \wedge \mathsf{false} & \Longrightarrow & \mathsf{false} & A \wedge \mathsf{true} & \Longrightarrow & \mathsf{true} \\ A \vee \mathsf{false} & \Longrightarrow & A & A \vee \mathsf{true} & \Longrightarrow & \mathsf{true} \\ \exists x\ \mathsf{false} & \Longrightarrow & \mathsf{false} & \exists x\ \mathsf{true} & \Longrightarrow & \mathsf{true} \\ \forall x\ \mathsf{false} & \Longrightarrow & \mathsf{false} & \forall x\ \mathsf{true} & \Longrightarrow & \mathsf{true} \end{array}$$

Simplifications that depend on the law of the excluded middle, such as $A \vee \neg A \Longrightarrow \mathsf{true}$, are not part of this definition because in general A is a formula and we do not get into recognizing formula equivalences.

Definition 9.3 *Formula Substitution:* Let $G \stackrel{\text{def}}{=} P_1 \vee \cdots \vee P_m$ where P_i are atoms in A. Then $A[G/\mathsf{false}]$ denotes a formula in which each occurrence of P_i in A is replaced by false.

Steps 1-5 of the algorithm traverse the structure of the input formula and step 5 performs the transformations into allowed form on the subformulas that violate this property. If step 5a holds, then there is nothing to do here and we can simply continue traversing the formula. Step 5b must hold in order for the formula to be evaluable and if it does not then an error is produced. If variable x is not free in subformula A, this means that x must not appear in A and so we drop the existential quantifier for x and continue traversing the formula. The key step of the algorithm

is 5(b)ii where F is rewritten into the equivalent $\hat{F}$ form. The purpose of this step is to form a conjunction of the original subformula A with a generator G for the constrained variable x; in effect making $gen(x, G \wedge A)$ hold. The role of $\mathcal{R}$ is to act as the "remainder" of the subformula which moves copies of subformulas that are independent of x (i.e., don't contain x) outside the existential quantifier for x. This is necessary to make F and $\hat{F}$ equivalent because the conjunction of G with A changes the meaning of the subformula.

The result of applying *genify* to our example formula F from Example 9.1 is the formula:

$$F' \stackrel{\text{def}}{=} \exists p(P(p) \ \wedge \ ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o))) \ \vee \ T(o)$$

which is allowed. The steps that produce this formula are as follows:

- The algorithm falls through to step 5 since F has the form $\exists x A$ where:

$$A \stackrel{\text{def}}{=} (P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o)$$

- Step 5a fails, but step 5b succeeds with $con(p, A, G)$ where $G \stackrel{\text{def}}{=} P(p) \ \vee \ \bot$.

- Thus, the algorithm proceeds to step 5(b)ii and the result of applying this step to our example formula defines the following:

$$\begin{array}{lll} \mathcal{R} & \stackrel{\text{def}}{=} & T(o) \\ \hat{F} & \stackrel{\text{def}}{=} & \exists p((P(p) \ \vee \ \bot) \ \wedge \ ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o))) \ \vee \ T(o) \end{array}$$

  $\hat{F}$ is in allowed form and by replacing all occurrences of $\bot$ with false and carrying out truth value simplification we get the output formula F'.

9.2.3 ANFify: Making Subformulas Independent

The next step of translation is to normalize an allowed formula by putting it into *Allowed Normal Form* (ANF). The reason for converting a formula into ANF is that every proper *constructive subformula* (see Definition 9.6 below) can generate objects for all the free variables in the subformula. This in effect makes every constructive subformula independent of atoms that appear outside the quantifier for the subformula, meaning that in the final translation to the algebra we can translate subformulas independent of the atoms outside the quantifier for the subformula. The transformation of an ANF formula into an object algebra expression is straightforward by simple pattern matching starting with the inner subformulas and moving to the outer formula. At times our discussion assumes a tree structured representation for a formula where the leaves represent atoms from the calculus and the internal nodes are the connectives $\exists, \vee, \wedge, \neg$. Algorithm 9.3 (*ANFify*) and the definition of ANF depend on the following definitions that extend those presented in [GT91] by including a notion for membership.

Definition 9.4 *Simplified Form:* A formula (with universal quantifiers replaced) is call *simplified* if the following conditions are met:

1. There is no occurrence of $\neg\neg A$. It is replaced by the logical equivalent A.

2. There are no occurrences of $\neg(s = t), \neg(s \neq t), \neg(s \in t), \neg(s \notin t)$. They are replaced by their logical equivalents $(s \neq t), (s = t), (s \notin t), (s \in t)$, respectively.

3. The operators $\wedge, \vee, \exists$ are made polyadic and are flattened, meaning:

   (a) in a subformula $A_1 \wedge \cdots \wedge A_n$, $n \geq 2$ and no operand A_i is itself a conjunction,

   (b) in a subformula $A_1 \vee \cdots \vee A_n$, $n \geq 2$ and no operand A_i is itself a disjunction,

   (c) in a subformula $\exists \vec{x} A$, operand A does not begin with $\exists$.

4. In a subformula $\exists \vec{x} A$, $free(x_i, A)$ holds for every variable x_i.

An algorithm to translate a formula into simplified form follows immediately from the definition. We assume the existence of a function *simplify* that transforms an arbitrary formula into its equivalent simplified form satisfying Definition 9.4. The following three definitions formalize the notion of *Existential Normal Form* (ENF).

Definition 9.5 *Negative/Positive Formulas:* A simplified formula is *negative* if its root is "$\neg$"; otherwise, it is *positive*. An arbitrary formula is *negative* (resp. *positive*) if its simplified form is negative (resp. positive).

Definition 9.6 *Restrictive/Constructive Subformulas:* A subformula A of a simplified formula F is *restrictive* if the parent of A is "$\wedge$" and either A is negative or A is an atom and $edb(A)$ does not hold; otherwise A is *constructive*.

Definition 9.7 *Existential Normal Form:* A formula is in *Existential Normal Form* (ENF) if:

1. it is simplified; and

2. for each disjunction in the formula:

   (a) the parent of the disjunction, if it has one, is "$\wedge$"; and

   (b) each operand of the disjunction is a positive formula.

The existential normal form prohibits certain parent/child combinations illustrated by the non-blank entries in Figure 9.3 and these entries specify rewrite rules that correct the violations. The s along the diagonal indicates a call to *simplify* on the formula and has the highest priority.

Defining an algorithm for converting any arbitrary formula into ENF is straightforward from Figure 9.3 and one is presented in [GT91]. Furthermore, a Lemma is provided stating that if the input formula to the algorithm is allowed, then so is the output formula. This means that we can put an allowed formula into ENF without losing the allowed property. ENF is important for the final translation into ANF. Let *ENFify* be a function that performs ENF normalization.

The following two definitions formalize the notion of *allowed normal form*.

Definition 9.8 *genall:* The property $genall(F)$ holds for a formula F if and only if $gen(x_i, F)$ holds for every free variable appearing in F.

Definition 9.9 *Allowed Normal Form:* A formula F is in *Allowed Normal Form* (ANF) if it is in ENF, $genall(F)$ holds, and every constructive subformula A of F is in ANF.

	Parent			
Child	$\vee$	$\wedge$	$\exists$	$\neg$
$\vee$	s		R3	R2
$\wedge$		s		R1[a]
$\exists$			s	
$\neg$	R1A			s

[a] Only if every conjunct of $\wedge$ is negative.

$$
\begin{array}{rrcl}
\text{R1}: & \neg(\neg A_1 \wedge \cdots \wedge \neg A_n) & \Longrightarrow & A_1 \vee \cdots \vee A_n \\
\text{R1A}: & \neg A \vee B_1 \vee \cdots \vee B_n & \Longrightarrow & \neg(A \wedge \neg B \wedge \cdots \wedge \neg B_n) \\
\text{R2}: & \neg(A_1 \vee \cdots \vee A_n) & \Longrightarrow & (\neg A_1 \wedge \cdots \wedge \neg A_n) \\
\text{R3}: & \exists \vec{x}(A(\vec{x}) \vee B(\vec{x})) & \Longrightarrow & (\exists \vec{x} A(\vec{x}) \vee \exists \vec{x} B(\vec{x}))
\end{array}
$$

Figure 9.3: Prohibitive parent/child combinations in ENF formulas and rewrite rules to correct the violations. The s entry indicates a call to *simplify* on the formula and has highest priority.

Algorithm 9.3 (*ANFify*) transforms an allowed ENF formula into an equivalent ANF formula. The algorithm is based on the repeated application of the rewrite rules in Figure 9.3. Application of rules for Case 1 and Case 2 require the resulting formula to be simplified before recursing on the formula. Case 3 may produce a non-ENF formula (e.g., $D \wedge \neg((A_1 \vee A_2) \wedge B)$) and so a call to *ENFify* is necessary before recursing. A fixpoint of the algorithm is reached when no changes are made to the input formula F and at this point F is in allowed normal form.

The purpose of the *ANFify* algorithm is to rewrite every proper constructive subformula so that all free variables in the subformula are generated by the subformula itself. This ensures that every constructive subformula is allowed and therefore can be "evaluated" independently of the atoms outside the quantifier for this formula. This motivates the following Lemma that removes the recursion in Definition 9.9, but yields the same class of ANF formulas.

Lemma 9.1 An ENF formula F is in ANF if and only if F is allowed and every constructive subformula A of F is allowed.

Proof: Immediate from the definition of ANF and structural induction on F. We refer the reader to [GT91] for the formal proof.

The result of applying *ANFify* to the allowed formula F' produced by the *genify* algorithm in the previous section is the formula:

$$F'' \stackrel{\text{def}}{=} \exists p(P(p) \ \wedge \ o = p.\alpha) \ \vee \ \exists p(P(p) \ \wedge \ T(o)) \ \vee \ T(o)$$

which is in ANF. The steps that produce this formula are as follows:

- The algorithm matches on Case 3 with the following being defined from the formula F':

$$
\begin{array}{rcl}
F_1 & \stackrel{\text{def}}{=} & P(p) \ \wedge \ ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o)) \\
B_1 & \stackrel{\text{def}}{=} & P(p) \\
G & \stackrel{\text{def}}{=} & ((P(p) \ \wedge \ o = p.\alpha) \ \vee \ T(o)) \\
A_1 & \stackrel{\text{def}}{=} & (P(p) \ \wedge \ o = p.\alpha) \\
A_2 & \stackrel{\text{def}}{=} & T(o)
\end{array}
$$

Algorithm 9.3 *ANFify:*

Input: An allowed formula F in ENF.

Output: An ANF formula equivalent to F.

Comments:

The algorithm assumes a tree structure representation of formulas. The notation $F[A/B]$ where A is a subtree (subformula) of F denotes an operation that replaces the subtree of A in F by the tree representation of formula B.
In each of the cases below, F_1 is an allowed (not necessarily proper) subformula of F to be replaced and F_2 is the equivalent allowed formula that replaces F_1. The notation "$F_1 \stackrel{\text{def}}{=} \cdots$" means that F_1 matches the allowed formula pattern on the right-hand side. If none of the patterns can be matched to some subformula of F, the algorithm falls through to the otherwise clause which causes the procedure to terminate.

Procedure:

Case 1: $F_1 \stackrel{\text{def}}{=} \exists \vec{y} A \wedge B_1 \wedge \cdots \wedge B_n$, and $genall(A)$ does not hold:

- Let $\vec{x}$ be the set of variables that are free in A such that $gen(x_i, A)$ fails (since F_1 is allowed, this set is disjoint from $\vec{y}$).
- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that $genall(A \wedge B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Let $F_2 \stackrel{\text{def}}{=} \exists \vec{y}(A \wedge B_1 \wedge \cdots \wedge B_k) \wedge B_{k+1} \wedge \cdots \wedge B_n$
- return $ANFify(simplify(F[F_1/F_2]))$

Case 2: $F_1 \stackrel{\text{def}}{=} \neg A \wedge B_1 \wedge \cdots \wedge B_n$, and $genall(A)$ does not hold:

- Let $\vec{x}$ be the set of variables that are free in A such that $gen(x_i, A)$ fails.
- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that all $\vec{x}$ are free in $B_1 \wedge \cdots \wedge B_k$ and $genall(B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Let $G \stackrel{\text{def}}{=} ANFify(B_1 \wedge \cdots \wedge B_k)$
- Let $F_2 \stackrel{\text{def}}{=} \neg(A \wedge G) \wedge B_1 \wedge \cdots \wedge B_n$
- return $ANFify(simplify(F[F_1/F_2]))$

Case 3: $F_1 \stackrel{\text{def}}{=} G \wedge B_1 \wedge \cdots \wedge B_n$, where $G \stackrel{\text{def}}{=} A_1 \vee \cdots \vee A_m$ and $genall(G)$ does not hold:

- Let $B_1 \wedge \cdots \wedge B_k$ be a prefix (possibly after rearrangement) of $B_1 \wedge \cdots \wedge B_n$ such that $genall(G \wedge B_1 \wedge \cdots \wedge B_k)$ holds (at worst $k = n$ because $genall(F_1)$ holds).
- Distribute $B_1 \wedge \cdots \wedge B_k$ over G.
- For $1 \leq i \leq m$ do: let $G_i \stackrel{\text{def}}{=} ANFify(A_i \wedge B_1 \wedge \cdots \wedge B_k)$
- Let $F_2 \stackrel{\text{def}}{=} (G_1 \vee \cdots \vee G_m) \wedge B_{k+1} \wedge \cdots \wedge B_n$
- return $ANFify(ENFify(F[F_1/F_2]))$

Otherwise: return F

Carrying out the distribution of B_1 over G produces two G_i formulas that are in ANF and define the final result formula as follows:

$$\begin{array}{rcl} G_1 & \stackrel{\text{def}}{=} & (P(p) \ \wedge \ P(p) \ \wedge \ o = p.\alpha) \\ & \equiv & (P(p) \ \wedge \ o = p.\alpha) \\ G_2 & \stackrel{\text{def}}{=} & (P(p) \ \wedge \ T(o)) \\ F_2 & \stackrel{\text{def}}{=} & (P(p) \ \wedge \ o = p.\alpha) \ \vee \ (P(p) \ \wedge \ T(o)) \\ F[F_1/F_2] & \stackrel{\text{def}}{=} & \exists p((P(p) \ \wedge \ o = p.\alpha) \ \vee \ (P(p) \ \wedge \ T(o))) \ \vee \ T(o) \end{array}$$

The call to *ENFify* on F distributes the $\exists p$ over the disjunct. The resulting formula is in ANF and is the output of *ANFify* as formula F''.

9.2.4 Transform: Translating into Algebra

The final step of translation involves the transformation of an ANF formula into an equivalent series of object algebra operations. This step follows immediately from the structure of an ANF formula. Every range atom $C(x)$ is translated into a name C' that represents the collection of the range predicate. Atoms $x = c$ and $x \in c$ are translated as part of appropriate select or generate operations (see below) or into appropriate collections as follows:

$$\begin{array}{rcl} x = c & \Longrightarrow & newcoll(c)_x \\ x \in c & \Longrightarrow & c_x \end{array}$$

The first case creates a collection containing the single constant c and the second case uses c as the name for the collection. Recall that the subscript x is our notation from Section 7.2 which indicates that the result collection is a range for variable x.

Next, we apply the transformations shown in Figure 9.4 to the remaining proper constructive subformulas and then combine the subformulas. In the figure, A and B refer to subformulas, A' and B' refer to the algebraic equivalents of A and B respectively, F refers to a predicate, *mop* refers to a mop function, and θ is one of $=$ or $\in$. Algebraic expressions are subscripted with the variables that they represent (or that their components represent in the case of product objects). Furthermore, $A(\vec{x})$ is used to denote that $\vec{x}$ are the only free variables in A. The same applies to $F(\vec{x})$ and $mop(\vec{x})$. For join terms of the form $\vec{x} = \vec{x}$, it is assumed that one set of $\vec{x}$ refer to components of A' while the other set refers to components of B'.

Transformation (9.5) is known as a *generalized set difference* [HHT75] and could be defined as a primitive derived operator in the algebra so that efficient join techniques could be defined to process it.

Transformation (9.7) defines a join between the common variables (components) of A and B. Transformation (9.8) defines a join using a predicate (general mop function) over the components of A and B. Transformation (9.9) is a general case of (9.7) and (9.8) which defines a join between an A and a B that have some variables in common (namely, $\vec{w}, \vec{x}$), some variables not in common (A has $\vec{u}, \vec{v}$ and B has $\vec{y}, \vec{z}$), and a predicate over some of the common and uncommon variables of A and B (namely, $\vec{u}, \vec{w}, \vec{y}$). Transformation (9.11) defines a generate join over an A and a B that have no variables in common, and a generating atom over over some of the variables of A and B. The reason A and B cannot have common variables is that the relationship between these variables would be lost in the operation. If A and B have common variables, then they should be joined instead. Transformation (9.10) is a special case of (9.11).

$$
\begin{aligned}
A(\vec{x}) \vee B(\vec{x}) &\implies (A'_{\vec{x}} \cup B'_{\vec{x}})_{\vec{x}} && (9.1)\\
A(\vec{x}) \wedge B(\vec{x}) &\implies (A'_{\vec{x}} \cap B'_{\vec{x}})_{\vec{x}} && (9.2)\\
A(\vec{x}) \wedge B(\vec{y}) &\implies (A'_{\vec{x}} \times B'_{\vec{y}})_{\vec{x},\vec{y}} && (9.3)\\
A(\vec{x}) \wedge \neg B(\vec{x}) &\implies (A'_{\vec{x}} - B'_{\vec{x}})_{\vec{x}} && (9.4)\\
A(\vec{x},\vec{y}) \wedge \neg B(\vec{y}) &\implies (A'_{\vec{x},\vec{y}} - (A'_{\vec{x},\vec{y}} \bowtie_{\vec{y}=\vec{y}} B'_{\vec{y}})_{\vec{x},\vec{y}})_{\vec{x},\vec{y}} && (9.5)\\
A(\vec{x},\vec{y}) \wedge F(\vec{x}) &\implies (A'_{\vec{x},\vec{y}}\ \sigma_F\ )_{\vec{x},\vec{y}} && (9.6)\\
A(\vec{x},\vec{y}) \wedge B(\vec{x},\vec{z}) &\implies (A'_{\vec{x},\vec{y}} \bowtie_{\vec{x}=\vec{x}} B'_{\vec{x},\vec{z}})_{\vec{x},\vec{y},\vec{z}} && (9.7)\\
A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge F(\vec{y},\vec{z}) &\implies (A'_{\vec{x},\vec{y}} \bowtie_F B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z}} && (9.8)\\
A(\vec{u},\vec{v},\vec{w},\vec{x}) \wedge B(\vec{w},\vec{x},\vec{y},\vec{z}) \wedge F(\vec{u},\vec{w},\vec{y}) &\implies (A'_{\vec{u},\vec{v},\vec{w},\vec{x}} \bowtie_{\vec{w}=\vec{w}\wedge\vec{x}=\vec{x}\wedge F} B'_{\vec{w},\vec{x},\vec{y},\vec{z}})_{\vec{u},\vec{v},\vec{w},\vec{x},\vec{y},\vec{z}} && (9.9)\\
A(\vec{x},\vec{y}) \wedge o\theta mop(\vec{x}) &\implies (A'_{\vec{x},\vec{y}}\ \gamma^{o}_{o\theta mop}\ )_{\vec{x},\vec{y},o} && (9.10)\\
A(\vec{x},\vec{y}) \wedge B(\vec{w},\vec{z}) \wedge o\theta mop(\vec{y},\vec{z}) &\implies (A'_{\vec{x},\vec{y}}\ \gamma^{o}_{o\theta mop}\ B'_{\vec{w},\vec{z}})_{\vec{x},\vec{y},\vec{w},\vec{z},o} && (9.11)\\
\exists\vec{y} A(\vec{x},\vec{y}) &\implies (A'_{\vec{x},\vec{y}} \triangle_{\vec{y}})_{\vec{x}} && (9.12)
\end{aligned}
$$

Figure 9.4: Transformations from object calculus to object algebra.

Transformations of join and generate join over two operands can be generalized over multiple operands. For example, there is the opportunity to perform the following transformations on the given formula:

$$
\begin{aligned}
A(\vec{x}) \wedge B(\vec{y}) \wedge C(\vec{z}) \wedge F(\vec{x},\vec{y},\vec{z}) &\implies (A_{\vec{x}} \bowtie_F \langle B_{\vec{y}}, C_{\vec{z}} \rangle)_{\vec{x},\vec{y},\vec{z}}\\
A(\vec{x}) \wedge B(\vec{y}) \wedge C(\vec{z}) \wedge o = mop(\vec{x},\vec{y},\vec{z}) &\implies (A_{\vec{x}}\ \gamma^{o}_{o=mop}\ \langle B_{\vec{y}}, C_{\vec{z}} \rangle)_{\vec{x},\vec{y},\vec{z},o}
\end{aligned}
$$

This groups the collections involved in the operation with the operator and may provide some opportunities for optimization such as grouping together collections that may be clustered on disk.

The last stage of the transformation is to apply the necessary project operation using behavioral projections in the target list of the object calculus expression. This operation does not change the extent of the result collection. Rather, it has the effect of generalizing a new membership type for the collection that only includes the behaviors specified in the projection.

The result of applying the transformations to the ANF formula F'' output by the *ANFify* algorithm in the previous section is the algebraic expression:

$$\Big((P_p\ \gamma^{o}_{o=p.\alpha}\ )_{p,o} \triangle_p\Big)_o \ \cup\ ((P_p \times T_o)_{p,o} \triangle_p)_o \ \cup\ T_o$$

Written using the constructs of the original query it is:

$$\Big((\textbf{C\_person}\ \gamma^{o}_{o=p.B\_residence.B\_inZone}\ ) \triangle_p\Big) \cup ((\textbf{C\_person} \times \textbf{C\_transport}) \triangle_p)\ \cup\ \textbf{C\_transport}$$

There are opportunities for optimization on this expression, but the importance of this section was to show the correct translation from calculus to algebra. The expression should also be type checked to ensure that the behaviors used in the expression are actually defined for the objects to which they're being applied. During type checking the test for operand finiteness can also take place. The resulting example query is safe in all respects that we have considered in this report.

Chapter 10

Conclusions and Future Work

In this paper, we give a complete formal specification of the TIGUKAT uniform extensible query model. This includes the user-level definition language, user-level query language, user-level control language, the object calculus, the object algebra, and completeness proofs of the languages. In keeping with the uniformity aspects of the TIGUKAT object model [PÖS92], the query model is defined in a consistent way as type and behavior extensions to the base object model. Thus, queries are objects with well-defined behavior. This is a uniform object-oriented approach to developing an extensible query model that is seamlessly integrated with the object model. This kind of natural extension is possible due to the uniformity built into the object model which treats everything as a first-class object and allows the consistent abstraction of an object's "attributes" into the uniform semantics of behaviors. This specification is being used as a foundation for implementing the query model.

The TIGUKAT Query Language (TQL) is a user-level language with similarities to the SQL3 standardization efforts [Gal92]. Therefore, TQL should be SQL ready when standards finally emerge. The TIGUKAT Definition Language (TDL) provides the ability to extend the object model with types, classes, behaviors and functions. The TIGUKAT Control Language (TCL) provides a simplified set of commands for controlling a session within the TIGUKAT query processor.

The formal object calculus is a powerful declarative object creating language that incorporates the behavioral paradigm of the object model. Safety is based on the evaluable class of queries [GT91] which is arguably the largest decidable subclass of the domain independent class [Mak81]. The class of evaluable queries we define are *wide-sense evaluable* with respect to equality and membership atoms, meaning a broader class of safe queries is recognized by our approach. Furthermore, a notion of *operand finiteness* is defined that checks if a query adds objects to the collections and classes that it is ranging over and rejects it if it does.

The object algebra includes a powerful, complete set of the behavioral/functional operators that fully support the object-creating nature of the calculus. A novel operator is *behavioral projection* which is a form of type generalization. Other notable operators include a generalized *map* for applying behaviors to elements of collections, a *select* and the derived *join* and *generate join* operators. We have chosen not to include a *powerset* operator in our algebra because one of our primary interests is to produce an efficient implementation of our query model. Use of powerset may cause exponential growth of collections and the costs associated with this is unacceptable for our implementation. Furthermore, a clean definition of safety with respect to powerset that complies with the efficient translation of evaluable formulas (i.e., without forming a *DOM* domain)

is not apparent. The calculus and algebra are proven to be equivalent in expressive power and the reduction from TQL to the calculus is complete. Furthermore, a feasible translation algorithm from calculus to algebra is presented that does not depend on the formation of (potentially) large *DOM* domains. Object creating languages require the ability to perform type inferencing because newly created objects may not correspond to any type in the lattice. As part of the algebra, we define how the operators relate to the schema in terms of the creation and integration of new types.

There are a number of ongoing research activities related to this project. A main memory version of the TIGUKAT object model has been implemented. We are coupling this implementation with the EXODUS storage manager [CDV88] to provide persistence. We are implementing a compiler for the query model and user language on top of the object model implementation. We are developing and implementing an extensible query optimizer for the algebra, including an execution plan generator with a generic interface to an object storage manager such as EXODUS. The query optimizer design loosely follows the strategy in [LV91] to the extent that both employ object-oriented techniques in defining the components of the optimizer in an extensible way. The empirical experiences of [BMG92] in implementing an object-oriented optimizer has several lessons and recommendation to help guide the development of new optimizer technologies for non-traditional models. Our query optimizer is uniformly integrated with the object model, meaning the query optimizer is defined as a collection of objects with well-defined behavior. The final step of our query processor implementation is the design of its interface with the object storage manager and the generation of an efficient execution plan for retrieving objects in the result collection of the query.

Another issue we are addressing is the definition of the update semantics for the model. In this paper, we have defined the syntax and semantics of a TIGUKAT Definition Language (TDL) that allows for the consistent creation of schema such as types, classes and collections. Furthermore, we have defined the syntax of TQL statements **insert, update** and **delete** to perform updates on the objectbase and are currently working out the semantics of these constructs. A related issue involves handling behaviors with side effects and we are hoping to develop some rules for dealing with these in our languages.

The definition of the object query model, user language and optimizer provides efficient declarative access to objects. However, this model currently works on the physical database as defined by its *conceptual schema*. A conceptual schema can have various meanings to different applications. That is, there can be several *external schemas* for a single *conceptual schema*. These various perspectives must be specifiable in a model without having to redefine the physical or conceptual schema for each application. The definition and management of external schemas is part of a view management facility. We are currently developing a view manager (with an update semantics) for the model. Following uniformity, views are defined as first class objects with well-defined behavior.

The object-oriented approach is a suitable candidate for facilitating an integration of the data abstraction and computation model of object-oriented programming languages with the performance and consistency of an object query model. Traditionally, these two areas have developed orthogonally to each other. An integration would alleviate many problems (e.g., impedance mismatch) associated with embedded languages in use today. We are investigating how a uniform behavioral model like TIGUKAT may lead to a merger of these two disciplines and are looking at developing a methodology for specifying a seamless interface between them. The definition of a uniform objectbase programming language is one of the possibilities we are exploring.

Bibliography

[AB93] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex object. Technical report, INRIA, France, 1993.

[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.

[AG89] R. Agrawal and N.H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 36–45, May 1989.

[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 Databases*, pages 433–442, 1989.

[Bac78] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and it's Algebra of Programs. *Communications of the ACM*, 21(8):613–641, 1978.

[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.

[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.

[BMG92] J.A. Blakeley, W.J. McKenna, and G. Graefe. DARPA Open Object-Oriented Database: Experiences Building the Open OODB Query Optimizer. Technical Report DBSB-92-12-01, Texas Instruments, December 1992.

[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, September 1988.

[Cha92] E.P.F. Chan. Containment and Minimization of Positive Conjunctive Queries in OODB's. In *Proc. of the 11th ACM Symposium on Principles of Database Systems*, pages 202–211, June 1992.

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

[Dav90] K.C. Davis. *A Formal Foundation for Object-Oriented Algebraic Query Processing.* PhD thesis, University of Southwestern Louisiana, 1990.

[Day89] U. Dayal. Queries and Views in an Object-Oriented Data Model. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 80–102, June 1989.

[Dem81] R. Demolombre. Assigning Meaning to Ill-Defined Queries Expressed in Relational Calculus. In *Advances in Database Theory.* Plenum Press, 1981.

[Dem82] R. Demolombre. Syntactical Characterization of a Subset of Domain Independent Formulas. Technical report, ONERA-CERT, 1982.

[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.

[DiP69] R.A. DiPaola. The Recursive Unsolvability of the Decision Problem for the Class of Definite Formulas. *Journal of the ACM*, 16(2), 1969.

[ES90] M.A. Ellis and B. Stroustroup. *The Annotated C++ Reference*, chapter 10. Addison Wesley, 1990.

[Fag82] R. Fagin. Horn Clauses and Database Dependencies. *Journal of the ACM*, 29(4), 1982.

[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. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[FKMT91] E. Fong, W. Kent, K. Moore, and C. Thompson. X3/SPARC/DBSSG/OODBTG Final Report. Technical report, NIST, September 1991.

[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.

[GT87] A.V. Gelder and R.W. Topor. Safety and Correct Translation of Relational Calculus Formulas. In *Proc. of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, pages 313–327. ACM Press, March 1987.

[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.

[HHT75] P. Hall, P. Hitchcock, and S. Todd. An Algebra of Relations for Machine Computation. In *Proc. of the ACM Symposium on Principles of Programming Languages*, pages 225–232, 1975.

[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.

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

[Kuh67] J.L. Kuhns. Answering Questions by Computer: A Logical Study. Technical Report RM-5428-PR, Rand Corp., 1967.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, 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 Databases*, 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.

[LV91] R.S.G. Lanzelotte and P. Valduriez. Extending the Search Strategy in a Query Optimizer. In *Proc. 17th Int'l Conf. on Very Large Databases*, pages 363–373, September 1991.

[Mak81] J.A. Makowsky. Characterizing Data Base Dependencies. In *Proc.*8^{th} *Colloquium on Automata, Languages and Programming.* Springer Verlag, 1981.

[MB90] F. Manola and A.P. Buchmann. A Functional/Relational Object-Oriented Model for Distributed Object Management. Technical Memorandum TM-0331-11-90-165, GTE Laboratories Incorporated, December 1990.

[MD86] F. Manola and U. Dayal. PDM: An Object-Oriented Data Model. In K.R. Dittrich and U. Dayal, editors, *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 18–25. IEEE Computer Science Press, 1986.

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

[ND82] J.M. Nicolas and R. Demolombre. On the Stability of Relational Queries. Technical report, ONERA-CERT, 1982.

[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.

[Osb88] S.L. Osborn. Identity, Equality and Query Optimization. In *Proc. of the 2nd Int'l Workshop on Object-Oriented Database Systems*, pages 346–351. Springer Verlag, September 1988.

[ÖSP93] M.T. Özsu, D.D. Straube, and R.J. Peters. Query Processing Issues in Object-Oriented Knowledge Base Systems. In F.E. Petry and L.M. Delcambre, editors, *Emerging Landscape of Intelligence in Database and Information Systems*. JAI Press, 1993. In press.

[OW89] G. Ozsoyoglu and H. Wang. A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages. *IEEE Transactions on Software Engineering*, SE-15(9):1038–1052, September 1989.

[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.

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

[SÖ90a] D.D. Straube and M.T. Özsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[SÖ90b] 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.

[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.

[SS90] M. Scholl and H. Schek. A Relational Object Model. In *Third Int'l Conf. on Database Theory*, pages 89–105, December 1990.

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

[SZ89] G. Shaw and S. Zdonik. An Object-Oriented Query Algebra. In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 103–112, June 1989.

[SZ90] G. Shaw and S. Zdonik. A Query Algebra for Object-Oriented Databases. In *Proc. 6th Int'l. Conf. on Data Engineering*, pages 154–162, February 1990.

[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.

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

[YO91] L. Yu and S.L. Osborn. An Evaluation Framework for Algebraic Object-Oriented Query Models. In *Proc. 7th Int'l. Conf. on Data Engineering*, pages 670–677, April 1991.

Appendix A

Primitive Type System

Table A.1 displays the signatures for the behaviors defined on the non-atomic types (except the container types) of the primitive type system. Table A.2 shows the signatures for the behaviors defined on the container types of the primitive type system. Table A.3 lists the signatures for the behaviors defined on the atomic types of the primitive type system. The receiver type of a behavior is not given because the receiver must be an object of a type that is compatible with the type defining the behavior.

Note that the type specifications for the behaviors are the *most general* types. Types for some of the behaviors are revised in the subtypes that inherit them. For example, the result type of *B_self* is always the type of the receiver object and the result type of *B_new* is always the membership type of the receiver class.

Type	Signatures	
T_object	*B_self*:	T_object
	B_mapsto:	T_type
	B_conformsTo:	T_type → T_boolean
	B_equal:	T_object → T_boolean
	B_notequal:	T_object → T_boolean
	B_newprod:	T_list⟨T_object⟩ → T_list⟨T_collection⟨T_behavior⟩⟩ → T_object
T_type	*B_interface*:	T_collection⟨T_behavior⟩
	B_native:	T_collection⟨T_behavior⟩
	B_inherited:	T_collection⟨T_behavior⟩
	B_specialize:	T_type → T_boolean
	B_subtype:	T_type → T_boolean
	B_subtypes:	T_collection⟨T_type⟩
	B_supertypes:	T_collection⟨T_type⟩
	B_sub-lattice:	T_poset⟨T_type⟩
	B_super-lattice:	T_poset⟨T_type⟩
	B_classof:	T_class
	B_tmeet:	T_collection⟨T_type⟩ → T_type
	B_tjoin:	T_collection⟨T_type⟩ → T_type
	B_tproduct:	T_list⟨T_type⟩ → T_type
T_product	*B_compTypes*:	T_list⟨T_type⟩
T_behavior	*B_name*:	T_string
	B_argTypes:	T_list⟨T_type⟩
	B_resultType:	T_type → T_type
	B_semantics:	T_object
	B_associate:	T_type → T_function → T_behavior
	B_implementation:	T_type → T_function
	B_primitiveApply:	T_object → T_object
	B_apply:	T_object → T_list → T_object
	B_defines:	T_collection⟨T_type⟩
T_function	*B_argTypes*:	T_list⟨T_type⟩
	B_resultType:	T_type
	B_source:	T_object
	B_primitiveExecute:	T_object → T_object
	B_execute:	T_list → T_object
	B_compile:	T_object
	B_executable:	T_object

Table A.1: Behavior signatures of the non-atomic types of the primitive type system.

Type	Signatures
T_collection	*B_memberType*: T_type
	B_union: T_collection → T_collection
	B_diff: T_collection → T_collection
	B_intersect: T_collection → T_collection
	B_collapse: T_collection
	B_select: T_string → T_list⟨T_collection⟩ → T_collection
	B_project: T_collection⟨T_behavior⟩ → T_collection
	B_map: T_string → T_list⟨T_collection⟩ → T_collection
	B_product: T_collection⟨T_collection⟩ → T_collection
	B_join: T_string → T_list⟨T_collection⟩ → T_collection
	B_genjoin: T_string → T_list⟨T_collection⟩ → T_collection
	B_setEqual: T_collection → T_boolean
	B_containedBy: T_collection → T_boolean
	B_cardinality: T_natural
	B_elementOf: T_object → T_boolean
	B_insert: T_object → T_collection
	B_delete: T_object → T_collection
T_bag	*B_occurrences*: T_object → T_natural
	B_count: T_natural
	Behaviors from T_collection refined to preserve duplicates
T_poset	*B_ordered*: T_object → T_object → T_boolean
	B_ordering: T_behavior
	Behaviors from T_collection refined to preserve ordering
T_list	Behaviors refined to preserve duplicates and ordering
T_class	*B_deepExtent*: T_collection
	B_new: T_object
T_class-class	*B_new*: T_type → T_class
T_type-class	*B_new*: T_collection⟨T_type⟩ → T_collection⟨T_behavior⟩ → T_type
T_collection-class	*B_new*: T_type → T_collection
T_product-class	*B_new*: T_list⟨T_object⟩ → T_object

Table A.2: Behavior signatures of the container types of the primitive type system.

Type	Signatures	
T_atomic		
T_boolean	*B_not*:	T_boolean
	B_or:	T_boolean → T_boolean
	B_if:	T_object → T_object → T_object
	B_and:	T_boolean → T_boolean
	B_xor:	T_boolean → T_boolean
T_character	*B_ord*:	T_natural
T_string	*B_car*:	T_character
	B_cdr:	T_string
	B_concat:	T_string → T_string
T_real	*B_succ*:	T_real
	B_pred:	T_real
	B_add:	T_real → T_real
	B_subtract:	T_real → T_real
	B_multiply:	T_real → T_real
	B_divide:	T_real → T_real
	B_trunc:	T_integer
	B_round:	T_integer
	B_lessThan:	T_real → T_boolean
	B_lessThanEQ:	T_real → T_boolean
	B_greaterThan:	T_real → T_boolean
	B_greaterThanEQ:	T_real → T_boolean
T_integer	Behaviors from T_real refined to work on integers	
T_naturals	Behaviors from T_integer refined to work on naturals	

Table A.3: Behavior signatures of the atomic types of the primitive type system.

Appendix B

Syntax of the TIGUKAT Language

< 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 specification >*
| *< function declaration >*
| *< association >*

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

< collection declaration >
: **create collection** *< new reference >*
type *< type reference >*
[**with** *< obejct variable list >*

< class declaration >
: **create class** [*< new reference >*] **on** *< type reference >*

< *behavior specification* >
: **add to** < *type reference* > < *behavior list* >
| **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 list* >
: < *public list* > < *private list* >

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

< *private list* >
: /* 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* >:
[< *first parameter* >] < *parameter list* >

< *first parameter* >:
self : < *type reference* > [,]

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

< *parameter* >:
identyfier : < *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* >

$: < object\ refernce >$

$< right\ side >$
$: < TQL\ Statement >$
$| < term >$