The COCOON Object Model

M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch

Foreword

This report has long been awaited—primarily by us, the authors, but also by a number of colleagues who kept asking for it. We had hoped to complete it earlier, but several factors caused the final preparation to take longer than expected: for example, the removal of some of us from Zurich to Ulm, and the inclusion of new ideas that entered our minds, hence requiring ever more changes. Finally, now that it's done, we must apologize to those who were optimistically promised a copy by the end of 1991!

COCOON as a research project was part of a "major research programme" initiated by DFG, the German Research Foundation. It was partially funded by DFG, from 1988–1989, and by SNF, the Swiss National Science Foundation, from 1989–1992. The project started at TU Darmstadt, moved to ETH Zurich, and was completed by the end of 1992 (in terms of funding) at University of Ulm. In that respect, this is the final report on the COCOON project. The work will be continued in Ulm and in Zurich.

The ideas that have influenced the COCOON model and its query and update language have been presented in several papers before. This report, therefore, is more of a "reference guide"; a place where the model is defined completely, formally, and—hopefully—consistently. The reader who has followed earlier publications on COCOON will notice that some details have been altered and, as a result, some definitions may be inconsistent with those given in previous papers. This report, however, accurately represents the current state of our work. As it is intended to be a "reference guide", we have not included a separate section on related work. A discussion of related approaches can be found in previous papers and throughout the text.

We thank our colleagues with whom we have had many lively discussions about all aspects of COCOON: the general ideas, the specific details, the formalization, the implementation, and the various papers. Also we thank the students, without whom the implementations would not have been possible. This report will be available as a Technical Report from the Department of Computer Science of ETH Zurich, and—as a reprint—from the Faculty of Computer Science of the University of Ulm.

Zurich and Ulm, December 1992

Hans-J. Schek, Marc H. Scholl

Authors' addresses:

M.H. Scholl, C. Laasch, C. Rich, M. Tresch

University of Ulm, Faculty of Computer Science
Databases and Information Systems
Oberer Eselsberg, P. O. Box 4066
D–W–7900 Ulm, Germany

{scholl,laasch,rich,tresch}@informatik.uni-ulm.de

H.-J. Schek

ETH Zurich, Department of Computer Science
Information Systems – Databases
CH–8092 Zurich, Switzerland

schek@inf.ethz.ch

This is a reprint – with kind permission – of Technical Report No. 192, Department of Computer Science, ETH Zurich, December 1992.

The COCOON Object Model

M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch

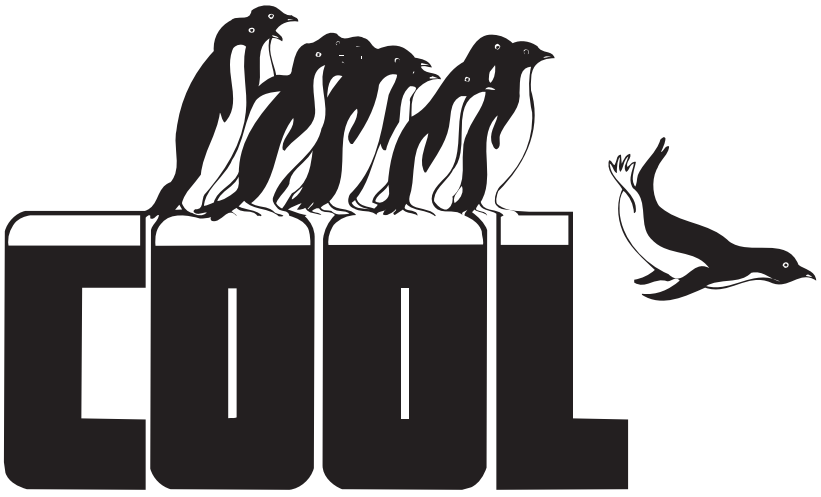
Abstract

The COCOON model was intended to extend the concepts of relational database management systems (DBMSs) beyond nested relational to object-oriented ones. Key characteristics of COCOON and its database language COOL are: generic, set-oriented query and update operators similar to relational algebra and SQL updates, respectively; object-preserving semantics of query operators, which allows for the definition of updatable views; update operations that keep model-inherent integrity constraints consistent; a separation of the two aspects of programming language "classes": type vs. collection; predicative description of collections, similar to "defined concepts" in KL-One-like knowledge representation languages; automatic classification of objects and views (positioning in the class hierarchy). This report gives a comprehensive introduction to the COCOON model and its language COOL as well as a formal definition. Our formalization uses denotational semantics, a popular technique in programming languages. We found that standard set-theoretic formalizations of data and object models were not equally well-suited to express update semantics. This, however, is essential for object-oriented as opposed to value-based languages. This can also be taken as an indication for the convergence of database languages to generell programming languages. Along these same lines, we emphasized static-type checking of COOL.

(Second revised edition January 1994)

Contents

Let's get into it ...



Chapter 1

Introduction

This report describes the object-oriented data model COCOON and its language COOL. The overall objective of COCOON[1] is to investigate the principal foundations and architecture of a cooperative object-oriented database system. The overall approach of COCOON can best be characterized by the term evolution instead of revolution [SS91], that is, the guideline has been to try to integrate, in a consistent way, concepts from other fields within computer science into the database context. Examples are structuring primitives from AI knowledge representation (we reviewed several techniques developed there and found that the KL-One direction was best suited for our purposes). Similarly, object-oriented concepts from programming languages had to be adopted for inclusion into databases. In contrast to many other OODBMSs, particularly most commercially available ones, our approach was not to extend an OOPL with persistence and transactions. Rather, we have emphasized the preservation of established DBMS advantages, such as data independence, set-oriented descriptive languages, optimizability, sometimes at the expense of expressiveness. As a result, COOL, the query and update language of COCOON, is not a (computationally) complete language.

COCOON is considered a *core* object model: we include only a limited number of concepts, aiming at an extensible model that facilitates inclusion of new (application-specific) concepts, such as new base types (geometry, text, graphics,...) or type constructors (tuples, sets,...). "Evolutionary" means that some of the *concepts* and *operations* of the object model have been carried over from the predecessor models; that is, rather than re-inventing known concepts, we try to integrate and extend them, if necessary.

The *key properties* of COCOON, as compared to other models and languages for object-oriented databases are:

- *object-preservation:* the central concept concerning the semantics of the query language. The results of queries are some of the already existing objects from the database. The other choices are object-generating or value-returning operations. Object preservation is crucial for many other key characteristics of COCOON,

[1] COCOON is a recursive acronym standing for COcoon... Complex-Object-Orientation based on Nested relations.

such as views and descriptive updates.

- *type/class separation:* a consequence of object preservation. If both projection and selection are to preserve objects, and if composite select/project queries are permitted, we need a type/class separation in order to give a proper description of the result of such a query.

- *multiple type instantiation and multiple class membership:* immediate consequences of object preservation. Objects may be instance of more than one type and member of many classes at the same time. Since projection, for example, changes the type, all objects in the result 'acquire' a new type, in addition to their original.

- *dynamic reclassification of objects during updates:* necessary when we take into account that objects can dynamically gain and lose types and class membership during their life time. Update operations may change the type and the classification of objects. (In COCOON, classes may be defined by a class predicate, so objects of the superclasses are automatically classified depending on their properties—cf. AI classification languages such as KL-One [BS85].)

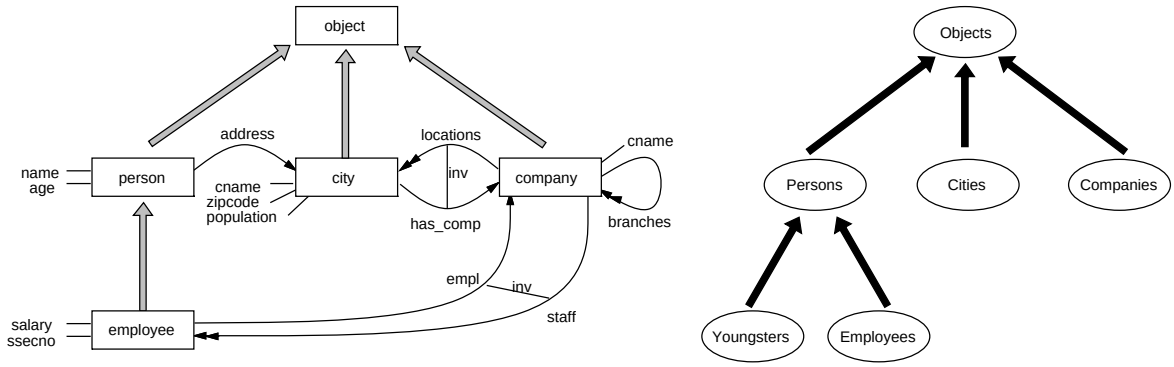
The report is organized as follows. Chapter 2 is an informal presentation of the model and its query and update language, in which we focus on the characteristics and objectives of the model without introducing formal details. A formal definition is given in Chapter 3. Chapter 4 describes how to add further concepts to the model. In Chapter 5 we sketch the directions of on-going work, including implementation considerations.

Chapter 2

Presentation of the Object Model

In this chapter, we present an informal tour through the COCOON object model. First, we describe the basic constituents of the model. Then, the COOL language is introduced, namely the object definition language, the query algebra, and the generic update operators. The presentation is based on the following running example.

EXAMPLE 1: The example database stores information about persons, cities, and companies. For each person the name, the age, and the address is stored. Employees



are specialized persons, with additional information about the salary, the social security number, and the company they work for. A company is described by a name, a location, a set of branch offices, and its staff, namely a set of employees. For each city, we have the city name, the zip code, the population, and all companies at this place.

The graphical representation shows types as rectangles and functions as lines with one arrowhead mean single-valued functions, with two arrowheads set-valued functions. Classes are drawn as ellipses. Subtype relationship is shown with gray arrows, whereas subclass hierarchy is indicated using black arrows. ◇

2.1 Basic Concepts (COOL–DDL)

COCOON is a so-called "object-function" model, similar to IRIS [WLH90] or DAPLEX [Day89], for example. The main constituents of the COCOON model are: *objects*, *functions*, *types*, and *classes*.

Objects are instances of *abstract object types* (AOTs, see below). They are pure abstractions, in the sense that none of the descriptive information "associated with" an object is considered to be "part of" the object in any sense. Rather, *functions* (or methods, see below) are used as the uniform abstraction of stored fields, computed attributes, and relationships.

Data (or **Values**) are distinguished from objects, similar to [Bee89]. They are instances of concrete *data types* (see below).

Functions model the AOT-specific operators. They are the abstraction of side-effect free retrieval functions, called *properties*, and update *methods* with side-effects, that is, the AOT-specific operators. Figure 2.1 below gives a taxonomy of COCOON functions. Properties are further separated into *stored properties* (attributes) and *computed properties*. The later ones are either *derived* by a COOL expression (derived property) or using any *foreign* programming language (e.g., Modula-2, C++). We do not want to distinguish stored and computed (derived) functions without side-effects here, since for many situations, it is unimportant whether a function value is stored or computed, e.g. for queries. Thus, we consider this terminology as a higher level of data independence to hide implementation facts from the logical database schema.

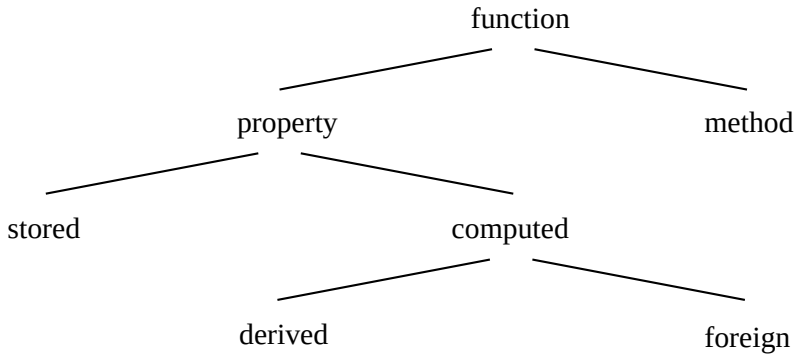


Figure 2.1: Taxonomy of COCOON Functions.

All functions are described by a name and signature (domain and range type). They are the interface operations of type instances. The implementation of functions is specified separately. The point behind our more abstract view is that we want to leave it up to the process of physical database design to make the decisions about what to store and what to derive (of course, there are restrictions on these decisions, so we

can mainly decide to materialize derived functions trading update effort for retrieval speed).

In order to express general relationships, functions may be set-valued. Furthermore, two functions may be defined as being inverses of each other. Of course, these integrity constraints are enforced by the system during updates. The example,

define function *name*: *person* → *string*;
define function *age*: *person* → *integer*;
define function *address*: *person* → *city*;
define function *empl*: *employee* → *company* **inverse** *staff*;
define function *staff*: *company* → **set of** *employee* **inverse** *empl*;

defines first three functions with domain type *person*, where *name* and *age* have primitive range types (*string, integer*), and the third one has an abstract range (*city*). The last two functions are inverses of each other, such that the constraint $x \in staff(empl(x))$ always holds.

Types are separated into (concrete) *data types* and (abstract) *object types*. Data types are either *primitive* (e.g., integer, real, string, boolean)[1] or *constructed* (e.g., set, tuple, function). Non-constructed types, namely primitive and abstract ones, are called atomic types. See Figure 2.2 below for a taxonomy of COCOON types. Object types describe the common interface of all instances of that type, the set of

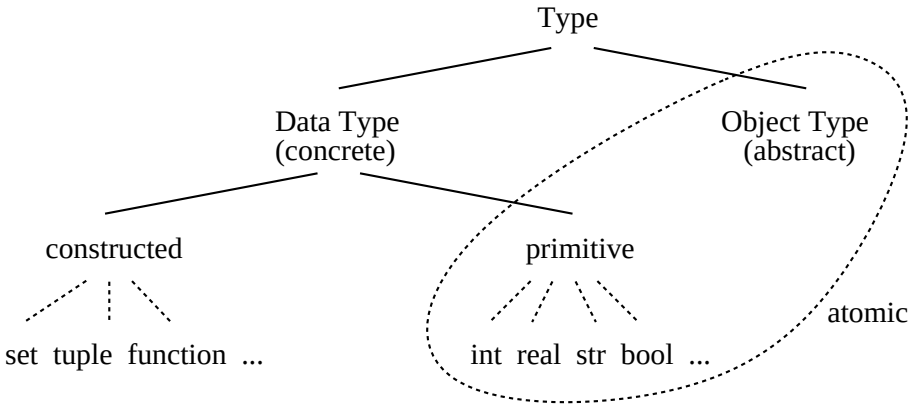


Figure 2.2: Taxonomy of COCOON Types.

applicable functions. So the definition of an object type usually consists of two parts: a type name and a set of functions.[2] The signature of the included functions can be defined separately, using **define function** ..., or together with the type definition. The example,

[1] In other OODMs, primitive types are also called printable, literal, or base types.

[2] As we will see later, queries can dynamically produce new types. These are unnamed, but their set of functions can be derived from the query by type inference rules.

define type *person* **isa object** = *name, age, address* ;
define type *city* **isa object** = *cname* : *string* ,
zipcode : *integer* ,
population : *integer* ,
has_comp : **set of** *company* **inverse** *location* ;
define type *company* **isa object** = *ident* : *string* ,
location : *city* **inverse** *has_comp* ,
branches : **set of** *company* ,
staff ;

defines three object types. The interface of the type *person* has three applicable functions, *name, age, address*, each of which was already defined above. In contrast, some of the functions of the types *city* and *company* are defined together with the type.

COCOON's query language is strongly typed, that is, a type checker (statically) guarantees that only type-valid expressions are ever executed.

Subtyping allows a new type to be defined as a subtype of an existing one. Every instance of a subtype is also an instance of its supertypes. This is called *multiple instantiation*. The complete definition of the subtype relationship ($\preceq$) is given in Chapter 3.

Abstract object types are arranged in an inheritance hierarchy (actually, due to multiple inheritance, not a strict hierarchy). The subtype hierarchy is essentially defined by the superset relationship between the sets of functions defined on the subtype as compared to its supertype(s):

$$t \preceq t' \iff \mathit{functs}(t) \supseteq \mathit{functs}(t')$$

A subtype inherits all the functions of all its supertypes and (usually) adds new ones.

define type *employee* **isa** *person* = *salary* : *integer* ,
ssecno : *integer* ,
empl ;

The new object type *employee* is defined as a subtype of *person*. Hence, it inherits its functions and adds new ones, *salary, ssecno, empl*. Each instance of type *employee* is also an instance of *person*. This reflects the second aspect of subtyping, namely the set inclusion between the associated domains (set of all possible instances). A type can also be viewed as an intensional description of the set of all possible instances (extension in AI terms, domain in DB terms). We will show later that these two intuitive meanings of "type" coincide.

Subtyping between abstract object types defines a partial order, forming a *type lattice*, such that for any two types their lowest upper bound and greatest lower bound is always defined. The top element of the lattice is the most general type **object** where no user-defined function is applicable (therefore, all instances of defined abstract types in the database are also instances of **object**), the bottom element is the type ($\perp$) that is associated with the set that includes all functions.

We allow multiple inheritance, that is, types may have more than one supertype. We assume that naming conflicts have already been resolved (for instance, by prefixing function names with type names).

Classes are strictly distinguished from types in the sense that types are interface specifications (a collection of functions), whereas classes are *containers* for objects of some type, see also [ACO85, Bee89]. So, each class, C, represents a (typed) set of objects and associates the member type, $mtype(C)$, to the objects in the set $extent(C)$. The extent of a class is manipulated (so as to include or exclude member objects) either explicitly by corresponding operations or implicitly by membership predicates on objects' properties. For example:[3]

define class *Persons* : *person* ;
define class *Cities* : *city* ;
define class *Companies* : *company* ;

The three classes *Persons, Cities*, and *Companies* are defined as containers for objects of type *person, city*, and *company*, respectively. So, for example, *mtype(Persons)* = *person*.

Classes represent polymorphic sets: Members may be instances of several other types, particularly subtypes, in addition to the member type. Type checking in the COOL language always refers to the unique member type of a class, since classes are the primary operands of COOL operations. Due to the separation of types and classes, there may be any number of classes for a particular type.

As the result of the design process it is most likely to obtain one class per type. However, it is possible to define several classes over the same type, that is, to distinguish between multiple collections of objects over that type. On the other hand, a type T might serve the only purpose to "factor out" commonalities of the subtypes, such that only collections of the subtypes, but not of T itself are explicitly maintained in a database (cf. opaque types, abstract classes in some OOPLs).

Unlike in many other models, we do not explicitly maintain "type extents" (active domains), that is, the sets of *all* instances for each type. Rather, classes are used as named, user-defined containers. However, we can always derive the active domains of a type T by a query on class *Objects* selecting those members that are of type T (see Sec. 2.2).

Subclassing. There are several choices how to define a subclass relationship. Depending on whether the member types of two classes are the same or one is a *subtype* of the other, and depending on whether the extent of one class is a *subset* of the extent of the other. That is, we have two known relationships to consider: subtype and subset. We will always distinguish carefully which one of them holds, since they are often correlated, but they need not be. We will speak of a subclass relationship $C_1 \sqsubseteq C_2$, iff for two classes the following predicate is true in any database state:[4]

[3] As a naming convention for this paper, type identifiers start with a small letter and are in singular, whereas class names are capitalized and in plural.

[4] A more precise definition of the subclass relationship is given in Section 3.3.3.

$mtype(C_1) \preceq mtype(C_2)$ **and** $extent(C_1) \subseteq extent(C_2)$.

Usually, at least one of the ordering relationships ($\preceq$ or $\subseteq$) will be proper. Continuing the example:

define class *Persons*: *person* **some** *Objects*;
define class *Employees*: *employee* **some** *Persons*;
define class *Youngsters*: *person* **some** *Persons* **where** *age* < 30;

Notice that the definition of *Persons* given earlier is equivalent to this one; *Objects* is the predefined root of the class hierarchy. *All* objects are contained in *Objects* (that is, for the **object** type, we *do* have an active domain). The class *Employees* is defined as a subclass of *Persons* with a more specific member type *employee*. The objects in *Employees* are some (possibly all) of the objects of class *Persons*: i.e., the subtype relationship between the member type of *Employees* and *Persons* is proper, and the extent of *Employees* is a subset of *extent(Persons)*.

The class *Youngsters* is also defined as a subclass of *Persons*. But here, the member types of both classes *Persons* and *Youngsters* are the same: *person*. The predicate given for class *Youngsters* is a constraint that all members of *Youngsters* have to be persons younger than 30 (that is, typically *Youngsters* will be a proper subset of *Persons*). The keyword **some** indicates that it is a necessary, but not sufficient, condition for members of *Persons* to become members of *Youngsters*. Changing the keyword **some** to **all** would indicate a necessary *and* sufficient condition: in this case the DBMS would automatically classify persons into the subclass, if the predicate evaluates to true.

In the remainder, we call classes specified with the keyword **all** / **some** as all-classes / some-classes, respectively. Notice that, unlike e.g. [O291, Kim89, SÖ90, SRH90], we define the extent of a class to include the members of all its subclasses.

Views are derived classes, where the member type and the extent are defined implicitly by a query expression[SLT91]. Thus, the extent of a view is usually not stored explicitly, but rather computed from the view-defining query.[5]

Views provide a specialized interface to some base objects. A user or an application programmer usually works only with a small portion of the global schema, a *subschema*, that is particularly tailored for the task performed. Such a subschema consists of a collection of base and/or view classes together with the functions defined on them.

Our view mechanism simply allows arbitrary queries to serve as view definitions, exactly as in relational DBMSs. For example:

define view *EmplView* **as project** [*name,empl*] (*Employees*) ;
define view *YoungEmpls* **as** *Youngsters* **intersect** *Employees* ;

The view *EmplView* represents the same objects as the class *Employees*, but only the two functions *name, empl* are applicable. The others, namely *age, address, salary, ssecno*, are hidden from the user. The second view illustrates that the set of objects can be restricted: *YoungEmpls* is defined as those objects that are in both classes, *Youngsters* and *Employees*, at the same time. Because the objects in the view belong

[5] The capability of materializing views is not considered in this report.

to both classes, all functions that are defined on either of the respective member types can be applied[6].

Views can be used as arguments for queries and updates, just as ordinary classes can. We will see that, in contrast to the relational model, only a few restrictions have to be imposed. In fact, all update operators have the same effect as if they were applied to the base class, because the views' extents are derived from them.

Variables. In order to be able to refer to objects and results of previous algebra expressions, we allow the use of variables. Variables are used as *temporary names* ("handles") for instances of any type, i.e., data (integer, ...), objects, sets of any type, or functions. They have to be declared with their type in the database language, such that compile-time type checking applies to variables too. For example,

var *john* : *person* ;
var *BigCities* : **set of** *city* ;

declares variables of type *person*, and **set of** *city*, respectively. Hence, the object variable, for example, can be assigned the result of an object creation into class *Persons* in order to refer to the new object later; the set variable can keep the result of a selection on the database class *Cities* (the operators are shown in the next sections):

john := **insert** [*name*:='*john*', *age*:=35] (*Persons*) ;
BigCities := **select** [*population* > 1000000] (*Cities*) ;

In the example, a new object of type *person* is created, inserted into the class *Persons*, and assigned to the object variable *john*. After that, the name and age functions for the new object are set to the value '*john*', and 35 respectively. Then, the query defines *BigCities* to have as value a subset of the persistent objects from the input class *Cities*.

2.2 Query Algebra (COOL–QL)

The COOL query language is an extension of the (nested) relational algebra. That is, there is an evolutionary path from relational via nested relational (NF^2) to object-oriented query languages (such as COOL) [SS91]. The following characteristics of the COOL algebra are preserved from the relational algebra (with some adaptions due to the modeling power of object-oriented models, see also [HS91]):

- **set-orientation:** the inputs and the outputs of the query operations are sets of objects at a time. Hence, query operators can be applied to extents of classes, set-valued function results, query results, or set variables. Similarly, also update operations should be applicable to sets of objects. Thus, the gap between the *navigational/one-object-at-a-time* paradigm of updates and the *set-oriented* query facility disappears. Additionally, updates as well as queries can be processed in a more efficient way, because the system gets more flexibility in executing the update request.

[6] The member types and the extents of views are derived from the query expressions.

- **genericity:** generic query and update operations are applicable to all types of objects in a database schema (in contrast to type-specific functions, such as methods in object-oriented models).

- **closure:** the result of a query is representable in the data model. This is important for many features such as composite queries, query optimization, and views. Similarly, update operations should respect model-inherent constraints.

- **orthogonality:** query operations should allow for an orthogonal combination of all operations (subject with respect to the type-constraints of operands of a query). Update operations should be combinable into complex sequences of updates *and* queries in order to realize non-trivial requests (with deterministic semantics). These update sequences can be used to define more powerful update units, by which (application-specific) integrity constraints are kept consistent.

Additionally, there are two characteristics of COOL, of which the former one is essential for features such as orthogonality, and views:

- **object preservation:** the operators have object-preserving semantics, such that the results of queries are (some of) the existing objects from the database. Other (algebraic) languages with object-preserving semantics are [SÖ90, HFW90, HS90]. The other choices were: object-generating operators (results are objects, but newly created ones), or value-generating operators (results are data values, i.e. tuples, and not objects). Examples of query languages with value-generating and/or object-generating semantics are [AK89a, ASL89, SZ89]. Since values-generation is useful for output purposes, the COOL language includes one such operator (**extract**, see below). Object-generation might be useful, too. In COOL, however, this is considered to be an update rather than a query.

- **static type-checking:** using a strongly typed algebra that allows for *static type checking* facilitates early detection of incorrect statements and reduces run-time effort. This is achieved by using the type associated with classes, sets, and function values to check whether the operations on the respective elements are legal. In order to allow more flexibility, COOL includes type guards [Dij75].

2.2.1 Object Preserving Operators

The effect of each query operation can be described by the type and the extent of the result set. Most of the operations change only one of them. Only the set operations **union**, **intersect**, **difference**, and **pick** have an effect on both.

Function application (f(e)) returns the value of function f applied to argument e. Syntactically, function application can also be written as f, if the argument is obvious from the context (see the selection example below).

Selection (**select** [*bool-expr*] (*set-expr*)) returns a subset of the input set of objects, namely those satisfying the predicate *bool-expr*. The type of the result set is the type of the input set.

Selection requires that the predicate is evaluable on the type of the input set. Predicates are built up in the standard way including arithmetic and set-comparison operators. For example, the following query

$$\textbf{select}\ [\emptyset \neq \textbf{select}\ [age < 18](staff)](Companies)$$

is a selection, where all companies are returned that have at least one employee, who is younger than 18. Notice that the selection predicate contains a nested selection operation, which illustrates the orthogonality. "*staff*" is the application of function *staff* to an argument c, the corresponding company object. Alternatively, we could use the more explicit syntax

$$\textbf{select}\ [\emptyset \neq \textbf{select}\ [age(s) < 18](s : staff(c))](c : Companies)$$

In addition, COOL provides type test predicates, every type name is also the name of a unary predicate. For example,

$$\textbf{select}\ [person(o)](o : Objects)$$

selects all instances of type person, that is, the active domain of this type. Notice that this selection might be a proper superset of $Persons$, if there were person objects not included in the class $Persons$.

Projection (**project** $[f_1, \ldots, f_n]$ (*set-expr*)) produces as output a set with a (usually new) type, a supertype of the input type: fewer functions are defined, only those given in the projection list $f_1, \ldots, f_n$. All objects of the input set are also elements of the output set (*object preservation*).

This operator is used in order to "hide" some function values from views (such as the salary component of employee objects). Consider as an example the following query:

$$\textbf{project}\ [name, age, address, ssecno, empl](Employees).$$

This projection returns the set of *Employees*, however, with a changed type, including all functions of the employee type, except *salary*.

Duplicate elimination, as in the relational case, is not an issue, since two objects are different even if they have some (or even all) function values in common. To our knowledge, none of the previous object algebras provided a means of projecting objects onto some of their functions while preserving their identity.

Extend (**extend** $[f_1 := expr_1, \ldots, f_n := expr_n]$ (*set-expr*)) defines new derived functions $f_1 := expr_1, \ldots, f_n := expr_n$. That is, it extends the type of the objects in *set-expr* by these new functions.

Obviously, each function name f_i must be different from all existing functions for the type of the input. The expressions $expr_i$ can be any legal arithmetic-, boolean-, or set-expressions. The result set contains exactly the same objects as the input, but a

new type, a subtype of the input type, is associated with it (all the old functions plus the new ones are defined on it).

Assume we are interested for each company in the set *local_empl* containing members of the staff that live in the city where the company is located. This function can be defined by the following operation:

extend [*local_empl* := **select** [*address* = *location*](*staff*)](*Companies*)

Set operations (**union, intersect, difference**) can be performed as usual, since the extent of classes are sets of objects.

With a polymorphic type system, we need no restrictions on operand types of set operations (ultimately, they are all sets of objects). The result type, however, depends on the input types.

The result type of the **union** operation is the lowest common supertype (in the lattice) of the input types; this is the (possibly unnamed) type on which only those functions are defined that are common to both input types. For example, the result type of

EmplView **union** *Youngsters*

contains the functions *name* and *empl*.

The **intersect** operation results in the greatest common subtype, which is the type that allows us to apply the union of the input types' functions. For example, the functions *age, name,* and *empl* can be applied to the young employees as a result of the following expression:

(**project** [*age*](*Youngsters*)) **intersect** *EmplView*,

Finally, the **difference** yields a subset of its first argument with the same type. In general, each **difference** operation can be expressed by a selection due to the following equivalence:

set-expr$_1$ **difference** *set-expr*$_2$ $\equiv$ **select** [$o \notin$ *set-expr*$_2$](o : *set-expr*$_1$)

Pick (**pick** (*set-expr*)) is provided to convert a singleton set into the only element (i.e., drop the spurious set braces). The result of applying **pick** to a set with more than one object nondeterministically selects one object. The result type is the element type of the set. For example the expression

pick (**select** [*name* = 'Smith'](*Persons*))

denotes the (or an) object of class *Persons* whose name is *'Smith'*.

Type guards (**guard** [*T, then-expr, else-expr*](e)) returns the value of *then-expr* applied to e, if e is of type T, otherwise *else-expr* is applied to e. This is a means of combining dynamic type tests with static type-checking: *then-expr(e)* is evaluated only if the dynamic test $T(e)$ returns true, but *then-expr* is statically type-checked under the assumption that e is of type T. For example,

select [**guard** $[employee, grant+salary > 2000, grant > 1000](s)](s : Students)$

selects all students who get a grant of more than 1000 or—if they are also employees—get a sum of grant plus salary greater than 2000.

These are the basic *object preserving* query operators of our algebra. Other operators, such as join can be derived from them. Informal presentations of the algebra have also been given in [SS90a, SS90b].

2.2.2 Value Generating Operators

To communicate with value-oriented systems and for output purposes, there is one query operator that generates sets of tuples (relations), see Section 4.

Extract (**extract** $[f_1, \ldots, f_n]$ (*set-expr*)) generates a set of tuples. For each object in *set-expr* a tuple is generated, where each function f_i in the extraction list is used to form one column.

For example, the expression

extract $[name, age](Persons)$

yields a relation with two columns, *name* and *age*, which contains a tuple for each object of the class *Persons*. Each tuple has two components that carry the result of applying the corresponding function to a person.

Since the **extract** operator just extracts properties of objects, the identification of the original objects is lost.

2.3 Update Operators (COOL–DML)

In OODBMSs updates are usually carried out by type-specific methods that are defined by the type implementor. These methods provide a means to offer integrity-preserving updates to *clients*, but there is no support for the method *implementor* as far as integrity preservation is concerned. Therefore, COOL provides generic update operations that maintain model-inherent integrity constraints, and make the implementation of methods simpler. Additionally, applications might make direct use of these update operations. Therefore, object manipulations that involve the generic modeling facilities of COCOON, do not necessarily need to be coded into methods over and over again. Rather, applications can use our generic operators directly where appropriate.

The update operators can be divided into four groups, the first three of which are defined according to the three modeling concepts (variables including functions, types, and classes): Assignments, operations for object evolution, and operations for manipulating the extents of classes and views. The last group are sequences of update operations that allow to keep integrity constraints consistent that are not covered by the modeling capabilities of the OODBMS: type-specific methods.

In the first place, all update operations are applied to single objects so as to obtain simple semantics. However, in order to apply set-oriented updates we provide a

descriptive iterator (**apply_to_all**[*upd-seq*](*set-expr*)) that takes a sequence of update operations *upd-seq* as a parameter, and executes it for each element of a set *set-expr* (e.g., a query result). Thus, we can not only apply single generic update operations in a set-oriented fashion, but also update sequences or type-specific methods.

2.3.1 Assignments

Assignments to Variables ($v := e$). The value of a variable v can be explicitly modified by an assignment. As usual for object-oriented languages, the inferred type of the right-hand side expression e can be a subtype of the variable's type. For example, the assignment

$p :=$ **pick** (**select** [$name$ = 'Smith'](*Persons*));

has the effect that (after the execution of the assignment) the variable p denotes an object with the name *'Smith'*.

Since functions are also regarded as variables, it is possible to assign a set of function pairs, which is produced by an appropriate expression, to functions. The effect would be that the function is redefined for all arguments at the same time. Typically, however, we want to redefine function values only for particular arguments. This is achieved by the following.

Partial Assignments to Functions (**set** [$f := e'$](e)).
The operation **set** changes the function f such that the function application $f(e)$ results in the value of the expression e'.

Continuing the example from above, we change the name of the person denoted by the variable p by a partial assignment to the *name* function from *'Smith'* to *'Jones'*:

set [$name :=$ 'Jones'](*p*).

Notice, however, that objects denoted by variables might change their function values, even though the variables are not used. This is due to object sharing, one of the most important properties of object-oriented models (see also object-oriented programming languages such as Eiffel [Mey88]). Consider the following example: Let the variable p denote a person, whose name is *'Jones'*; i.e., $name(p) =$ *'Jones'*. The execution of the two COOL statements

$q := p$;
set [$name :=$ 'Miller'](*q*)

yields a state, where p and q denote the same object, whose name is *'Miller'*. That is, the value of $name(p)$ changed from *'Jones'* to *'Miller'*, although the variable p was not changed explicitly. Let us now extend this example by considering the following assignment to the set-valued variable *jonesset*:

jonesset := **select** [$name$ = 'Jones'](*Persons*)

If we suppose that this operation was executed before the above operations, the variable *jonesset* contains all persons with the name *'Jones'*, especially the object denoted by the variables p and q. The "usual" semantics of such an assignment in

databases is the snapshot semantics, that the query is evaluated once and the result is assigned to the variable. This distinguishes set-variables from database views, which are reevaluated each time the view is used. However, the above **set** operation propagates to the elements in the variable *jonesset* in the same way as to p: I.e., after the **set** operation the variable *jonesset* still denotes the same set of objects, but their names might be different from *'Jones'*. Thus, in object-oriented models the snapshot semantics of assignments interacts in a subtle way with object sharing.

2.3.2 Operations for Object Evolution

Object Creation(**create** $[T](v)$).
The creation of an object by **create** $[T](v)$ instantiates type T and assigns the new object to the variable v (whose type has to be T or a supertype thereof). Notice that object creation involves "invention" of new OIDs, that is, the object (OID) assigned to v has to be different from all existing ones. [7]

In general, **create** $[T](v)$ changes the database state such that the active domains of all supertypes of T are extended by the newly created object. Thus, by making the created object an instance of T and all its supertypes, the subset semantics of the subtype relationship is maintained. [8]

For example, by the operation **create** $[employee](v)$ we instantiate an object of type *employee*, which is also an instance of type *person* because of the subtype relationship between the two types.

Dynamically Acquiring More Types (**gain** $[T](e)$).
The **gain** operation is used to dynamically establish new instance-type relationships between an object denoted by e and a type T. It does not change any values of variables or functions, but it adds the object denoted by e to the active domains of all supertypes of T.

For example this operation is needed in the well-known hire/fire scenario, when a person becomes an employee. In this case all functions that are applicable to employees should be applicable to the hired person, too. Therefore, the person denoted by the variable p has to become an instance of the type *employee* by the following operation:

gain $[employee](p)$.

A possible extension of the semantics could be to add the specification of default values for functions that now become applicable. Currently, none of the new functions gets a value, that is, they are all undefined ($\perp$) for the object e.

Dynamically Losing Types (**lose** $[T](e)$).
In contrast to the **gain** operation, **lose** deletes instance-type relationships. This is, the object denoted by e is removed from the active domains of T and all its subtypes. The effect of the operation is that all functions that are defined on the type T or a subtype of T are no longer applicable to the object denoted by e. Additionally, since

[7] Since we do not show OIDs to users, we do not have to insist on not reusing OIDs. Therefore, we do not have to keep track of OIDs of deleted objects.

[8] Here, we intentionally combined object creation with the instantiation of a type. A more minimalistic approach that separates both operations is defined in Chapter 3.

COOL is a strongly-typed language and objects might be shared, the semantics of the **lose** operation has to be defined with respect to all typing constraints of variables and functions. As a consequence, we have to remove each occurrence of the object denoted by e from variables, sets and functions, if they are typed as T or a subtype of T.

Assume that there are three variables, *empset* of type **set of** *employee*, *pset* of type **set of** *person*, and p of type *person*. Let the values of the variables be defined by the expressions:

$empset :=$ **select** $[salary > 100K](Employees)$;
$pset :=$ **project** $[name, age](empset)$;
$p :=$ **pick** (**select** $[salary > 100K \wedge name =$ 'Miller'$](Employees)$;

The variable p holds an object that is also element of both set variables. The difference between *empset* and *pset* is just on the type level, the object sets represented by them are the same. Miller's retirement by

lose $[employee](p)$

has the consequence that the object representing *'Miller'* is removed from the active domain of the type *employee*. Additionally, since functions that are defined on a subtype of *employee* must not be applied to the object that represents *'Miller'*, the object is also removed from the set denoted by *empset*, because the elements of this set have to be instances of type *employee*. Thus, the sets denoted by *pset* and *empset* become different, since *'Miller'* is still an element of *pset*, but not of *empset* anymore.

These proposed semantics guarantee that static type checking is sufficient despite operations that change types of objects dynamically. More intuitively, this kind of semantics implements the point of view that type information is part of the constraints that every valid database state has to fulfill. Once such constraints fail to hold, the state is changed by removing the objects from variable values.

As already mentioned above, we need no explicit **delete** operation, since its functionality is subsumed by the **lose** operation as follows:

delete $(e) =$ **lose** [**object**](e)

This is, an object denoted by e is destroyed if it loses its most general type *object*.

2.3.3 Manipulating the Extents of Classes and Views

The operation **add** $[e](C)$ is provided to add an already existing object denoted by e into the extent of the class C. According to the subset-semantics of the subclass relation, the object also becomes a member of C's superclasses.

For example, by the operation

add $[p](Employees)$

the object denoted by p becomes a member of the class *Employees*. Due to the subclass relationship, the object is also added to the class *Persons*.

An object denoted by e can be removed from the extent of the class C by the operation **remove** $[e](C)$. This operation has no effect on the existence of the object, but only on the membership relations between the object and classes. That is, the

object is not only removed from the extent of the class C, but also from those of C's subclasses, since these have to be subsets of C's extent.

Continuing the above example, the object denoted by p can be removed from the class *Employees* by the following operation:

remove $[p](Employees)$

If there are subclasses of *Employees*, p would also be removed from their extents. However, the object remains in the extent of the class *Persons*.

A detailed discussion of the semantics of the operations **add** and **remove** with respect to class predicates and views can be found in Section 3.3.4.

Chapter 3

Formalization of COCOON

The formalization of the COCOON model is carried out as follows: First, we formally define a subset of the COCOON model, which consists of objects, functions, types, and variables and includes generic update operations besides a query algebra. Afterwards the remaining COCOON concepts such as classes, views, and inverse functions as well as some derived operations are defined by a mapping onto the formal model.

We include updates so as to clearly define their semantics, and because important concepts such as object-sharing can only be clarified this way. Because the update operations applied to *objects* have effects that are usually not restricted to one set (due to essential object oriented features like inheritance and sharing), we do not use set-theory for the formalization, where updates are incrementally defined by adding or removing tuples from a relation, respectively. Instead, we use denotational semantics that allows us to specify the effects of generic update operations more conveniently. For example, deletion of an object requires appropriate actions on other objects in order to manage the references to the deleted object as well as to remove the object from variables.

In this chapter we present the formal model by defining its syntax, the state of a database, and the semantics of the type system and the operations. Finally, we map classes and views with their operations **add** and **remove** onto the formal level and define inverse functions.

3.1 A Formal Model

The formal model consists of only a few basic concepts. Nevertheless, they are sometimes more flexible and powerful than necessary in COCOON. This is because the formal model is fully orthogonal, which makes the definitions easier. The overall decisions in designing this model are the following:

- **Static Type-Checking.** Since we are interested in languages, which can be statically typed-checked, the model is based on a type system. This type system should at least allow us to specify COCOON's single- and set-valued functions.
- **Hiding Object-Identity.** If the identity of objects is not explicit for users, handles are needed in order to refer to objects (e.g., newly created ones). Therefore

we integrate variables.

We now define the type system and type-valid (query-)expressions and update operations.

Type Expressions.
Types can be atomic or constructed (see [HK87]). The set of atomic types (ι) contains predefined (or *printable*) ones such as INTEGER, BOOL (denoted by $\iota_{Int}, \iota_{Bool}$) and one type that represents the countable set of objects ι_{Object}. Object types need not be named, they are denoted by function labels ($[f_1, ..., f_n]$). Type constructors are function ($\rightarrow$) and set ($\{\ \}$). So, the following type expressions τ are provided:

$$
\begin{array}{lll}
\tau & = & (\ \tau\ ) \qquad\qquad /* \text{ types } */ \\
& | & \iota_{Int} \qquad\qquad /* \text{ Integers } */ \\
& | & \iota_{Bool} \qquad\qquad /* \text{ Booleans } */ \\
& & \vdots \\
& | & \iota_{Object} \qquad /* \text{ Objects } */ \\
& | & [f_1, ..., f_n] \qquad /* \text{ object types } */ \\
& | & \{\ \tau\ \} \qquad\qquad /* \text{ set types } */ \\
& | & \tau \rightarrow \tau \qquad /* \text{ function types } */
\end{array}
$$

Notice that this type system is more general than we actually need (e.g., sets of sets can be defined). We do not restrict it, though, in order to get a homogeneous, orthogonal type system for functions and variables, that can easily be formalized. Also, the operations we will define, focus on functions on objects ($\iota_{Object} \rightarrow \tau$) and sets of atomic types, but their definitions take the full type system into account. We omit type names, which can be regarded as abbreviations for the sets of applicable functions.

Expressions Including Query Operations.
Except of some minor changes according to the functional flavour of the formal level, most query operations of Section 2.2 are carried over.

$$
\begin{array}{lll}
e & = & (\ e\ ) \qquad\qquad\qquad\qquad /* \text{ expressions } */ \\
& | & v \qquad\qquad\qquad\qquad /* \text{ variables } */ \\
& | & c \qquad\qquad\qquad\qquad /* \text{ constants } */ \\
& | & \textbf{new}\ () \qquad\qquad\qquad /* \text{ creating objects } */ \\
& | & \textbf{adom}\ ([f_1, ..., f_n]) \qquad /* \text{ active domains of object types } */ \\
& | & \{\ e\ \} \qquad\qquad\qquad /* \text{ sets } */ \\
& | & \textbf{pick}\ (e) \qquad\qquad\qquad /* \text{ pick one element of a set } */ \\
& | & \lambda x : \tau.\ e \qquad\qquad\qquad /* \text{ function expression } */ \\
& | & f\ (e) \qquad\qquad\qquad /* \text{ function applications } */ \\
& | & \textbf{guard}\ [\tau, e, e]\ (e) \qquad /* \text{ guarded function applications } */ \\
& | & e \cup e \qquad\qquad\qquad /* \text{ unions } */ \\
& | & e \cap e \qquad\qquad\qquad /* \text{ intersections } */ \\
& | & \textbf{select}\ [x.e]\ (e) \qquad /* \text{ selections } */ \\
& | & \textbf{project}\ [f_1, ..., f_n]\ (e) \qquad /* \text{ projections } */ \\
& | & \textbf{let}\ x = e\ \textbf{in}\ e\ \textbf{end} \qquad /* \text{ name bindings } */
\end{array}
$$

Besides the above expressions we make use of the standard operations for integer and boolean. From [Zan84] we adopt the semantics for the comparison operations ($=, \neq, <, \leq, >, \geq$), which is based on three-valued logic.

Instructions.
On the formal level, we only define the following update operations:

$$\begin{array}{lll} i & = & (\ i\ ) \qquad\qquad\quad /* \text{ instructions } */ \\ & | & v := e \qquad\qquad\quad /* \text{ setting variables } */ \\ & | & \textbf{set}\ [f := e](v) \quad /* \text{ setting function values } */ \\ & | & \textbf{new}\ () \qquad\qquad\quad /* \text{ creating objects } */ \\ & | & \textbf{gain}\ [\tau](e) \qquad /* \text{ adding types to objects } */ \\ & | & \textbf{lose}\ [\tau](e) \qquad /* \text{ removing types from objects } */ \end{array}$$

Additionally, we define the following control flow instructions for the sequential and set-oriented application of instructions:

$$\begin{array}{lll} i & = & i\ ;\ i \qquad\qquad\quad /* \text{ composition of instructions } */ \\ & | & \textbf{apply\_to\_all}\ [i](e) \quad /* \text{ set-oriented application } */ \end{array}$$

The remaining instructions of Section 2.3 are mapped onto (sequences of) these instructions later.

3.2 Semantic Framework of COOL

In the denotational approach to the definition of language semantics, (higher order) functions play a major role. A "semantic domain" is defined that represents interpreted constructs used as the "denotation" (semantics) of syntactic constructs. For each syntactic construct, such as constants, expressions and statements, a function is given that maps syntax to semantics. In particular, for updates the target of this denotation function is again a function (from an old to a new state). The semantics of the formal level is defined by five such denotation functions that are successively introduced in the next sections:

- the function σ models the database state;
- the function A is used to represent the type declarations of variables and functions;
- the function $[\![\ ]\!]$ is applicable to type expressions and yields their semantics;
- the function $E[\![\ ]\!]_\sigma$ can be applied to expressions and returns their value relative to the state σ;
- the function $U[\![\ ]\!]_\sigma$ represents the state transitions of the state σ resulting from instructions.

3.2.1 Formalization of the State

The appropriate definition of the "database state" is crucial to all formalizations of database updates. In our approach, the database state must contain the following information:

(i) for each object type the current set of instances,

(ii) the values of all possible function applications,

(iii) the values of variables.

Due to the object-function approach, we do not model an internal state of an object directly. Rather, in accordance with the denotational approach (e.g., [Sto77]), all information about the state is represented by a function. That is, the state function σ can be applied to object types, functions, and variables. It returns the current sets of instances, the sets of pairs representing functions, and the values that are bound to variables, respectively. Formally, we consider functions as variables that are declared over function types.

In order to specify the domain of the function σ, we introduce the following sets: X denotes the set of variable names and includes the set F that contains function names. Object types are denoted by the elements of the powerset 2^F, using square brackets. Thus, the domain of names N is defined by the coalesced sum of $X + 2^F$.[1]

The range V of the function σ denotes the semantic domain of values that can be used in states. It is defined by the following recursive domain equation using the domain operators sum (+), and continuous function space ($\rightarrow$).

$$V = B_{Bool} + B_{Int} + B_{String} + B_{Object} + F + S + W$$
$$B_{Bool} = \{true, false\},$$
$$B_{Int} = \{0, 1, -1, 2...\},$$
$$B_{String} = \{"a", "A", ...\},$$
B_{Object} contains countably infinite objects,
$$F = B_{Object} \rightarrow_{fin} V,$$
$$S = P_{fin}(V),\text{and}$$
$$W = \{\perp, \omega\}.$$

B_i are domains of basic values (e.g., boolean, and integer). F denotes the domain of finite mappings from B_{Object} to V, and S all finite powersets over V. W is included in V in order to specify errors. Type errors are indicated by ω, undefined values by the bottom element ($\perp$). In order to improve readability we omit the type information and use $\perp$ instead.

The formalization of the database state is simply a function σ from names N to values V, i.e., the signature of the function σ is $\sigma : N \rightarrow V$.

3.2.2 Static Typing

As a requirement for static type-checking, each variable v must be declared with a type τ by a variable declaration: **define var** $v : \tau$. These declarations are collected in

[1] Notice that $[f]$ denotes an object type, whereas f denotes a function.

the meta function A that returns a type expression if applied to a variable. Because the function A does not depend on the database state, it can be used for static type checking. For ease of presentation we extend the function A such that it returns the type $\{\iota_{Object}\}$, if applied to abstract object types (elements of 2^F).

Besides the effect on the function A, variable declarations generate the sets X and F; e.g., the variable declaration **var** $v : \tau$ adds the element v to the set X (and F, if $\tau = \iota_{Object} \to \tau_2$). Notice, however, that the sets X and F do not depend on the state. Thus, they are constant during the evaluation of expressions and update operations.

3.2.3 Semantics of Type Expressions, Subtyping

Except for object types, our semantics of types and subtyping is quite usual and follows [BdV91, BF91, MCB90]: i.e., the denotations of basic types are given by the following equations:

$$\begin{aligned}
&[\![\, \iota_i \,]\!] = B_i, \text{ in case that } B_i \text{ is a summand of } V \\
&[\![\, \{\ \tau\ \} \,]\!] = \{x \in S \mid x \subseteq [\![\, \tau \,]\!]\} \ \in\ V \\
&[\![\, \tau_1 \to \tau_2 \,]\!] = \{f \in F \mid x \in [\![\, \tau_1 \,]\!] \implies f(x) \in [\![\, \tau_2 \,]\!]\ \} \ \in\ V.
\end{aligned}$$

The subtype relationship ($\preceq$) is based on set inclusion: i.e., if a type is defined as a subtype of another, then every instance of the subtype is also an instance of its supertype (which allows *substitutability*): $\tau_1 \preceq \tau_2 \implies [\![\, \tau_1 \,]\!] \subseteq [\![\, \tau_2 \,]\!]$. This leads to the following inference rules for constructed types:[2]

$$\text{[SETS]}\ \frac{\tau_1 \preceq \tau_2}{\{\tau_1\} \preceq \{\tau_2\}} \qquad\qquad \text{[FUNS]}\ \frac{\tau_1^{dom} \preceq \tau_2^{dom},\ \tau_2^{rng} \preceq \tau_1^{rng}}{\tau_2^{dom} \to \tau_2^{rng} \preceq \tau_1^{dom} \to \tau_1^{rng}}$$

Here and in the following we use a common notation for type inference rules. For example the rule [SETS] is to be read as "if τ_1 is a subtype of τ_2, then $\{\tau_1\}$ is a subtype of $\{\tau_2\}$".

Let us now focus on the semantics of object types. In Section 3.2.3 we argued to use the basic type B_{Object} as the domain of all functions, such that function signatures are homogeneous. Then, however, type-checking is less meaningful, because each function may be applied to any object. The more specific function domains are, the more errors are detected at compile-time. Therefore, we use object types (denoted by $[f_1, ..., f_n]$) as subtypes of B_{Object}. In particular, there is a object type $[f_1, ..., f_n]$ for any subset of F that contains the functions defined in a database schema. In order to get concise type inference rules in Section 3.2.4, the object types are arranged to a lattice. Intuitively, applications of the function f on instances of $[f_1, ..., f_n]$ pass static type-checking, iff f is contained in $\{f_1, ..., f_n\}$.

Nonetheless, the *semantic domains* of all object types are the same, in order to allow for object evolution, such that objects can gain and lose instance relationships dynamically.

In contrast to instances of data types like integers, objects can be created and deleted. That is, it is not possible to refer to arbitrary instances of object types, rather

[2] The horizontal bar corresponds to logical implication. Notice the antimonotonicity (contravariance) in [FUNS], which is needed for the set-inclusion semantics of the subtype relationship.

only to those that have been created and not yet deleted. Therefore, the domains of object types (denoted by $[\![\,[f_1, ..., f_n]\,]\!]$) have to be distinguished from the *active domains* (denoted by $\sigma([f_1, ..., f_n])$) that contain the instances of these types in the current state σ.

Thus, we use the state function in order to refer to active domains already in this section, though its definition is given in Section 3.2.5 by the semantics of operations that manipulate the active domain. This is done, because the notion of active domains is not only important for the restriction of function types (as discussed above), but also for the definition of subtyping on object types. Notice that only the active domains of the type without functions ($[\,]$) and the types with only one function ($[f_i]$) are explicitly maintained. The other active domains (of types with more functions $[f_i, f_j, \ldots]$) are derived from the former according to the type lattice (see below).

There are two requirements for the definition of the active domains:

- the active domain has to be a subset of the domain in any state: [3]
  $\sigma([f_1, ..., f_n]) \subset [\![\,[f_1, ..., f_n]\,]\!]$.
- the subtype relationship has to capture the substitutability of objects:
  $[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \implies \sigma([f_1, ..., f_n]) \subseteq \sigma([f'_1, ..., f'_m])$

The second requirement adapts the usual notion for subtyping to object types, which implies a subset relationship on the respective domains (see above). In particular, this adaptation is important, because the domains of object types have to be the same in order to allow for object evolution.[4]

After this motivation of object types, let us now define the semantic domains and subtyping for object types.

$$[\![\,[f_1, ..., f_n]\,]\!] = [\![\,[\,]\,]\!] = B_{Object}, \text{ for } \{f_1, ..., f_n\} \subseteq F.$$

The subtype relationship between object types is defined on the superset relationship between their function sets, as follows:[5]

$$[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \iff \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\}.$$

In order to make type-inference more concise, object types are arranged in a lattice:

> The set of object types forms a lattice, where the subtype relationship is the partial order. The least upper bound ($\sqcup$), and the greatest lower bound ($\sqcap$) are defined as follows:
> $[f_1, ..., f_n] \sqcup [f'_1, ..., f'_m] = [f''_1, ..., f''_l]$,
> where $\{f''_1, ..., f''_l\} = \{f_1, ..., f_n\} \cap \{f'_1, ..., f'_m\}$.
> $[f_1, ..., f_n] \sqcap [f'_1, ..., f'_m] = [f''_1, ..., f''_l]$,
> where $\{f''_1, ..., f''_l\} = \{f_1, ..., f_n\} \cup \{f'_1, ..., f'_m\}$.

[3] Because all active domains are finite sets, the subset relationship is always proper.

[4] In case that objects are not allowed to become instances of arbitrary other types as in the case with "abstract types" in [HK87] or "clusters" in OSCAR [HFW90], our approach can be extended to different sub-lattices of objects. This would require more complex semantics of set operations, since both input sets have to belong to the same cluster (see also Sec. 5.3).

[5] Notice that this definition captures inheritance of interfaces in the usual way.

Finally, let us define the active domains of object types containing *more than one* function dependent on the types with exactly one function. (The active domains of object types with none or one function are defined in Section 3.2.5).

$$\sigma([f_1, ..., f_n]) \;=\; \bigcap_{f \in \{f_1,...,f_n\}} \sigma([f]).$$

Notice that this definition guarantees the subset relationship between the active domains of a subtype and its supertypes (see the second requirement for active domains above), because of the superset relationship between their function sets:

$$[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \Longrightarrow \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\}$$
$$\Longrightarrow \sigma([f_1, ..., f_n]) \subseteq \sigma([f'_1, ..., f'_m]).$$

3.2.4 Semantics of Expressions

In order to specify expressions we restrict expressions to syntactically correct ones; this is done by type inference rules. Because type inference rules include preconditions and the result type of expressions, we leave them out of the semantic denotations.

In contrast to [CW85] the assertions of variable declarations (A) and of subtyping information, which are of course used to infer the premisses, are omitted for the sake of readability.

The semantics of expressions is defined by the function E that returns the value $E[\![\, e \,]\!]_\sigma$ of the evaluation of expression e in the current state σ. That is, $E \in \; Exp \rightarrow \Sigma \rightarrow V$ is the semantic function where Exp are syntactic expressions, Σ are states, and V the semantic domain.

In the following we list the type inference rules and semantic denotations of expressions. The semantics of constants (c) and variables (v), with type information represented by the function A, are straightforward:

$$c :: \tau \qquad\qquad E[\![\, c \,]\!]_\sigma \in [\![\, \tau \,]\!]$$

$$\frac{A(v) = \tau}{v \;::\; \tau} \qquad\qquad E[\![\, v \,]\!]_\sigma = \sigma(v) \in [\![\, \tau \,]\!]$$

New creates new objects. That is, the application of the operation **new** yields an object that does not yet occur in the current state of the database. Formally, the "invention" of objects is non-deterministic, such that $E[\![\;]\!]$ is a mapping rather than a function (as claimed above). This non-determinism could be eliminated, for example, by assuming an order on objects (such that **new** yields the "smallest" not yet created object). However, since the object identities are not visible on the formal level, we do not need to care about different states that are isomorphic up to renaming object identities (see the notion of O-isomorphisms in [AK89b]). Therefore, we take the freedom to allow the non-deterministic invention of object identities: This is certainly desirable on the implementation level, furthermore it allows **new** operations to be executed in parallel within an **apply_to_all** statement [LS93].

$$\textbf{new}\;() :: \iota_{Object} \qquad\qquad E[\![\, \textbf{new}\;() \,]\!]_\sigma = o \;\textbf{with}\; o \in [\![\, [\,] \,]\!] \wedge o \notin \sigma([\,]).$$

Active domains of object types are returned by the operation **adom**:

$$\frac{[f_1,...,f_n] \preceq \iota_{Object}}{\mathbf{adom}\ ([f_1,...,f_n])\ ::\ \{[f_1,...,f_n]\}} \qquad E[\![\,\mathbf{adom}\ ([f_1,...,f_n])\,]\!]_\sigma = \sigma([f_1,...,f_n]).$$

Sets can be constructed by including an expression in braces:[6] [7]

$$\frac{e :: \tau}{\{e\} ::\ \{\tau\}} \qquad E[\![\,\{\ e\ \}\,]\!] = \{E[\![\,e\,]\!]\}$$

Pick is used to deconstruct sets, i.e., to get rid of the braces in case of singleton sets. Notice that **pick** is not deterministic in case of sets with more than one element. In order to avoid a non-deterministic semantics in case that the set contains more than one object, we could restrict the applicability (which, however, can only be checked at run-time) or assume an order on objects (see above):

$$\frac{e :: \{\tau\}}{\mathbf{pick}\ (e) ::\ \tau} \qquad E[\![\,\mathbf{pick}\ (e)\,]\!] = \begin{cases} v\ \in\ E[\![\,e\,]\!] & \text{if } E[\![\,e\,]\!] \neq \emptyset, \\ \bot & \text{otherwise.} \end{cases}$$

Lambda abstractions define functions on the active domains of object types, where $\sigma' = \sigma\{v/x\}$ denotes the substitution of v for x, i.e., it is identical to σ except that $\sigma'(x) = v$. This form of abstractions does not cause problems w.r.t. to the equality test on functions, because the free variable x is restricted to the active domain of the respective object type. Thus, functions can be regarded as finite sets of pairs[8]. The type inference rule means that $e :: \tau_2$ can be inferred under the assumption that x is a variable of the object type τ_1:

$$\frac{A(x) = \tau_1, \tau_1 \preceq \iota_{Object} \vdash\ e :: \tau_2}{\lambda x : \tau_1 .\, e ::\ \tau_1\ \rightarrow\ \tau_2}\ E[\![\,\lambda x : \tau_1 .\, e\,]\!]_\sigma = \{\langle v, E[\![\,e\,]\!]_{\sigma\{v/x\}}\rangle \mid v\ \in\ [\![\,\tau_1\,]\!]\}$$

Function applications return the function value if defined. Notice that there is no difference whether a tuple with the null value ($\bot$) as second component is included in the set of function tuples or not:

$$\frac{f ::\ \tau_1\ \rightarrow\ \tau_2,\ e :: \tau_1}{f(e) ::\ \tau_2} \qquad E[\![\,f\ (e)\,]\!] = \begin{cases} v_2 & \text{if } \langle E[\![\,e\,]\!], v_2\rangle\ \in\ E[\![\,f\,]\!], \\ \bot & \text{otherwise.} \end{cases}$$

Guarded function applications allow to apply functions that are defined on subtypes, if the argument is instance of this subtype. Therefore, the expression e_1 is type-checked under the assumption that e is instance of τ_1:

$$\frac{e :: \tau, x_1.e_1 :: \tau_1 \rightarrow \tau_1', x_2.e_2 :: \tau \rightarrow \tau_2'}{\mathbf{guard}\ [\tau_1, x_1.e_1, x_2.e_2](e) :: \tau_1' \sqcup \tau_2'}$$

$$E[\![\,\mathbf{guard}\ [\tau_1, x_1.e_1, x_2.e_2](e)\,]\!] = \begin{cases} E[\![\,e_1\,]\!]_{\sigma\{E[\![\,e\,]\!]/x_1\}} & \text{if } E[\![\,e\,]\!]\ \in \sigma(\tau_1), \\ E[\![\,e_2\,]\!]_{\sigma\{E[\![\,e\,]\!]/x_2\}} & \text{otherwise.} \end{cases}$$

[6] Notice that we require that $\{\bot\} = \emptyset$.

[7] Since the current state σ is used only as an input parameter to the function E, it is sometimes left out in order to improve readability.

[8] One might argue that functions are total, because they yield the $\bot$ value except for a finite set of arguments. Notice, however, that there is no means to refer to instances of ι_{Object} that are not contained in the active domain.

Unions of two sets result in a set that is associated to the least upper bound of the element types:

$$\frac{e_1 :: \{[f_1, ..., f_n]_1\}, e_2 :: \{[f_1, ..., f_n]_2\}}{e_1 \cup e_2 :: \{[f_1, ..., f_n]_1 \sqcup [f_1, ..., f_n]_2\}} \quad E[\![e_1 \cup e_2]\!] = E[\![e_1]\!] \cup E[\![e_2]\!]$$

Intersections of two sets are related to the greatest lower bound of the element types:

$$\frac{e_1 :: \{[f_1, ..., f_n]_1\}, e_2 :: \{[f_1, ..., f_n]_2\}}{e_1 \cap e_2 :: \{[f_1, ..., f_n]_1 \sqcap [f_1, ..., f_n]_2\}} \quad E[\![e_1 \cap e_2]\!] = E[\![e_1]\!] \cap E[\![e_2]\!]$$

Selections are used to specify subsets according to predicates: [9]

$$\frac{\lambda x : \tau.\, e_1 :: \tau \to \mathbf{bool}, e_2 :: \{\tau\}}{\mathbf{select}\,[x.e_1](e_2) :: \{\tau\}} \qquad E[\![\mathbf{select}[x.e_1](e_2)]\!] = \{y \in E[\![e_2]\!] \mid E[\![e_1]\!]_{\sigma\{y/x\}}\}$$

Projections restrict the interfaces of set elements as in the relational algebra (see also transformational filters in [AB88]). In contrast to projections in [DD91, SZ90, SÖ90] that generate objects or values, **project** is defined with object preserving semantics [SS90a]. Therefore, it can be used for hiding information, similar to assignments of instances to variables of a supertype:

$$\frac{e :: \{[f'_1, ..., f'_m]\},\ [f'_1, ..., f'_m] \preceq [f_1, ..., f_n]}{\mathbf{project}\,[f_1, ..., f_n]\,(e) :: \{[f_1, ..., f_n]\}} \qquad E[\![\mathbf{project}\,[f_1, ..., f_n]\,(e)]\!] = E[\![e]\!]$$

Name Bindings allow for a macro mechanism, such that an expression e_2 can be substituted in e_1 by a name. In particular, this macro mechanism can be used, for example, to express joins by virtual "'functions"':[10]

$$\frac{e_2 :: \tau_2, A(f) = \tau_2 \vdash e_1 :: \tau_1}{\mathbf{let}\ f = e_2\ \mathbf{in}\ e_1\ \mathbf{end}\ :: \tau_1} \qquad E[\![\mathbf{let}\ f = e_2\ \mathbf{in}\ e_1\ \mathbf{end}\ ]\!]_\sigma = E[\![e_{1\{e_2/f\}}]\!]$$

Let us illustrate by the following two examples how joins can be expressed. In the first example, we are looking for employees that are managed by their parents. In order to improve readability, we use COOL type names (such as *Empl*) instead of the respective BCOOL function sets ($[name, age, children, sal, manager]$). In the relational algebra this query would involve the join operation (if the schema fulfills at least third normal form). However, in object algebras composition of functions can be explored:

$$\mathbf{select}\ [x.x \in children(manager(x))](\mathbf{adom}\ (Empl)).$$

In the second example we make use of a virtual function for a more complex join.[11] All employees that have the same manager as x are collected in the function *colleagues*,

[9] This captures also the set difference that can be expressed by a predicate that checks whether an object is not contained in a set.

[10] The type inference rule means that $e_1 :: \tau_1$ can be inferred under the assumption that f is a variable of type τ_2.

[11] See [SS90a] for a discussion of different alternatives for joins: symmetric tuple/object generating; asymmetric as functions.

whose scope is limited by the **let** operation. If we are interested in all employees who have at least one colleague, the virtual function *colleagues* can be used in the subsequent selection:

$$\textbf{let}\ colleagues = \lambda x : [manager].\ \textbf{select}\ [y.\ manager(y) = manager(x) \wedge x \neq y](\textbf{adom}\ (Empl))$$
$$\textbf{in select}\ [x.\ colleagues(x) \neq \emptyset](\textbf{adom}\ (Empl))\ \textbf{end}\ .$$

Usually, operations of an algebra are orthogonal to each other, such that non of them can be defined by the others. However, up to now the operation **project** can be defined by other operations as follows:

$$\textbf{project}\ [f_1, ..., f_n](e) \equiv \textbf{select}\ [x.x \in e](\textbf{adom}\ ([f_1, ..., f_n]))$$

The reason for this lack of orthogonality is that casting the interface of objects has been bundled together with set-orientation. The extension of the relational algebra with object types (that are arranged in a lattice) needs active domains. These can be used together with operations like $\cap$ and $\cup$ that are defined with respect to the type lattice. Therefore type inference is more powerful than in relational algebra, in which set operations require the same schema on the input relations. That is, the problem originates from the extension of the relational algebra, in which operations are defined on sets, because of the potential for optimization. However, if not only sets but also other type constructors are integrated into the model, **project** is decoupled from sets, such that it only casts the type of a single object (see Sec. 4).

3.2.5 Semantics of Update Operations

The semantics of update operations can be defined by a function U that maps the old state σ, dependent on an update operation *upd-op*, onto the new one: $U[\![\ upd\text{-}op\ ]\!]_\sigma$. Update operations are defined only if certain typing restrictions are fulfilled. Notice that these preconditions can be verified by the static type-checker using type-inference rules. This results in a typing $e :: \tau$ of each expression e.

Assignment. Variables can be bound to new values by an assignment (:=). The new state is the same as the old one for all variables, types, and functions except for the variable v. The precondition ensures the substitutability of v's value:

If $v \in X$ and $e :: \tau$ and $\tau \preceq A(v)$, then

$$U[\![\ v := e\ ]\!]_\sigma(\phi) = \begin{cases} E[\![\ e\ ]\!]_\sigma & \text{if } \phi = v, \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

Since functions are regarded as variables, the above assignment can be used to change the value of a function globally.

Partial Assignments. In most cases a function is to be changed only for a single argument. For this case, we use the partial assignment to functions, **set** :

If $f \in F$ and $f :: \tau_1 \rightarrow \tau_2$ and $e' :: \tau'$ and $\tau' \preceq \tau_1$ and $e :: \tau$ and $\tau \preceq \tau_2$, then

$$U[\![\,\textbf{set}\ [f := e](e')\,]\!]_\sigma(\phi) = \begin{cases} f' & \text{if } \phi = f, \\ \sigma(\phi) & \text{otherwise,} \end{cases}$$

$$\textbf{with } f'(\psi) = \begin{cases} E[\![\,e\,]\!]_\sigma & \text{if } \psi = e', \\ \sigma(f)(\psi) & \text{otherwise.} \end{cases}$$

The **set** operation only affects the value of the function f. It is substituted by a new function value f' that differs from f only for the argument designated by the expression e' and yields the result e.

New. The creation of an object by **new** () instantiates the type ι_{Object} with a new object (see $E[\![\,\textbf{new}\ ()\,]\!]_\sigma$). Therefore, the active domain of this type is extended by one object, the return value of the operation (see also Section 3.2.4):

$$U[\![\,\textbf{new}\ ()\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \ \cup\ \{E[\![\,\textbf{new}\ ()\,]\!]_\sigma\} & \text{if } \phi = [\,], \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

The following two operations facilitate the evolution of object. Objects can **gain** or **lose** object types dynamically.

Gain. An existing object can be made an instance of object type τ by the operation **gain**:

$$\text{If } e \ ::\ \tau' \text{and } \tau' \preceq \iota_{Object} \text{ and } \tau \preceq \iota_{Object}, \text{then}$$

$$U[\![\,\textbf{gain}\ [\tau](e)\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \ \cup\ \{E[\![\,e\,]\!]_\sigma\} & \text{if } \tau \preceq \phi, \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

In order to maintain the subset relationship of the type lattice, the object is not only added to the active domain of type τ, but also to the active domains of τ's supertypes.

Lose. In contrast to the **gain** operation, **lose** deletes instance-type relationships. The effect of **lose** $[\tau](e)$is that all functions defined on type τ or a subtype of τ are no longer applicable to the object denoted by o. Consequently, we have to remove each occurrence of the object denoted by e from variables, sets and functions, if these are related to τ or a subtype. That is, we interpret type information as constraints that every valid database state has to fulfill. Once such constraints fail to hold, the state is changed by removing the objects from variable values.

In general, the state after an operation **lose** $[\tau](e)$ can be derived in two steps. First, the object denoted by the expression e is excluded from the active domain of type τ and all its subtypes (see the first case). Secondly, the values of all variables (and functions) are recursively derived from the new active domains:

$$\text{If } e \ ::\ \tau' \text{ and } \tau' \preceq \iota_{Object} \text{ and } \tau \preceq \iota_{Object}, \text{then}$$

$$U[\![\,\textbf{lose}\ [\tau](e)\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \setminus \{E[\![\,e\,]\!]_\sigma\} & \text{if } \phi \preceq \tau, \\ \sigma(\phi) & \text{if } \phi \preceq \iota_{Object} \wedge \phi \not\preceq \tau, \\ nv(\sigma(\phi), A(\phi)) & \text{otherwise} \end{cases}$$

where the function nv propagates the update operation. The function nv is applied to an old value $v \in V$ and its type μ and returns the new value $nv(v, \mu) \in V$. The new value is different form the old one only if the old one would not fulfill the type constraints. Therefore, the function is defined as follows:

$$nv(v,\mu) = \begin{cases} \bot & \text{if } \mu \preceq \iota_{Object} \wedge \\ & v \notin U[\![\, \textbf{lose}\,[\tau](e)\,]\!]_\sigma(\mu), \\ \bigcup\limits_{v' \in v} nv(v',\mu') & \text{if } \mu = \{\mu'\}, \\ \{\langle x,\ nv(v',\mu_2)\rangle \mid \langle x, v'\rangle \in v \wedge & \text{if } \mu = \mu_1 \to \mu_2, \\ \qquad x \in U[\![\, \textbf{lose}\,[\tau](e)\,]\!]_\sigma(\mu_1)\} & \\ v & \text{otherwise.} \end{cases}$$

The idea of the derivation is to use the structure of types in order to reduce the problem of specifying the new value of functions and sets to easier cases. In the first case, the old value, which is not element of the type μ anymore, is replaced by the null value ($\bot$). This case together with the last case, in which everything remains unchanged, are the anchors of the derivation. If the old value is a set (the second case), the derivation is evaluated for each element, recursively. The return value of the set is constructed by the union over all elements.[12] Analogously, the values of each function are also checked recursively (the third case).[13]

As we have seen, the values of functions and variables might change in a strongly-typed language that contains operations for (incremental) deletion. This is because of object sharing, which is one of the essential features of object-oriented data models [ABD+89]. Thus, a formal definition for update operations has to take such side-effects into consideration. In addition, the locality of these side-effects is not restricted to just one set like in the relational model. Therefore, we do not use the traditional approach to define the semantics of an update operation by stating the incremental change of the database state like in [AV87, Che91], for example.

3.2.6 Semantics of Control Flow Instructions

The semantics of the control flow instructions can be defined by specifying the combination of state transitions under certain restrictions.

Composition of Instruction. The composition of instruction (;) is defined as the sequential application of instructions:

$$U[\![\, i_1; i_2\,]\!]_\sigma = U[\![\, i_2\,]\!]_{U[\![\, i_1\,]\!]_\sigma}$$

Set-oriented Application. The standard semantics for an iterator **apply_to_all** (also called *filter, map,* or *replace* [BBKV87, AB88]) is the union of the operations applied to each element of the input set. In our case, this would be the union of the state transitions performed by applying instruction i to the initial state σ, for each element o of $E[\![\, \textit{set-expr}\,]\!]_\sigma$:

$$U[\![\, \textbf{apply\_to\_all}[i](o : \textit{set-expr})\,]\!]_\sigma = \bigcup_{v \in E[\![\, \textit{set-expr}\,]\!]_\sigma} U[\![\, i\,]\!]_{\sigma\{v/o\}}$$

[12] The union ignores null values as elements.

[13] Here, we only consider functions defined on object types. In case that functions are defined on constructed types, we have to ensure the type-validness of x in a more elaborated way because the derivation must not derive different tuples with the same first component.

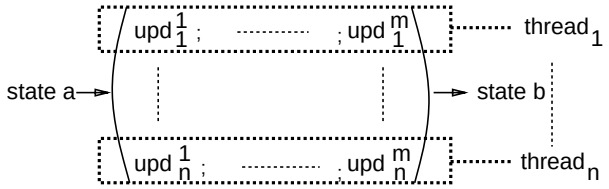


Figure 3.1: Transformation of the database state a to state b by the update operation **apply_to_all** $[upd^1; ...; upd^m](\{o_1, ..., o_n\})$

There are several problems with this approach: Consider a state where $f(p) = p'$, and $f(q) = q'$, and the update operation **apply_to_all** [**set** $[f := o](o)](o : \{p, q\})$, which sets the function f to the identity function for the objects p, q. If we look at the final states of both updates separately, one contains the pairs $\langle p, p\rangle$ and $\langle q, q'\rangle$ and the other one the pairs $\langle p, p'\rangle$ and $\langle q, q\rangle$. First of all, the union of these sets would not be a function in the first argument anymore, because the union of both states would contain pairs with a common first component. Therefore, we need a more sophisticated definition of that union (e.g. recursive, overriding). However, overriding unions are only adequate for "increasing" state transitions (e.g., creation of objects). Assignments, like in our example, would have to be expressed by the *difference* between the state before and after the update. On the other hand, operations such as **lose** need something like the *intersection* of states, since we remove information. For example, consider the deletion of the two objects p and q, in a state where only p, and q exist: **apply_to_all** [**lose** $[\iota_{Object}](o)](o : \{p, q\})$. According to the above definition, the active domain of the type ι_{Object} in the state $U[\![$ **lose** $[\iota_{Object}](p)\,]\!]_\sigma$ contains q but not p and ι_{Object}'s active domain in the state $U[\![$ **lose** $[\iota_{Object}](q)\,]\!]_\sigma$ includes p but not q, vice versa. The *union* of these states would not exclude both objects p, q from the state.

Thus, we conclude that merging the states needs different mechanisms depending on the update operation. Such a differentiation, however, would lead to the definition of set-oriented elementary update operations, but not to a definition of an iterator independent of the elementary update operations. This is an explanation for the SQL approach, where set-oriented single updates are possible. Furthermore, it shows that set orientation and updates do not go together too well. Obviously, we have to use another approach to define "set-orientation". For an easier presentation, let us introduce the following notation: Following [DV91], we call the application of update sequence to each individual element of the set a *thread* (illustrated as a row in Fig. 3.1) and the particular element the *thread object*. The j-th operation in an update sequence is denoted as upd^j, and the j-th operation within $thread_i$ (that is associated to the thread object o_i) as upd_i^j.

Since it seems impossible to combine threads uniformly using set-operations, we take the freedom to define a "set-oriented" update as one where the order of application to the elements of the set is not significant. That is, we could sequentialize in any order (e.g., depending on execution plans) or we could even parallelize. Hence, our formalization of set-oriented update sequences is based on the intuition that several

objects can be manipulated in one operation, if the effects of different threads are *independent* of each other. Then, the final state of the set-oriented update sequence does not depend on the execution order of the threads. This is, all possible execution orders yield the same result. Therefore the semantics of **apply_to_all** can be defined as the composition of threads, if all threads commute. The commutativity is satisfied, if the elementary update operations do not conflict with each other, which can be derived by a model-specific conflict relation containing the dependencies between the elementary update operations. Let us now present this idea in more detail.

Formally, we define the semantics of the **apply_to_all** instruction in three levels. The highest level defines the *semantics of* **apply_to_all** depending on the commutativity of its threads; the second level defines *commutativity of threads* depending on the conflict relation; and the lowest level defines the *conflict relation* according to data model specific operations:

Semantics of apply_to_all . If and only if all threads of **apply_to_all** $[i](o : \textit{set-expr})$ are pairwise commutative (w.r.t. the pairwise composition ";"), the **apply_to_all** instruction is defined by the overall composition ($\prod$) of its threads, as follows:

$$U[\![\, \textbf{apply\_to\_all}[i](o : \textit{set-expr}) \,]\!]_\sigma = \prod_{v \in E[\![\, \textit{set-expr} \,]\!]_\sigma} U[\![\, i \,]\!]_{\sigma\{v/o\}}.$$

Notice that this pairwise commutativity always results in executions with no conflicts; that is, *all* sequential executions of threads are equivalent.

Commutativity of Threads. Two threads $thread_i$, $thread_j$ are commutative if, in every possible database state, no elementary update operation upd_i^k, which is contained in $thread_i$, conflicts with any update upd_j^l of $thread_j$.

The conflicts between elementary update operations have to be defined by a conflict relation for the update operations of the data model. This is illustrated by the following table. Notice, however, the trade-off between the functionality of the instructions and the number of conflicts: Conflicts can be avoided if the applicability of instructions is restricted. Therefore, we also state the restrictions. For a detailed discussion of this trade-off and a discussion of the strong similarities between the **apply_to_all** instruction and semantic concurrency control see [LS93].

The sign '+' means that two operations do not conflict, and the variable *self* is the iterator variable: i.e., *self* denotes the object that is associated to the thread. Since the conflict relation is symmetric, the values of the conflict relation for empty cells are specified by changing row and column. The table describes also the conflicts between update and query operations. We state the query operations independent of their arguments, because the conflict between update operations and the arguments of query expressions are recursively contained in the table.

In addition to the content of the table, notice that query expressions do not conflict with each other. Furthermore, only assignments to local variables, whose scope is delimited by the thread, are allowed; the operations **set** $[f := e]$ and **lose** $[\iota_{Object}]$ may only be applied to the thread object.

	set $[f := ..](\textit{'self'})$	**new**	**gain** $[\tau_1]$	**lose** $[\tau_1]$
$\sigma(\tau_2)$	+	$\tau_2 =' \iota_{Object}'$	$\tau_1 \preceq \tau_2$	$\tau_2 \preceq \tau_1$
v	+	+	+	v defined on τ_1 or its subtypes
$\{e\}$	+	+	+	+
pick	+	+	+	+
$\lambda x : \tau.\, e$	+	+	+	+
$f(v)$[14]	$v \neq$ *'self'*	+	+	f defined on τ_1 or its subtypes
$\tau_2(e)$	+	+	$\tau_1 \preceq \tau_2$	$\tau_2 \preceq \tau_1$
guard $[\tau_2, ...](e)$	+	+	$\tau_1 \preceq \tau_2$	$\tau_2 \preceq \tau_1$
$\cup$	+	+	+	+
$\cap :: \tau_2$[15]	+	+	$\tau_1 \preceq \tau_2$	$\tau_2 \preceq \tau_1$
select $[P]$	+ [16]	+	+	+
project $[\tau_2]$	+	+	+	+
set $[f := ..](\textit{'self'})$	+	+	+	f defined on τ_1 or its subtypes
new		+	+	+
gain $[\tau_2]$			+	$\tau_2 \preceq \tau_1$
lose $[\tau_2]$				$\tau_1 =' \iota_{Object}'$

Figure 3.2: Summary of conflicts

Since all generic update operations are defined with respect to (sub-)typing constraints, we can state the following proposition about the type-validness of reachable states, after defining "reachable" as follows:

Reachable State. A database state is *reachable*, if it can be constructed by repeated application of COOL update statements, starting from an empty database.

Proposition. Each reachable state σ is type-valid: this is, the typing constraint $\sigma(\phi) \in [\![\tau]\!]$ is fulfilled, whenever $A(\phi) = \tau$.

The proof is by induction on the number of operations in the derivation of σ, and follows from the fact that all updates respect type-validness.

[14] v is a meta variable that denotes a variable: I.e., if the function f is applied to the variable *self* no conflicts arise.

[15] The notation $:: \tau_2$ indicates that the result of the query operation is of type τ_2.

[16] If f occurs in P, the conflicting operations are **set** and the function application $f(o_j)$

3.3 Additional Concepts

We now show how additional COCOON concepts not formalized so far, can be mapped to the formal level.

3.3.1 Declaration of Variables, Functions, and Types

Variable declarations in COCOON directly correspond to the declarations of the formal model:

var $v \ : \tau \ \rightsquigarrow v :: \tau$

If we map the function declarations of COOL **function** $name : PersonT \rightarrow$ **string** straightforwardly, we end up with the problem that the application of the $name$ function in the selection operation is not allowed after the **project** operation, because the element type of the query result is a supertype of the domain type of the $name$ function:

select $[name =$ 'Miller'$]($**project** $[name](PersonC))$

Therefore, the domain type of functions f is set to be the (usually unnamed) type that allows only the application of this function, i.e., the type $[f]$. In general, COOL function declarations are mapped to the formal definitions as follows:

function $f \ : \tau_1 \rightarrow \tau_2 \rightsquigarrow f :: [f] \rightarrow \tau_2$

Derived functions (like **function** $f \ : \tau_1 \rightarrow \tau_2 == expr$) are considered as syntactic abbreviations, because they can be used only in queries, but not in updates. Therefore, the function can be substituted like a macro by the defining expression $expr$.

On the formal level, object types have been denoted by function sets. Therefore, we define how such function sets can be derived from COOL type declarations. Later on, we can apply the function ft to a COOL type t in order to get the formal object type $ft(t)$. Therefore, the function ft can be derived from t's type declaration

type t **isa** $sup\_t_1, ..., sup\_t_n = f_1, ..., f_m$

as follows:

$$ft(t) := ft(sup\_t_1) \sqcap ft(sup\_t_n) \sqcap [f_1, ..., f_m]$$

As anchore of this derivation, we state that the object type ι_{Object} is the formal type of COOL's predefined type **object** : i.e., $\iota_{Object} = ft($**object**$)$.

3.3.2 COOL Update and Query Operations

In this section we map COOL's query and update operations (except **add** and **remove** , which are defined after the introduction of classes in Section 3.3.4) onto the expressions and instructions defined on the formal level.

For the sake of simplicity, we overload names of operations that are avaiable on the formal and on the COOL level. Thus, the COOL query operations **select**, **project**, **union**, **intersect**, **pick**, **guard**, the update operations **set**, **gain**, **lose**, and assignments (:=) as well as the control flow instructions (";" and **apply_to_all**) are already formally defined, since they are the same in COOL.

However, there are some differences between the COOL language in [SS90a] and the formal level, because of renaming and omitting classes and views on the formal level.

The name of the operation **extend** has not been taken over to the formal level, since name bindings (also denoted as macros) are of much broader use in programming languages:

If $e :: \{\tau\}$ and $e_i :: \tau_i$ for all $i = 1$ to n, then

$$\begin{aligned}\textbf{extend}\ [f_1 := e_1, ..., f_n := e_n](x : e) \equiv\ & \textbf{let}\ f_1 = x : \tau.\ e_1\ \textbf{in} \\ & \vdots \\ & \textbf{let}\ f_n = x : \tau.\ e_n\ \textbf{in} \\ & e''\ \textbf{end}\ \ldots\textbf{end}\end{aligned}$$

where e'' is the scope, in which the function names f_1 to f_n can be used. In COOL, the scope of the operation **extend** is at least bounded by the instruction, in which the operation is used.

Because a newly created object is almost always used as an instance of a more specific type than ι_{Object}, the operation **create** is provided in COOL. The operation **create** $[t](v)$ generates an object of type $ft(t)$ and assigns it to a variable v (that must be of type $ft(t)$ or a supertype). This semantics can be expressed by the following sequence of operations, where *newobj* is used as a temporary variable of type ι_{Object} and the newly created object becomes by the **add** operation a member of the predefined class *Objects* that contains all objects (see the next Sections):

$$\begin{aligned}\textbf{create}\ [t](v) \equiv\ & newobj := \textbf{new}\ (); \\ & \textbf{gain}\ [ft(t)](newobj); \\ & v := \textbf{project}\ [ft(t)](newobj); \\ & \textbf{add}\ [v](Objects);\end{aligned}$$

Finally, because COCOON contains classes and views, it is not intended that users raise queries against active domains of object types (see below). Therefore the operation **adom** cannot be used in COOL language. However, there are type predicates provided that allow to check whether an object is instance of a particular object type. These type predicates are be defined by the element test in the active domain:

$$T(o) \equiv o \in \textbf{adom}\ (T)$$

Alternatively, these predicates could also be defined by type guards:

$$T(o) \equiv \textbf{guard}\ [T, true, false](o)$$

3.3.3 Representing COCOON Classes

Even though "class" is a central concept in a COCOON database schema, it is a derived concept in terms of our formalization, that can be defined using objects and functions. There is only a slight extension of the formal level for static type-checking.

Introducing meta classes

We introduce a new object type *class* whose instances represent classes and that includes the following functions representing the class properties. Notice that we use COOL statements that are already mapped onto the formal level:

$$\begin{array}{lll} \textbf{define type } class \textbf{ isa object } = & cname & : \textbf{string}, \\ & bases & : \textbf{set of } class, \\ & suffp & : \textbf{object } \rightarrow \textbf{bool}, \\ & pmemb & : \textbf{set of object } ; \end{array}$$

As top element of the class hierarchy we require the existence of class *Objects*, which represents all existing objects (i.e., the active domain of type [ ]). Additionally, we provide the meta class $Classes$, whose extent represents all classes (including itself). The following COOL statements initialize the class hierarchy:

[Init] **define var** $Classes, Objects : class$;
create $[class](Classes)$; **create** $[class](Objects)$;
set $[cname :=$ 'Classes'$](Classes)$;
set $[cname :=$ 'Objects'$](Objects)$;
set $[bases :=$ 'Classes'$](Classes)$;
set $[bases :=$ 'Objects'$](Objects)$;
set $[pmemb := \{Classes, Objects\}](Classes)$;
set $[pmemb := pmemb(Classes)](Objects)$;
apply_to_all [**set** $[suffp := x.x \in pmemb(c)](c)](c : Classes)$

Extension to the type system

The representation of the member type of a class is still missing. In general, COOL type information could also be represented on the schema level. For example the COOL type $Pers$ could be modelled by an object of type *type* on which functions are applicable that contain the DDL information such as the type name $tname$, local defined functions *localf*, and supertypes *supert* (compare B). Furthermore, this object could result from the application of the member type function *mtype* to the class $Persons$. Such a representation is an elegant way to make retrievals on the schema level and schema evolution feasible with a similar set of operations for the schema- and the instance-level [TS92]. However, if we are interested in compile-time type-checking, we have to restrict this flexibility, because the type information must not be changed at run-time. Additionally, we have to exclude mixed-level operations, such as, for example, the application of an operation to members of a set of classes, which is constructed at run-time by a meta-level query.

Therefore, as already mentioned above, we have to extend the formal level: The "meta-" function A is also applicable to classes and returns the type of class members [17]. In order to improve readability we define the function *mtype* as the restriction of

[17] Due to the static nature of class definitions, it does not matter whether the function A is applied to the object representing a class or the class name.

A to classes. The result of the member type function *mtype* applied to the classes $Classes$ and $Objects$ is $ft(class)$ and **object** , respectively.

Representation of class predicates

As already mentioned in Chapter 2 the extent of classes can be constrained by necessary or necessary and sufficient predicates. In order to deal with some-classes, all-classes, and views uniformly, we "complete" the class predicate of some-classes to become necessary and sufficient. Then, maintaining consistency w.r.t. class membership during updates can easily be achieved by the fully determining predicate: the extent of a class (or view) is derived as all instances of the member type that fulfill the necessary and sufficient class predicate *suffp*. Therefore, the extent of a class C is defined by the following COOL query:

$$extent(C) = \textbf{select } [suffp(C)](\textbf{adom } (mtype(C)))$$

Definition of class objects

In order to represent COCOON classes by objects, each definition of a class C results in sequences of update operations on the schema level. The following statements are common to all kinds of class definitions:

[Common] **define var** $C : class$;
create $[class](C)$;
set $[cname := \text{'C'}](C)$;
set $[pmemb := pmemb(Classes) \cup \{C\}](Classes)$;

An object representing the class is generated, gets its classname assigned, and becomes a member of the (meta-)class $Classes$.

In order to obtain necessary *and sufficient* predicates for some-classes, we have to internally keep track of objects that have explicitly been added by the user (and not removed since). We collect, for each some-class, those objects in a set that have been added in the function *pmemb*. If applied to a some-class C, it returns the set of "potential members" of C (which is initialized by the empty set). Thus, the sufficient predicate for a some-class is the conjunction of the necessary condition (if any) and the test whether an object is included in $pmemb(C)$. The set of base classes, *bases*, which is needed for the view update mechanism, is the singleton set including the class itself. Thus, the **some** class definition

define class C $: \tau$ **some** $C_1, ..., C_n$**where** p

consists of the above [Common] statements plus the following ones:

[Some] **set** $[pmemb := \emptyset](C)$;
set $[bases := \{C\}](C)$;
set $[suffp := x.\ x \in pmemb(C) \wedge\ p(x) \bigwedge_{i=1}^{n} suffp(c_i)](C)$;

In case of **all** classes, the set of base classes is the union of the base classes of the superclasses, and the sufficient predicate is defined by the predicate p and the class membership in all superclasses. Therefore, the **all** class definition

define class C $: \tau$ **all** $C_1, ..., C_n$**where** p

is mapped onto the [Common] statements plus the following operations:

[All] **set** $[bases := \bigcup_{i=1}^{n} bases(C_i)](C)$;

set $[suffp := x.\ p(x) \bigwedge_{i=1}^{n} suffp(C_i)](C)$;

In case of a view definition

define view C **as** e

the predicate *suffp* is the membership in the defining expression:

[View] **set** $[suffp := x.\ x \in e](C)$;

The *bases* of the view can be derived by the following recursive predicate:

$$bases(e) := \begin{cases} e & \text{if } e \text{ is a some-class,} \\ bases(e') & \text{if } e = \textbf{project}[\ldots](e'), \\ & \text{or } e = \textbf{extend}[\ldots](e'), \\ & \text{or } e = \textbf{select}[\ldots](e'), \\ bases(e') \ \cup \ bases(e'') & \text{if } e = e' \cap e'', \text{or } e = e' \cup e''. \end{cases}$$

Class hierarchy

The partial reflexive order between classes ($\sqsubseteq$) is defined by the conjunction of the subtype relationship between the member types and the predicate inclusion between the necessary and sufficient predicates:

$$C_1 \ \sqsubseteq \ C_2 \ \stackrel{\text{def}}{=} \ mtype(C_1) \ \preceq \ mtype(C_2) \ \wedge \ suffp(C_1) \implies suffp(C_2)$$

3.3.4 Mapping the Update Operations add and remove

Since classes are modeled by using objects and functions, we can specify the semantics of the operations **add** and **remove** in terms of the elementary operations. That is, they are not elementary operations of our language, but derived as follows :

add $[e](C)$ $\equiv$ **apply_to_all** [**set** $[pmemb := pmemb$ **union** $\{e\}](y)]$
(y : **select** $[\emptyset \neq$ **select** $[x \sqsubseteq c](x : bases(C))$
and $bases(c) = \{c\}](c : Classes))$

remove $[e](C)$ $\equiv$ **apply_to_all** [**set** $[pmemb :=$ **select** $[x \neq e](x : pmemb)](c)]$
($c : bases(C)$)

The **add** operation is defined by applying the **set** operation that adds the object e to the set $pmemb(c)$, for each class c contained in the result of the **select** operation. The predicate of the selection chooses all some-classes of the database (identified by the condition $bases(c) = \{c\}$) that are superclasses of a class included in $bases(C)$. Thus, adding objects to a class is propagated to all its superclasses. Removing objects from a class can be specified easier, because the propagation to the subclasses is carried out by the necessary and sufficient predicates. Therefore, the semantics of the **remove** operation is to take the object out of the *pmemb*-sets of the classes contained in $bases(class)$. For a detailed discussion of the motivation and alternatives for the semantics of **add** and **remove** see [LS92].

3.3.5 Inverse Functions

Derivation of Inverse Functions

In a COCOON schema two functions can be defined to be inverses of each other. However, it is sometimes useful to refer to the inverse of a function that is not explicitly defined in the database schema. We provide a shorthand to derive the inverse of any function. However, let us define the semantics of "inverse of a function" first.

In general, we conceive functions as binary relations (similar to IRIS [WLH90]). In case of single-valued functions, the first component of the relation denotes an argument of the function and the second component is the result of the function application. Set-valued functions can be mapped to binary relations by inserting a pair $\langle o, o' \rangle$ in the relation, if o' is an element of the set that results from the function application to o. The inverse of a function is interpreted as the same relation, with the components exchanged. Since, in general, a function may yield the same value for different arguments, the inverse function has to be defined as set-valued.

Assuming that function f is defined with the signature $[f] \rightarrow \tau_2$, the shorthand f^{-1} is defined by the following expression:

$$f^{-1}(o) = \textbf{select } [o = f(o')](o' : \textbf{adom } ([f])), \quad \text{if } \tau_2 \text{ is not a set type}$$
$$f^{-1}(o) = \textbf{select } [o \in f(o')](o' : \textbf{adom } ([f])), \quad \text{if } \tau_2 \text{ is a set type}$$

Notice, however, that the key word **unique** can be used for single-valued and set-valued functions to indicate that the inverse function is single-valued. In this case, the above definitions can be extended by applying the **pick** operation to the selection.

Updating Inverse Functions

Because some updates are more easily expressed on one function, and others on the inverse, it is convenient to allow updates on either of two inverse functions. Of course, if one of them is updated, the changes have to be propagated to the other automatically.

For example, let us consider the inverse functions *works_for* that is applicable to employees and *staff* that is defined on departments (*Dept*). More formally, the signatures of the functions are as follows:

$$works\_for :: [works\_for] \rightarrow Dept \textbf{ inverse } staff$$
$$staff :: [staff] \rightarrow \{Empl\} \textbf{ inverse } works\_for$$

If a new department d is assigned to an employee e by the operation

set $[works\_for := d](e)$

the inverse function must be changed by adding e to $staff(d)$. However, if the staff of a department gets exchanged, it is ruther cumbersome to use assignments of the $works\_for$ function (to nullify the old staff and set the new ones). Instead, it should be possible to exchange d's staff by

set $[staff := \{e_1, \ldots, e_n\}](d)$

such that the $works\_for$ function returns d only for the employees e_1 through e_n. Thus, we provide a flexible update capability for inverse functions that is independent of any storage strategy, since typically only one function will be stored, the other one will be derived. Similar to the view update mechanism, we provide a default semantics for elementary update operations. In case that a more sophisticated semantics is needed in an application, one has to define appropriate methods using the elementary update operations.

The propagation of updates to inverse functions depends on whether the functions are single- or set-valued. Therefore, we have to consider the four cases for the three kinds of operations that can change a function: (i) partial assignments, (ii) global assignments, and (iii) **lose** operations that might remove instances from either domain. Assume the following function definitions:

function $f : [f] \rightarrow \tau_f$ **inverse** g
function $g : [g] \rightarrow \tau_g$ **inverse** f

where the four cases result from different function ranges:

I	$\tau_f \preceq [g]$	$\tau_g \preceq \{[f]\}$
II	$\tau_f \preceq [g]$	$\tau_g \preceq [f]$
III	$\tau_f \preceq \{[g]\}$	$\tau_g \preceq \{[f]\}$
IV	$\tau_f \preceq \{[g]\}$	$\tau_g \preceq [f]$

Let us now discuss how the application of the three kinds of operations to the function f propagates to the inverse g [18].

(i) The effect of the operation **set** $[f := e'](e)$ on the binary relation is: first, delete all pairs that have e as their first component; then, in the cases I and II, insert the new pair $\langle e, e' \rangle$; otherwise (case III and IV) insert a pair $\langle e, e'' \rangle$ for each $e'' \in e'$. In the cases II and IV the old tuple $\langle \hat{e}, e' \rangle$ is removed from the relation, such that g is a single-valued function. The effect to $\hat{e}$ is that $f(\hat{e})$ is set to undefined ($\perp$) in case II, and e' is removed from $f(\hat{e})$ in case IV.

Therefore the effect on g is as follows: (I) e is added to the set $g(e')$; (II) the value $f(g(e')$ is undefined for the old value $g(e')$, the new value of $g(e')$ is e; (III) $g(e'')$ contains e if and only if $e'' \in e'$; (IV) $g(e'')$ is e if and only if $e'' \in e'$, $g\hat{e})$ is undefined if it was e and $\hat{e} \notin e'$.

[18] Because the update capability is symmetric, we do not have to discuss the application of the operations to g.

(ii) The operation $f := x.e$ can be mapped to the **set** $[f := e'']([e'])$ where e'' results from substituting x in e by e', which iterates over the instances of $[f]$.

(iii) By the operation **lose** $[f](e)$ the function f is not applicable to e anymore, because e is removed from the domain $[f]$ (see Sec. 3.2.5). Additionally, e is removed from function values $g(e')$ according to the derivation rule nv, because the range of g is either a subtype of $[f]$ or a subtype of $\{[f]\}$. Therefore, the inverse constraint is fulfilled anyway.

Chapter 4

Extensions of the Model

4.1 General Idea

In Chapter 2 we have introduced those components of the COCOON model that we considered most essential for query languages for object models, generic update operations that cope with object sharing, and a powerful view mechanism. We have not included (in full generality) complex values such as sets, lists, and tuples into the model for the sake of simplicity. However, since complex values are useful if sharing is not desired, we want to show in this chapter, how the COCOON model can be extended, accordingly. Moreover, we take these extensions as archetypes for other extensions that might appear useful for some applications of OODBMSs. Such extensions can be carried out by:

- specifying the additions to the type system, e.g., subtyping rules,
- defining operations, e.g., retrieval operations as well as constructors and destructors.

This is done in the following section for *sets* and *tuples*, as an example. Further extensions could include *lists*, *arrays*, and *bags*, to name a few.

4.2 Examples for Extensions

4.2.1 Sets

Sets occur in the model in several places: set-valued functions, extents, which are the sets of class members, and almost all query operations are applied to and return sets. In fact, the type system already uses *set* as an orthogonal type constructor (on the formal level, see Section 3.2.3): E.g., variables, and ranges of domains can be defined as **set of set of** *person*. On the COOL level, though, we have to extend the definition of *typeDef* such that not only object types, but also set types can be defined. Therefore the *typeDef* rule in Appendix A can be replaced, for example, by the following rules:

$$
\begin{array}{lcl}
typeDef & = & objectType \mid setType \\
objectType & = & \textbf{type}\ typeIdent\ [\ \textbf{isa}\ superTypeList\ ]\ ["="\ functionList] \\
setType & = & \textbf{set of}\ typeIdent
\end{array}
$$

In order to work with sets, we have already defined the operations **union**, **intersect**, **select** and **extend**, as well as the predicate $\in$, which tests set membership. Putting set braces around expressions constructs sets, and the operation **pick** serves as a destructor. However, there are still some useful operations missing: Numerous papers have considered such orthogonal sets of operations for complex values (see [AB88] as a survey). We could just include them into our model, if necessary. For example sets of sets can be destructed by an operation such as **set-collapse**, that returns the union of all sets that are elements of the argument.

$$\frac{e :: \{\{\tau\}\}}{\textbf{set-collapse}(e) :: \{\tau\}} \qquad E[\![\,\textbf{set-collapse}(e)\,]\!]_\sigma = \{v' \in v \mid v \in E[\![\,e\,]\!]_\sigma\}$$

Similarly, there could be an additional set constructor **powerset** that generates the powerset of a set.

$$\frac{e :: \{\tau\}}{\textbf{powerset}(e) :: \{\{\tau\}\}} \qquad E[\![\,\textbf{powerset}(e)\,]\!]_\sigma = \{v \subseteq E[\![\,e\,]\!]_\sigma\}$$

Additionally, we might need an iterator for sets that allows us to apply an operation to the elements of the set. Therefore, we can introduce the iterator **map**:

$$\frac{f :: \tau_1 \rightarrow \tau_2,\ e :: \{\tau_1\}}{\textbf{map}\ [f](e) :: \{\tau_2\}} \qquad E[\![\,\textbf{map}\ [f](e)\,]\!] = \{f(e') | e' \in E[\![\,e\,]\!]\}$$

That operation yields a set whose elements are the values of the application of f to each element of e. For example, the iterator **map** can be used to construct the set of all persons' names as follows: [1]

$\textbf{map}\ [name](Persons)$

Notice that the above operations are algebra-like extensions. Instead, COCOON can be brought closer to functional languages by introducing a more general set iterator (such as *homu* in Machiavelli [OBBT89, BO93] and *pump* in FAD [BBKV87]) that need not necessarily return sets, but executes retrieval expressions similar to the **apply_to_all** iterator that executes updates in a set-oriented way.

4.2.2 Tuples

The inclusion of tuples in COCOON could, above all, be motivated by the following two objectives:

- modeling n-argument functions (see [FBC+87, HK87]), and
- representing relations (sets of tuples).

[1] Instead of the **map** operation, we also use the asterisk as a shorthand: e.g., $name^*(Persons)$.

As you might have noticed, there are only unary functions in our example. However, in some cases n-ary functions are useful in applications. These n-ary functions could be modeled by objects that serve as connectors (like in CODASYL systems), but modeling them by tuples is often more reasonable.

Let us, for example, introduce binary functions by extending the running example such that the hiredate belongs to the relation between employees and companies. Now, we might be interested in a function that returns the company for an employee and a hiredate, or a function that returns the sets of employees that where hired from a company on a certain day, or a function that returns all companies with the hiredates for which an employee has worked. That is, we need the facility to define functions that are defined on or return tuples.

Relations are especially important as the interface between an object-oriented system, in which sets of objects are the result of query expressions, and application programs or ad-hoc user interfaces, which can only receive descriptive values, not the objects (OIDs) themselves ([ASL89, SS90b]).

Thus, in general, we need tuple as a type constructor, which allows us to define, for example, tuple-valued variables, tuples of tuples, and relations, which are sets of tuples. In order to specify tuple-types we can extend the Object Definition Language of COOL by the following statements:

$$\begin{aligned} typeDef &= objectType \mid setType \mid tupleType \\ tupleType &= \textbf{tuple of } \text{"("} \; label \; \text{" : "} \; typeIdent \; \{\text{","} \; label \; \text{" : "} \; typeIdent\} \; \text{")"} \\ label &= ident \end{aligned}$$

On the formal level we again have to define subtyping rules and operations that are generic for tuples. First, however, we have to introduce the set of label names L which contains the names of labels that are defined in a database schema. Now, the subtype relationship for tuples can be defined as follows (see record types in [Car84]), where a_i are elements of L:

$$[\text{TUPLES}] \; \frac{\tau_1 \preceq \tau_1', \ldots, \tau_n \preceq \tau_n'}{(a_1 : \tau_1, \ldots, a_{n+m} : \tau_{n+m}) \preceq (a_1 : \tau_1', \ldots, a_n : \tau_n')}$$

Considering the above relations between employees, companies, and hiredates, we can define the following tuple-types

define type *empdate* **tuple of** $(e : employee, d : date)$;
define type *compdate* **tuple of** $(c : company, d : date)$

that can be used in function declarations as follows:

define function $employer : empdate \rightarrow company$;
define function $hired : compdate \rightarrow employee$;
define function $employment\_history : employee \rightarrow$ **set of** *compdate*

As generic operations we can, for example, provide the following operations for constructing tuples (by $\langle\rangle$), for concatenating tuples with disjoint label sets (**concat**), and for the selection of a tuple component (by .) as follows:[2]

[2]The semantic domains of tuples are considered as functions that map the labels (elements of L) to their component values.

$$\frac{a \in L, e :: \tau}{\langle a = e \rangle :: (a : \tau)} \qquad E[\![\langle a = e \rangle]\!] = \{\langle a, E[\![e]\!] \rangle\} \in F$$

$$\frac{\begin{array}{l} e_1 :: (a_1 : \tau_1, ..., a_n : \tau_n), \\ e_2 :: (b_1 : \tau'_1, ..., b_m : \tau'_m), \forall i, j : a_i \neq b_j \end{array}}{\begin{array}{l} e_1 \textbf{ concat } e_2 :: \\ (a_1 : \tau_1, ..., a_n : \tau_n, b_1 : \tau'_1, ..., b_m : \tau'_m) \end{array}} \qquad E[\![e_1 \textbf{ concat } e_2]\!] = E[\![e_1]\!] \cup E[\![e_2]\!]$$

$$\frac{e :: (a_1 : \tau_1, ..., a_n : \tau_n)}{e.a_i :: \tau_i} \qquad E[\![e.a_i]\!] = \begin{cases} v & \text{if } \langle a_i, v \rangle \in E[\![e]\!], \\ \perp & \text{otherwise.} \end{cases}$$

In Chapter 2 the operation **extract** was introduced in order to present the results of query operations in the form of (nested) relations. The result of this operation is not representable within the formal model, unless we include tuples. With the inclusion of tuples and sets, the operation **extract** can be defined as follows:

$$\textbf{extract } [f_1, \ldots, f_n](\textit{set-expr}) = \textbf{map } [\, \langle f_1 = f_1(e), \ldots, f_n = f_n(e) \rangle \,](e : \textit{set-expr})$$

Chapter 5

Ongoing and Future Work

5.1 Meta modeling and schema evolution

The COCOON model is powerful enough to represent COCOON schemata, that is, as a part of each COCOON objectbase, we have a meta schema that comprises information about the types, classes, and functions that have been defined in the schema. These meta objects not only serve as the "data dictionary" or objectbase catalog, they are also used as the basis for schema evolution. The COOL update operators can also be applied to the meta objects, so as to express schema modifications. The problem attacked in [TS92] was to define the update operations and the meta types/classes in such a way that schema modifications automatically propagate to the instance level. Of course, an implementation will avoid eager transformation and try to use views, schema versions, or lazy transformation of the instance objects to avoid load peaks at schema modification time.

Another application of meta modeling is the interoperability between several autonomous databases. In [SST92] we have presented the foundations on which we want to build composite schemas that import (parts of) local databases into a federated, multi-database system. It turned out that the availability of meta objects is extremely useful in this context, too. This idea is developed even further in [TS93], where complete schema transformation processors for federated objectbases are identified that also make use of meta information. We have left out the details of this aspect in this report, these can be found elsewhere. To give an overview, Appendix B contains the COCOON meta schema.

Interoperability between database systems with heterogeneous data models was investigated in a joint project with EPF Lausanne: The database group in Lausanne extended an entity-relationship model with a powerful query language. The foundations for a federated multilingual database system, FEMUS, were investigated, so as to show the transformation of entity-relationship databases (ERC+) into COCOON databases, including the transformation of ERC+ calculus expressions into COOL queries and updates. The experiences of this project are described in [ADS+93].

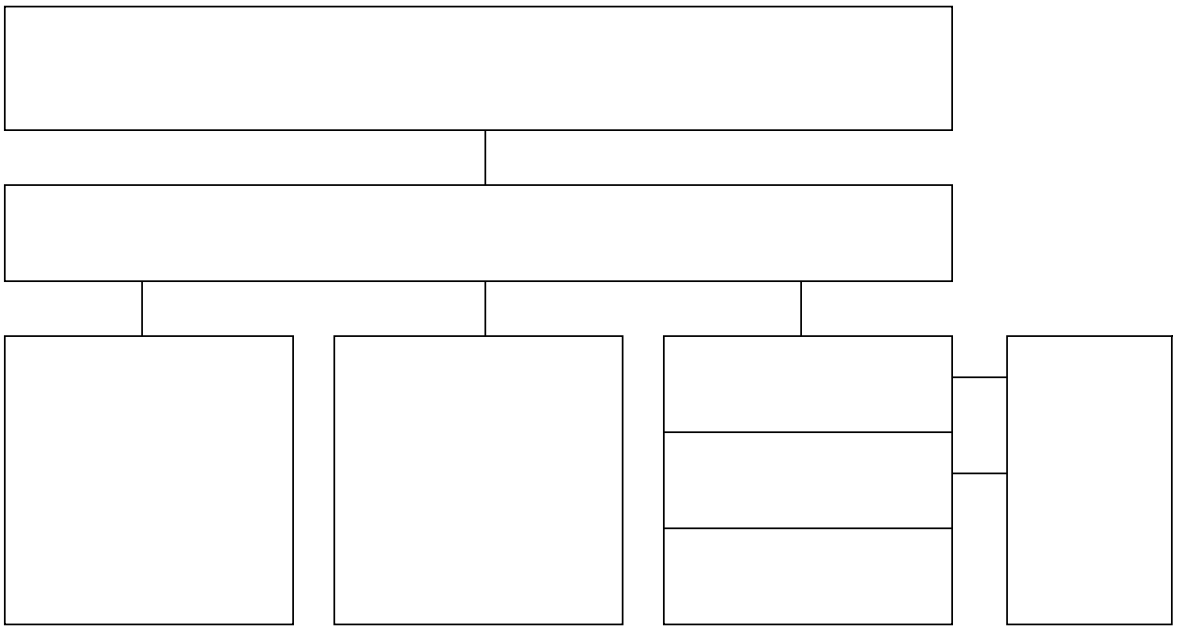


Figure 5.1: Architecture of the COCOON implementations

5.2 System architecture and optimization

One of the main objectives of the COCOON project has been to investigate architectural alternatives for the implementation of OODBMSs, as well as to find appropriate query processing and optimization strategies, to support object-oriented paradigms efficiently. In particular, the goal was to evaluate the use of the nested relational DBMS kernel DASDBS [SPSW90] as the storage manager. This allows for hierarchical clustering techniques that can be exploited to reduce the amount of physical disk I/O when large structured objects are loaded into main memory. A discussion of this approach is contained in [DHL+92, Sch93]. A prototype of a physical database design tool has been implemented to aid the database administrator in selecting a good physical design for a given COOL objectbase and load description.

At query processing time, the query optimizer has to transform COOL queries into execution plans that consist of DASDBS kernel calls and higher-level query processing strategies, such as joins and address dereferencing [RS93b, RRS92]. The EXODUS query optimizer generator [GD87] was used to build parts of the COOL query optimizer.

In addition to the DASDBS realization of COCOON, we have built two further prototypes in order to allow comparisons (cf. Fig. 5.1), both qualitative (how difficult is the implementation?) and quantitative (performance experiments): one uses the ONTOS object-oriented DBMS product, the other one the Oracle RDBMS [TS91]. Particularly the Oracle vs. DASDBS experiments are intended to evaluate the effects of hierarchical clustering, since Oracle allows for the definition of "clusters" that can simulate two-level nested relations. Some qualitative experiences with the two commercial platforms have been given in [TS91], results concerning the DASDBS realization are given in [TRSB93, RS93a, RS93b].

5.3 Extensions to the formal model

Sorts vs. types. We have already started to investigate further extensions of the formal model. One such extension is the distinction of several, disjoint *sorts* of objects. Unlike in our current model, where object evolution is completely dynamic (i.e., objects can dynamically gain and lose arbitrary types), this would introduce two levels of sorts/types information: a sort would be a higher level concept, objects can not change sorts. However, below each sort, there would be a whole lattice of types in our sense. Several other models, for example [HFW90, AH87, BO93] have similar concepts.

The technical consequence is that we have to introduce more than one type (sub-) lattice, one per sort. The sorts will be the top elements of their sublattices and dynamic type changes will be restricted to one sublattice. The modifications of the formal model are rather straightforward.

Due to the pairwise disjointness of sorts, this extension can also be used as a framework for the composition of local databases into multi-databases. That is, the objects of each database are regarded as a sort (or a set of sorts in case that single databases already provide different sorts), the integration of different databases can be studied by functions that map objects from one sort to others.

Persistent vs. transient objects. A second extension of the formal model is to introduce a distinction between persistent and transient objects. Again, a consequence is that we have to extend the meta-schema so as to include a class of persistent and a class of transient objects. Here, we do not need two sorts, since the property of being persistent should be variable over time. The related question is: how do we want persistence to work. There are two classical approaches, both of them can easily be realized in our model.

First, persistence can be based on reachability (from some persistence roots). This approach is favored by many current OODBMSs and persistent OOPLs. Objects are automatically converted to and from persistent/transient states depending on their reachability. We could distinguish persistent and transient classes, types, functions, and variables (the keyword **define** identifies persistent declarations). Objects would be persistent, if they are reachable from persistent classes or variables via persistent functions, recursively.

The second approach is explicit control over persistence of objects, which is more the classical database approach with operations such as store and delete. In this case, persistence can be defined by class membership. The active domain of the **object** type, $adom(\textbf{object})$, would be separated from the extent of *Objects*, such that *Objects* contains only the *persistent* objects. Therefore, $extent(Objects)$ might be a proper subset of $adom(\textbf{object})$. Now, objects become persistent (transient) by being added to (removed from) class *Objects*. The next question is how to deal with references from persistent objects to transient ones. Our solution would be to use the propagation mechanism of the lose operation to eliminate transient objects from database states (sets, variables, function values) upon transaction commit by the statement

apply_to_all [**lose** (o)]$(adom(\textbf{object})$ **difference** $Objects)$.

Bibliography

[AB88] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, Paris, May 1988.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In Kim et al. [KNN89], pages 40–57.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[ADS+93] M. Andersson, Y. Dupont, S. Spaccapietra, K. Yétongnon, M. Tresch, and H. Ye. The FEMUS experience in building a federated multilingual database. In *Proc. 3st Int'l Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS)*, Vienna, Austria, April 1993. IEEE Computer Society Press.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[AK89a] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. Technical Report 1022, INRIA, Paris, April 1989.

[AK89b] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, pages 159–173, Portland, June 1989. ACM, New York.

[ASL89] A.M. Alashqur, S.Y.W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 433–442, Amsterdam, August 1989.

[AV87] S. Abiteboul and V. Vianu. A transaction language complete for database update and specification. In *Proc. ACM SIGACT/SIGMOD Symp. on Principles of Database Systems*, pages 260–268, San Diego, CA, March 1987.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Int. Conf. on Very Large Databases*, pages 97–105, Brighton, September 1987.

[BdV91] H. Balsters and C. C. de Vreeze. A semantic of object-oriented sets. In *Proc. of 3rd Intl. Workshop on Database Programming Languages*, pages 187–200, Nafplion, Greece, August 1991.

[Bee89] C. Beeri. Formal models for object-oriented databases. In Kim et al. [KNN89], pages 370–395. Revised version appeared in "Data & Knowledge Engineering", Vol. 5, North-Holland.

[BF91] H. Balsters and M. M. Fokkinga. Subtyping can have simple semantics. *Theoretical Computer Science*, 87:81–96, 1991.

[BO93] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 1993. to appear.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9:171–216, 1985.

[Car84] L. Cardelli. A semantics of multiple inheritance. In G. Kahn, D. B. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, pages 51–67. LNCS 173, Springer, Heidelberg, 1984.

[Che91] W. Chen. Database updates: Constructors and quantifiers. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. LNCS 566, Springer Verlag, Heidelberg.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Day89] U. Dayal. Queries and views in an object-oriented data model. In R. Hull, R. Morrison, and D. Stemple, editors, *2nd Int'l Workshop on Database Programming Languages*, pages 80–102, Oregon Coast, June 1989. Morgan Kaufmann, San Mateo, Ca.

[DD91] K.C. Davis and L.M.L. Delcambre. A denotational approach to object-oriented query language definition. In *Proc. Int'l. Workshop on Specifications of Database Systems*, Glasgow, Scotland, June 1991. Workshops in Computing, Springer.

[DHL+92] S. Dessloch, T. Härder, F.-J. Leick, N.M. Mattos, C. Laasch, C. Rich, M.H. Scholl, and H.-J. Schek. COCOON and KRISYS – a comparison –. In R. Bayer, T. Härder, and P. C. Lockemann, editors, *Objektbanken für Experten*, Informatik Aktuell. Springer, October 1992.

[Dij75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8), August 1975.

[DV91] K. Denninghoff and V. Vianu. The power of methods with parallel semantics. In *Proc. Int. Conf. on Very Large Databases*, Barcelona, Spain, 1991.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derret, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, and M.A. Neimat. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, January 1987.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, May 1987.

[GH91] J. Göers and A. Heuer. Definition and application of metaclasses in an object-oriented database model. Manuscript, Institut für Informatik, TU Clausthal, Germany, June 1991.

[HFW90] A. Heuer, J. Fuchs, and U. Wiebking. OSCAR: An object-oriented database system with a nested relational kernel. In *Proc. Int'l Conf. on Entity-Relationship Approach*, Lausanne, Switzerland, October 1990. North-Holland.

[HK87] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM90] S. Hong and F. Maryanski. Using a meta model to represent object-oriented data models. In *Proc. 6th Int'l IEEE Conf. on Data Engineering (ICDE)*, pages 11 – 19, Los Angeles, CA, February 1990. IEEE Comp. Soc. Press.

[HS90] A. Heuer and P. Sander. Preserving and generating objects in the LIVING IN A LATTICE rule language. In *Proc. GI Workshop Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990. Tech. Rep. 90/3, TU Clausthal.

[HS91] A. Heuer and M.H. Scholl. Principles of object-oriented query languages. In H.-J. Appelrath, editor, *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications*, pages 178–197, Kaiserslautern, March 1991. IFB 270, Springer Verlag, Heidelberg.

[Kim89] W. Kim. A model of queries for object-oriented databases. In *Proc. Int. Conf. on Very Large Databases*, pages 423–432, Amsterdam, August 1989.

[Kla90] W. Klas. *A Metaclass System for Open Object-Oriented Data Models.* PhD thesis, Technische Universität Wien, January 1990.

[KNN89] W. Kim, J.-M. Nicolas, and S. Nishio, editors. *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, December 1989. North-Holland.

[LS92] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. In *Proc. of 2. GI Workshop Information Systems and Artificial Intelligence*, pages 40–55, Ulm, Germany, February 1992. IFB 303, Springer Verlag, Heidelberg.

[LS93] C. Laasch and M.H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proc. of the IEEE Conf. on Data Engineering*, pages 4–13, Vienna, Austria, April 1993.

[MCB90] M.V. Mannino, I.J. Choi, and D.S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, November 1990.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, 1988.

[O291] The O2 book. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *The O2 book.* Morgan Kaufman, 1991.

[OBBT89] A. Ohori, P. Buneman, and B. Breazu-Tannen. Database programming in Machiavelli a polymorphic language with static type inference. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 46–57, Portland, OR, May-June 1989.

[RRS92] A. Rosenthal, C. Rich, and M.H. Scholl. Reducing duplicate work in relational join(s): A unified approach. Technical Report 172, ETH Zurich, Dept. of Computer Science, January 1992. Submitted for publication.

[RS93a] C. Rich and M.H. Scholl. An evaluation of alternative object reassembly strategies, 1993. In preparation.

[RS93b] C. Rich and M.H. Scholl. Query optimization in an OODBMS. In *Proc. GI Conf. on Database Systems for Office, Engineering and Scientific Applications*, Braunschweig, Germany, March 1993. Springer, Informatik aktuell. To appear.

[Sch93] M.H. Scholl. Physical database design for an object-oriented database system. In J.C. Freytag, G. Vossen, and D.E. Maier, editors, *Query Processing for Advanced Database Applications.* Morgan Kaufmann, Los Altos, Ca., 1993. To appear.

[SLT91] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In *Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991. Springer LNCS 566.

[SÖ90] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented databases. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[SPSW90] H.-J. Schek, H.-B. Paul, M.H. Scholl, and G. Weikum. The DASDBS project: Objectives, experiences and future prospects. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):25–43, March 1990. Special Issue on Database Prototype Systems.

[SRH90] M. Stonebraker, L.A. Rowe, and M. Hirohama. The implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):125–142, March 1990. Special Issue on Prototype Systems.

[SS90a] M.H. Scholl and H.-J. Schek. A relational object model. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 89–105, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[SS90b] M.H. Scholl and H.-J. Schek. A synthesis of complex objects and object-orientation. In *Proc. IFIP TC2 Conf. on Object Oriented Databases (DS-4)*, Windermere, UK, July 1990. North-Holland. To appear.

[SS91] H.-J. Schek and M.H. Scholl. From relations and nested relations to object models. In M.S. Jackson and A.E. Robinson, editors, *Aspects of Databases — Proc. 9th British Nat. Conf. on Databases*, pages 202–225, Wolverhampton, UK, July 1991. Butterworth-Heinemann, Oxford.

[SST92] M.H. Scholl, H.-J. Schek, and M. Tresch. Object algebra and views for multi-objectbases. In *Proc. Int'l Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.

[Sto77] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge (Mass.), 1977.

[SZ89] G.M. Shaw and S.B. Zdonik. An object-oriented query algebra. *IEEE Data Engineering*, 12(3):29–36, September 1989. Special Issue on Database Programming Languages.

[SZ90] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proc. of the IEEE Conf. on Data Engineering*, pages 154–162, Los Angelos, CA, February 1990.

[TRSB93] W.B. Teeuw, C. Rich, M.H. Scholl, and H.M. Blanken. An evaluation of physical disk i/os for complex object processing. In *Proc. of the IEEE Conf. on Data Engineering*, Vienna, Austria, April 1993. To appear. A more detailed version is available as Technical Report 183, ETH Zurich, Dept. of Computer Science, 1992.

[TS91] M. Tresch and M.H. Scholl. Implementing an object model on top of commercial database systems. In *Proc. 3rd GI Workshop on Foundations of Database Systems*, Volkse, Germany, May 1991. Technical Report 158, Dept. of Computer Science, ETH Zürich.

[TS92] M. Tresch and M.H. Scholl. Meta object management and its application to database evolution. In *Proc. 11th Int'l Conf. Entity-Relationship Approach*, Karlsruhe, Germany, October 1992. Springer.

[TS93] M. Tresch and M.H. Scholl. Schema transformation processors for federated objectbases. In *Proc. 3rd Int'l Symp. on Database Systems for Advanced Applications (DASFAA)*, Daejon, Korea, April 1993. To appear.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, March 1990. Special Issue on Prototype Systems.

[Zan84] C. Zaniolo. Database relations with null values. *Journal of Computer and System Sciences*, 28:142–166, 1984.

Appendix A

The Syntax of COOL

We use an EBNF to describe the syntax of the language COOL. Terminal symbols are **bold** faced or enclosed in quote marks (" "), whereas non-terminals are *italic*. The Meta-symbols are: braces (|), choose one of the items; angular brackets ([ ]), optional item; curly brackets ({ }), repeat item, possibly 0 times; and round brackets (( )), group items together. The start symbol is *dbSession*.

Special Characters and Reserved Words

+	-	*	/	=	≠	<	≤	>	≥
⊂	⊆	⊃	⊇	∈	{	}			
(	)	[	]	,	.	:	:=	;	→

all	**and**	**as**	**class**	**connect**
database	**define**	**difference**	**disconnect**	**div**
end	**function**	**in**	**intersect**	**inverse**
isa	**let**	**or**	**mod**	**not**
set of	**some**	**type**	**union**	**var**
view	**where**			

Standard Identifiers

false	true	null	#	∅
boolean	integer	real	string	
object	Objects			

Generic Function Identifiers

add	**apply_to_all**	**create**	**delete**	**extend**
extract	**gain**	**guard**	**lose**	**pick**
project	**remove**	**set**	**select**	

Vocabulary and Expressions

number = *integer* | *real*
string = "”" { *character* } "”"
ident = *letter* { *letter* | *digit* } | #
designator = *varIdent* | *classIdent* | *fcnIdent* ["(" *designator* ")"] | *pick*

expression = *simpleExpr* [*relOp simpleExpr*]
relOp = " = " | " ≠ " | " < " | " ≤ " | " > " | " ≥ " | " ∈ "
simpleExpr = ["+" | "-"] *term* { *addOp term* }
addOp = "+" | "-" | **or**
term = *factor* { *mulOp factor* }
mulOp = "*" | "/" | **div** | **mod** | **and**
factor = *number* | *string* | *setExpression* | *designator* |
"(" *expression* ")" | **not** *factor* |
guard "(" *typeIdent* "," *expression* "," *expression* ")"

setExpression = *simpleSetExpr* [*relSetOp simpleSetExpr*]
relSetOp = " = " | " ≠ " | " ⊂ " | " ⊆ " | " ⊃ " | " ⊇ "
simpleSetExpr = *setTerm* { *addSetOp setTerm* }
addSetOp = **union** | **difference**
setTerm = *setFactor* { *mulSetOp setFactor* }
mulSetOp = **intersect**
setFactor = *set* | *setDesignator* | *setQuery* | "(" *setExpression* ")" |
guard "(" *typeIdent* "," *setExpression* "," *setExpression* ")"

setDesignator = *designator*
set = "{" [*element* { "," *element* }] "}" | ∅
element = *designator* | *objectQuery*

databaseDef = **define database** *dbIdent* ";" *schema* **end** "."
dbIdent = *ident*
schema = { **define** *definition* ";" }
definition = *functionDef* | *typeDef* | *classDef* | *viewDef* | *varDef*

functionDef = **function** *functionIdent* ":" *functionSign*
functionIdent = [*typeIdent* "."] *ident*
functionSign = [*typeIdent*] "→" [**set of**] *typeIdent* [**inverse** *functionIdent*]

typeDef = **type** *typeIdent* [**isa** *superTypeList*] ["=" *functionList*]
typeIdent = *ident*
superTypeList = *superType* { "," *superType* }
superType = *typeIdent*
functionList = *function* { "," *function* }
function = *functionIdent* [":" *functionSign*]

classDef = **class** *classIdent* [":" *typeIdent*] [*selector* [*varIdent* ":"] *superClassList*]
[**where** *booleanExpr*]
classIdent = *ident*
selector = **all** | **some**
superClassList = *superClass* { "," *superClass* }
superClass = *classIdent*
booleanExpr = *expression*

viewDef = **view** *viewIdent* **as** *simpleSetExpr*
viewIdent = *ident*

varDef = **var** *varIdentList* ":" [**set of**] *typeIdent*
varIdentList = *varIdent* { "," *varIdent* }
varIdent = *ident*

COOL: Object Query Language

setQuery = *selection* | *projection* | *extension*
objectQuery = *pick*
selection = **select** "[" *booleanExpr* "]" "(" *boundSetExpr* ")"
boundSetExpr = [*varIdent* ":"] *simpleSetExpr*

projection = **project** "[" *projectList* "]" "(" *boundSetExpr* ")"
projectList = *typeIdent* | *functionIdentList*
functionIdentList = *functionIdent* ["::" *typeIdent*] { "," *functionIdent* ["::" *typeIdent*] }

extension = **extend** "[" *extendList* "]" "(" *boundSetExpr* ")"
extendList = *extend* { "," *extend* }
extend = *derivedFct* | (*functionIdent* ["::" *typeIdent*])
derivedFct = *functionIdent* ":=" *expression*

pick = **pick** "(" *simpleSetExpr* ")"

extraction = **extract** "[" *extractList* "]" "(" *boundSetExpr* ")"
extractList = *column* { "," *column* }
column = [*columnIdent* "="] (*extraction* | *expression*)
columnIdent = *ident*

COOL: Object Manipulation Language

manipulation = *assignment* | *objectEvolution* | *classManipulation*

assignment = *globalAssignment* | *partialAssignment*
globalAssignment = *varIdent* ":=" *expression*
partialAssignment = **set** "[" *functionIdent* ":=" *expression* "]" "(" *objectDesignator* ")"
objectDesignator = *designator*

objectEvolution = *create* | *gain* | *lose* | *delete*
create = **create** "[" *typeIdent* "]" "(" *varIdent* ")"
gain = **gain** "[" *typeIdent* "]" "(" *objectDesignator* ")"
lose = **lose** "[" *typeIdent* "]" "(" *objectDesignator* ")"
delete = **delete** "(" *objectDesignator* ")"

classManipulation = *add* | *remove*
add = **add** "[" *objectDesignator* "]" "(" *classIdent* ")"
remove = **remove** "[" *objectDesignator* "]" "(" *classIdent* ")"

applyToAll = **apply_to_all** "[" *manipList* "]" "(" *boundSetExpr* ")"
manipList = *manipulation* { ";" *manipulation* }

COOL: Database Session

dbSession = *databaseDef* | *databaseUse*
databaseUse = *connect instruction* { ";" *instruction* } *disconnect*
instruction = *extraction* | *manipulation* | *applyToAll* | *definition*
connect = **connect** *dbIdent* ";"
disconnect = **disconnect** *dbIdent* ";"

Appendix B

The Meta Database

In the sequel, we define the COCOON meta database. Its purpose is twofold: (i) to describe the object model using its own notation, and (ii) to represent data dictionary information.

The COCOON meta schema is composed of a minimal set of meta types (*type, object-type, set-type, fcn-type, function, class, class-def, view-def*), meta functions, and meta classes (*Types, Set-Types, Fcn-Types, Functions, Classes, Class-Defs, View-Defs*). A graphical overview is given in Figure B.1 below.

Meta information of object-oriented systems have been used for a variety of other applications. I.e. [HM90] use meta models to describe the semantics of object-oriented data models, such that data models can be compared. [GH91] describe a meta class system and possible schema level operations. [Kla90] uses meta classes for defining an open and extensible object model.

B.1 Meta Types and Meta Functions

The first meta type represents data and object types. That is, each COCOON type is represented by an object, being instance of the following meta type:

```
type type  isa object =
           tname :   string ,            // name of the type
           functs :  set of function ;   // functions applicable to the type's instances
```

Most types are defined by users in order to specify the interface of an abstract object type (i.e. to define the signatures of the applicable functions). As usual in object-oriented systems, such types can be ordered in type-hierarchies. The meta type *object-type* is a specialized subtype:

```
type object-type  isa type =
                  localf :   set of function ,     // set of local functions,
                                                      defined independent of inheritance
                  supert :   set of object-type ;  // set of explicitly defined
                                                      supertypes of the type
```

Whereas *localf(t)* are the functions defined to be applicable to t's instances independent of inheritance, for an abstract object type, the set of all applicable functions *functs(t)*

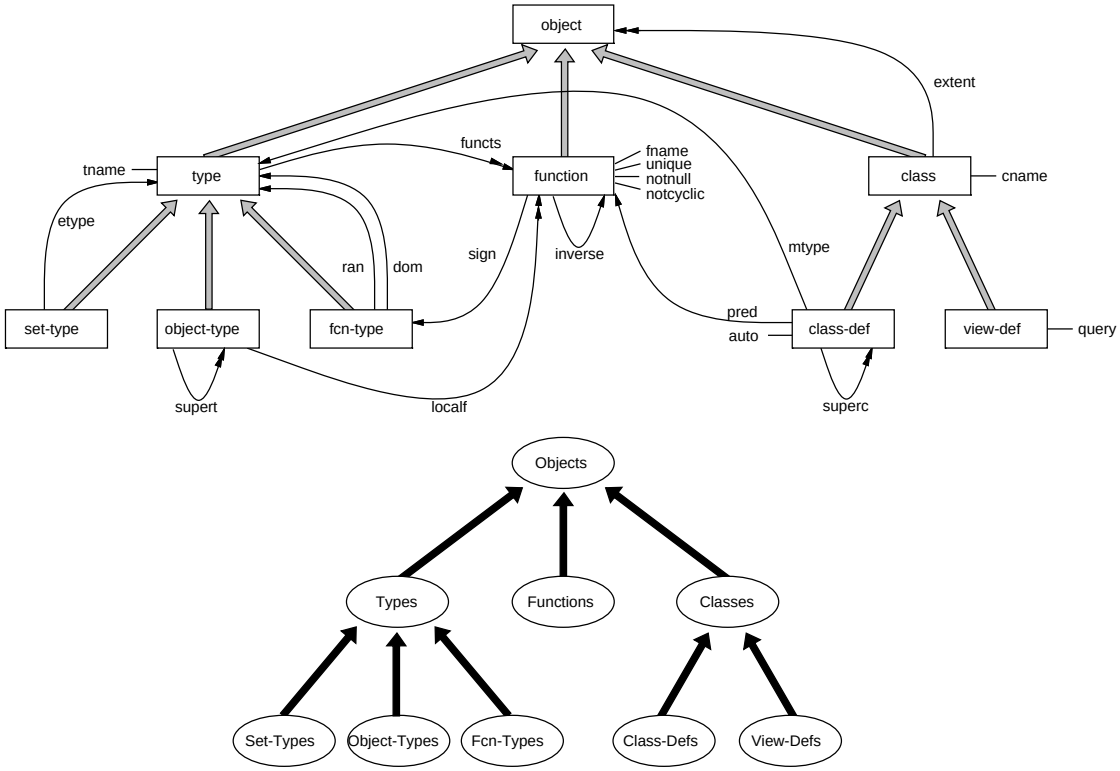


Figure B.1: The meta schema is a hierarchy of meta types with their meta functions, and a hierarchy of meta classes.

is derived by the union of the local functions *localf(t)* and the functions inherited from the supertypes:

$$\mathit{functs}(t) := \mathit{localf}(t) \bigcup_{t_i \in \mathit{supert}(t)} \mathit{functs}(t_i)$$

Notice, that type checking is based on all functions *functs(t)*, not only on the local ones. We distinguish for each type t between explicit and implicit supertypes. The former ones are those explicitly assigned with the meta function *supert(t)*, whereas the implicit ones are derived from the set of applicable functions as follows:
$t \preceq t' \iff \mathit{functs}(t) \supseteq \mathit{functs}(t')$
That is, a type t' is supertype of t, if the applicable functions are a superset the functions of t'.

In addition to abstract object types, two more subtypes represent constructed data types: the set and function types. Since these types are normally created and managed internally by the system, most of them are unnamed.

```
type set-type isa type =   etype : type ;   // type of the elements in the set

type fcn-type  isa type =
                  dom :  type ,   // domain type of the function
                  ran :  type ;   // range type of the function
```

The second meta type represents COCOON functions. They are named, have a signature, and their values can be restricted by a set of constraints.

```
type function  isa object  =
                fname :    string ,                        // name of the function
                sign :     fcn-type ,                      // signature of the function
                inverse :  function inverse inverse ;      // inverse function constraint
```

Information about the implementation of the function (e.g. whether the function result is stored or computed), is intentionally excluded from the meta schema, since this is irrelevant for schema evolution.

The third meta type represents COCOON classes.

```
type class  isa object  =
             cname :   string ,          // name of the class
             extent :  set of object ;   // set of actual class members
```

Since we treat views as classes with implicitly defined type and extension, two subtypes of meta type *class* are distinguished: those defined as a class, and those defined as a view.

```
type class-def  isa class  =
                 auto :    boolean ,        // is the class extent defined sufficiently?
                 mtype :   type ,           // explicit member type of the class
                 pred :    function ,       // class predicate, a boolean function
                 superc :  set of class ,   // explicitly defined super classes
                 pmemb :   set of object ;  // set of potential class member objects
```

```
type view-def  isa class  =
                query :  expression ,   // query expression defining the view
```

The value of *auto(c)* is true, iff the class c is defined with the selector **all**. In these cases the system can decide whether an object belongs to the extent of a class. If classes are defined by the selector **some** there are just necessary conditions defined. The information about class membership is specified by the user in terms of adding and removing objects to/from a class respectively. This information is stored by the set *pmemb* that represents the potential members of a class. These are objects that are added to a class, but need not to fulfill the class predicate (for more detail see [LS92]). *extent(c)* derives the actual set of member objects (extent) of a class, i.e. a subset of *pmemb* which elements fulfill the class predicate. The actual derived member type of the class objects is either equal to *mtype(c)*, if a member type is explicitly defined, or otherwise, it must be derived from the member type of c's superclasses and class predicate.

B.2 Meta Classes and Meta Views

Together with each meta type, there is a meta class holding the actual instances of the meta type.

```
class Types : type some Objects ;
class Set-Types : set-type some Types ;
class Fcn-Types : fcn-type some Types ;
class Object-Types : object-type some Types ;
```

class *Functions* : *function* **some** *Objects* ;

class *Classes* : *class* **some** *Objects* ;
class *Class-Defs* : *class-def* **some** *Classes* ;
class *View-Defs* : *view-def* **some** *Classes* ;

In addition, the following view collects all classes with implicitly defined extent. That is, the view-defined classes and the class-defined ones with an **all** -selector:

view *Views* **as** *View-Defs* **union select** [*auto(c)*] (*c: Class-Defs*);