21 Object Databases

Minkisi are complex objects clearly not the product of a momentary impulse. . . . To do justice to objects, a theory of them must be as complex as them.[1]

—Wyatt MacGaffey in *Astonishment and Power*

Alice:	*What is a Minkisi?*
Sergio:	*It is an African word that translates somewhat like "things that do things."*
Vittorio:	*It is art, religion, and magic.*
Riccardo:	*Oh, this sounds to me very object oriented!*

In this chapter, we provide a brief introduction to object-oriented databases (OODBs). A complete coverage of this new and exciting area is beyond the scope of this volume; we emphasize the new modeling features of OODBs and some of the preliminary theoretical research about them. On the one hand, we shall see that some of the most basic issues concerning OODBs, such as the design of query languages or the analysis of their expressive power, can be largely resolved using techniques already developed in connection with the relational and complex value models. On the other hand, the presence of new features (such as object identifiers) and methods brings about new questions and techniques.

As mentioned previously, the simplicity of the data structure in the relational model often hampers its use in many database applications. A relational representation can obscure the intention and intricate semantics of a complex data structure (e.g., for holding the design of a VLSI chip or an airplane wing). As we shall see, OODBs remedy this situation by borrowing a variety of data structuring constructs from the complex value model (Chapter 20) and from semantic data models (considered in Chapter 11). At a more fundamental level, the relational data model and all of the data models presented so far impose a sharp distinction between data storage and data processing: The DBMS provides data storage, but data processing is provided by a host programming language with a relatively simple language such as SQL embedded in it. OODBs permit the incorporation of behavioral portions of the overall data management application directly into the database schema, using methods in the sense of object-oriented programming languages.

This chapter begins with an informal presentation of the underlying constructs of OODBs. Next a formal definition for a particular OODB model is presented. Two directions of theoretical research into OODBs are then discussed. First a family of languages

[1] Reprinted with permission. Smithsonian Institution Press ©1993.

for data access is presented, with an emphasis on how the languages interact with the novel modeling constructs (of particular interest is the impact of generalizing the notion of complete query language to accommodate the presence of object identifiers, OIDs). Next two languages for methods are described. The first is an imperative language allowing us to specify methods with side effects.[2] The second language brings us to a functional perspective on methods and database languages and allows us to specify side-effect-free methods. In both cases, we present some results on type safety and expressive power. Checking type safety is generally undecidable; we identify a significant portion of the functional language, monadic method schemas, for which type safety is decidable. With respect to expressive power, the imperative language is complete in an extended sense formalized in this chapter. The functional language expresses precisely QPTIME on ordered inputs and so turns out to express the by-now-famous *fixpoint* queries. The chapter concludes with a brief survey of additional research issues raised by OODBs.

21.1 Informal Presentation

Object-oriented database models stem from a synthesis of three worlds: the complex value model, semantic database models, and object-oriented programming concepts. At the time of writing, there is not widespread agreement on a specific OODB model, nor even on what components are required to constitute an OODB model. In this section, we shall focus on seven important ingredients of OODB models:

1. objects and object identifiers;
2. complex values and types;
3. classes;
4. methods;
5. ISA hierarchies;
6. inheritance and dynamic binding;
7. encapsulation.

In this section, we describe and illustrate these interrelated notions informally; a more formal definition is presented in the following section. We will also briefly discuss alternatives.

As a running example for this discussion, we shall use the OODB schema specified in Fig. 21.1. This schema is closely related to the semantic data model schema of Fig. 11.1, which in turn is closely related to the **CINEMA** example of Chapter 3.

As discussed in Chapter 11, a significant shortcoming of the relational model is that it must use printable values, often called *keys*, to refer to entities or objects-in-the-world. As a simple example, suppose that the first and last names of a person are used as a key to identify that person. From a physical point of view, it is then cumbersome to refer to a person, because the many bytes of his or her name must be used. A more fundamental

[2] Methods are said to have side-effects if they cause updates to the database.

```
(* schema and base definitions *)

create schema PariscopeSchema ;
create base PariscopeBase;

(* class definitions *)

class Person
   type tuple ( name: string, citizenship: string, gender: string );
class Director inherit Person
   type tuple ( directs: set ( Movie ) );
class Actor inherit Person
   type tuple ( acts_in: { Movie },
         award: { tuple ( prize: string, year: integer ) } );
class Actor_Director inherit Director, Actor
class Movie
   type tuple ( title: string, actors: set ( Actor );
         director: Director );
class Theater
   type tuple ( name: string, address: string, phone: string );

(* name definitions *)

name Pariscope: set ( tuple ( theater: Theater, time: string, price: integer,
         movie: Movie ) );
name Persons_I_like: set ( Person );
name Actors_I_like, Actors_you_like: set ( Actor );
name My_favorite_director : Director

(* method definitions *)

method get_name in class Person : string
   { if (gender = “male”)
         return “Mr.” + self.name;
      else
         return “Ms.” + self.name }

method get_name in class Director : string
   { return ( “Director” + self.name ) };

method get_name in class Actor_Director : string
   { return ( “Director” + self.name ) };

(* we assume here that ‘+’ denotes a string concatenation operator *)
```

Figure 21.1: An OODB Schema

problem arises if the person changes his or her name (e.g., as the result of marriage). When performing this update, conceptually there is a break in the continuity in the representation of the person. Furthermore, care must be taken to update all tuples (typically arising in a number of different relations) that refer to this person, to reflect the change of name.

Following the spirit of semantic data models, OODB models permit the explicit representation of physical and conceptual objects through the use of object identifiers (OIDs). Conceptually, a unique OID is assigned to each object that is represented in the database, and this association between OID and object remains fixed, even as attributes of the object (such as name or age) change in value. The use of objects and OIDs permits OODBs to share information gracefully; a given object o is easily shared by many other objects simply by referencing the OID of o. This is especially important in the context of updates; for example, the name of a person object o need be changed in only one place even if o is shared by many parts of the database.

In an OODB, a complex value is associated with each object. This complex value may involve printables and/or OIDs (i.e., references to the same or other objects). For example, each object in the class *Movie* in Fig. 21.1 has an associated triple whose second coordinate contains a set of OIDs corresponding to actors. In this section, we focus on complex values constructed using the tuple and set construct. In practical OODB models, other constructs are also supported (including, for example, bags and lists). Some commercial OODBs are based on an extension of C++ that supports persistence; in these models essentially any C++ structure can serve as the value associated with an object.

Objects that have complex values with the same type may be grouped into classes, as happens in semantic data models. In the running example, these include *Person*, *Director*, and *Movie*. Classes also serve as a natural focal point for associating some of the behavioral (or procedural) components of a database application. This is accomplished by associating with each class a family of methods for that class. Methods might be simple (e.g., producing the name of a person) or arbitrarily complex (e.g., displaying a representation of an object to a graphical interface or performing a stress analysis of a proposed wing design). A method has a name, a signature, and an implementation. The name and signature serve as an external interface to the method. The implementation is typically written in a (possibly extended) programming language such as C or C++. The choice of implementation language is largely irrelevant and is generally not considered to be part of the data model.

As with semantic models, OODB models permit the organization of classes into a hierarchy based on what have been termed variously ISA, specialization, or class-subclass relationships. The term *hierarchy* is used loosely here: In many cases any directed acyclic graph (DAG) is permitted. In Fig. 21.1 the ISA hierarchy has *Director* and *Actor* as (immediate) specializations of *Person* and *Actor_Director* as a specialization of both *Director* and *Actor*. Following the tradition of object-oriented programming languages, a virtual class **any** is included that serves as the unique root of the ISA hierarchy.

In OODB models, there are two important implications of the statement that class c' is a subclass of c. First it is required that the complex value type associated with c' be a subtype (in the sense formally defined later) of the complex value type associated with c. Second it is required that if there is a method with name m associated with c, then there is also a method with name m associated with c'. In some cases, the implementation (i.e., the actual code) of m for c' is identical to that for c; in this case the code of m for c' need not be explicitly specified because it is *inherited* from c. In other cases, the implementation of m for c' is different from that for c; in which case we say that the implementation of m for c' *overrides* the implementation of m for c. (See the different implementations for method *get_name* in Fig. 21.1.) The determination of what implementation is associated with a given method name and class is called *method resolution*. A method is invoked with respect to an object o, and the class to which o belongs determines which implementation is to be used. This policy is called *dynamic binding*. As we shall see, the interaction of method calls and dynamic binding in general makes type checking for OODB schemas undecidable. (It is undecidable to check whether such a schema would lead to a runtime type error; on the other hand, it is clearly possible to find decidable sufficient conditions that will guarantee that no such error can arise.)

In the particular OODB model presented here, both values (in the style of complex values) and objects are supported. For example, in Fig. 21.1 a persistent set of triples called *Pariscope* is supported (see also Fig. 11.1). The introduction of values not directly associated with OIDs is a departure from the tradition of object-oriented programming, and not all OODBs in the literature support it. However, in databases the use of explicit values often simplifies the design and use of a schema. Their presence also facilitates expressing queries in a declarative manner.

The important principle of encapsulation in object orientation stems from the field of abstract data types. Encapsulation is used to provide a sharp boundary between how information about objects is accessed by database users and how that information is actually stored and provided. The principle of encapsulation is most easily understood if we distinguish two categories of database use: *dba mode*, which refers to activities unique to database administrators (including primarily creating and modifying the database schema), and *user mode*, which refers to activities such as querying and updating the actual data in the database. Of course, some users may operate in both of these modes on different occasions. In general, application software is viewed as invoked from the user mode.

Encapsulation requires that when in user mode, a user can access or modify information about a given object only by means of the methods defined for that object; he or she cannot directly examine or modify the complex value or the methods associated with the object. In particular, then, essentially all application software can access objects only through their methods. This has two important implications. First, as long as the same set of methods is supported, the underlying implementation of object methods, and even of the complex value representation of objects, can be changed without having to modify any application software. Second, the methods of an object often provide a focused and abstracted interface to the object, thus making it simpler for programmers to work with the objects.

In object-oriented programming languages, it is typical to enforce encapsulation except in the special case of rewriting method implementations. In some OODB models, there is an important exception to this in connection with query languages. In particular, it is generally convenient to permit a query language to examine explicitly the complex values associated with objects.

The reader with no previous exposure to object-oriented languages may now be utterly overwhelmed by the terminology. It might be helpful at this point to scan through a book or manual about an object-oriented programming language such as C++, or an OODB such as O_2 or ObjectStore. This will provide numerous examples and the overall methodology of object-oriented programming, which is beyond the scope of this book.

21.2 Formal Definition of an OODB Model

This section presents a formal definition of a particular OODB model, called the *generic OODB model*. (This model is strongly influenced by the IQL and O_2 models. Many features are shared by most other OODB models. While presenting the model, we also discuss different choices made in other models.) The presentation essentially follows the preceding informal one, beginning with definitions for the types and class hierarchy and then introducing methods. It concludes with definitions of OODB schema and instance.

Types and Class Hierarchy

The formal definitions of object, type, and class hierarchy are intertwined. An object consists of a pair (*identifier*, *value*). The identifiers are taken from a specific sort containing OIDs. The values are essentially standard complex values, except that OIDs may occur within them. Although some of the definitions on complex values and types are almost identical to those in Chapter 20, we include them here to make precise the differences from the object-oriented context. As we shall see, the class hierarchy obeys a natural restriction based on subtyping.

To start, we assume a number of atomic types and their pairwise disjoint corresponding domains: **integer**, **string**, **bool**, **float**. The set **dom** of atomic values is the (disjoint) union of these domains; as before, the elements of **dom** are called *constants*. We also assume an infinite set $\mathbf{obj} = \{o_1, o_2, \ldots\}$ of *object identifiers* (OIDs), a set **class** of class names, and a set **att** of *attribute names*. A special constant nil represents the undefined (i.e., null) value.

Given a set O of OIDs, the family of *values* over O is defined so that

(a) nil, each element of **dom**, and each element of O are values over O; and

(b) if $v_1, \ldots, v_n$ are values over O, and $A_1, \ldots, A_n$ distinct attributes names, the tuple $[A_1 : v_1, \ldots, A_n : v_n]$ and the set $\{v_1, \ldots, v_n\}$ are values over O.

The set of all values over O is denoted $\mathbf{val}(O)$. An *object* is a pair (o, v), where o is an OID and v a value.

In general, object-oriented database models also include constructors other than tuple and set, such as list and bag; we do not consider them here.

Example 21.2.1 Letting *oid*7, *oid*22, etc. denote OIDs, some examples of values are as follows:

$$[\mathit{theater} : \mathit{oid}7,\ \mathit{time} : \text{“16:45”},\ \mathit{price} : 45,\ \mathit{movie} : \mathit{oid}22]$$

$$\{\text{“H. Andersson”}, \text{“K. Sylwan”}, \text{“I. Thulin”}, \text{“L. Ullman”}\}$$

$$[\mathit{title} : \text{“The Trouble with Harry”}, \mathit{director} : \mathit{oid}77,$$
$$\mathit{actors} : \{\mathit{oid}81, \mathit{oid}198, \mathit{oid}265, \mathit{oid}77\}]$$

An example of an object is

$$(oid22\ ,\ [title: \text{“The Trouble with Harry”},\ director: oid77,$$
$$actors: \{oid81, oid198, oid265, oid77\}])$$

As discussed earlier, objects are grouped in classes. All objects in a class have complex values of the same type. The type corresponding to each class is specified by the OODB schema.

Types are defined with respect to a given set C of class names. The family of *types* over C is defined so that

1. **integer, string, bool, float,** are types;
2. the class names in C are types;
3. if τ is a type, then[3] $\{\tau\}$ is a (set) type;
4. if $\tau_1, \ldots, \tau_n$ are types and $A_1, \ldots, A_n$ distinct attribute names, then $[A_1 : \tau_1, \ldots, A_n : \tau_n]$ is a (tuple) type.

The set of types over C together with the special class name **any** are denoted **types**(C). (The special name **any** is a type but may not occur inside another type.) Observe the close resemblance with types used in the complex value model.

Example 21.2.2 An example of a type over the classes of the schema in Fig. 21.1 is

$$[name: \textbf{string},\ citizenship: \textbf{string},\ gender: \textbf{string}]$$

One may want to give a name to this type (e.g., *Person_type*). Other examples of types (with names associated to them) include

$$\begin{aligned}
Director\_type &= [name: \textbf{string},\ citizenship: \textbf{string},\ gender: \textbf{string},\\
&\qquad directs: \{Movie\}]\\
Theater\_type &= [name: \textbf{string},\ address: \textbf{string},\ phone: \textbf{string}]\\
Pariscope\_type &= [theater: Theater,\ time: \textbf{string},\ price: \textbf{integer},\ movie: Movie]\\
Movie\_type &= [title: \textbf{string},\ actors: \{Actor\},\ director: Director]\\
Award\_type &= [prize: \textbf{string},\ year: \textbf{integer}]
\end{aligned}$$

In an OODB schema we associate with each class c a type $\sigma(c)$, which dictates the type of objects in this class. In particular, for each object (o, v) in class c, v must have the exact structure described by $\sigma(c)$.

[3] In Fig. 21.1 we use keywords **set** and **tuple** as syntactic sugar when specifying the set and tuple constructors.

Recall from the informal description that an OODB schema includes an ISA hierarchy among the classes of the schema. The class hierarchy has three components: (1) a set of classes, (2) the types associated with these classes, and (3) a specification of the ISA relationships between the classes. Formally, a *class hierarchy* is a triple $(C, \sigma, \prec)$, where C is a finite set of class names, σ a mapping from C to $\mathbf{types}(C)$, and $\prec$ a partial order on C.

Informally, in a class hierarchy the type associated with a subclass should be a refinement of the type associated with its superclass. For example, a class *Student* is expected to refine the information on its superclass *Person* by providing additional attributes. To capture this notion, we use a subtyping relationship ($\leq$) that specifies when one type refines another.

Definition 21.2.3 Let $(C, \sigma, \prec)$ be a class hierarchy. The *subtyping relationship* on $\mathbf{types}(C)$ is the smallest partial order $\leq$ over $\mathbf{types}(C)$ satisfying the following conditions:

(a) if $c \prec c'$, then $c \leq c'$;

(b) if $\tau_i \leq \tau_i'$ for each $i \in [1, n]$ and $n \leq m$, then $[A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m] \leq [A_1 : \tau_1', \ldots, A_n : \tau_n']$;

(c) if $\tau \leq \tau'$, then $\{\tau\} \leq \{\tau'\}$; and

(d) for each τ, $\tau \leq \mathbf{any}$ (i.e., **any** is the top of the hierarchy).

A class hierarchy $(C, \sigma, \prec)$ is *well formed* if for each pair c, c' of classes, $c \prec c'$ implies $\sigma(c) \leq \sigma(c')$.

By way of illustration, it is easily verified that

$$\textit{Director\_type} \leq \textit{Person\_type} \qquad \textit{Director\_type} \not\leq \textit{Movie\_type}.$$

Thus the schema obtained by adding the constraint *Director* $\prec$ *Movie* would not be well formed.

Henceforth we consider only well-formed class hierarchies.

Example 21.2.4 Consider the class hierarchy $(C, \sigma, \prec)$ of the schema of Fig. 21.1. The set of classes is

$$C = \{\textit{Person}, \textit{Director}, \textit{Actor}, \textit{Actor\_Director}, \textit{Theater}, \textit{Movie}\}$$

with *Actor* $\prec$ *Person*, *Director* $\prec$ *Person*, *Actor_Director* $\prec$ *Director*, *Actor_Director* $\prec$ *Actor*, and (referring to Example 21.2.2 for the definitions of *Person_type*, *Theater_type*, etc.)

$$
\begin{aligned}
\sigma(\mathit{Person}) &= \mathit{Person\_type},\\
\sigma(\mathit{Theater}) &= \mathit{Theater\_type},\\
\sigma(\mathit{Movie}) &= \mathit{Movie\_type},\\
\sigma(\mathit{Director}) &= \mathit{Director\_type},\\
\sigma(\mathit{Actor}) &= [\mathit{name} : \mathbf{string},\ \mathit{citizenship} : \mathbf{string},\\
&\quad \mathit{gender} : \mathbf{string},\ \mathit{acts\_in} : \{\mathit{Movie}\},\\
&\quad \mathit{award} : \{\mathit{Award\_type}\}]\\
\sigma(\mathit{Actor\_Director}) &= [\mathit{name} : \mathbf{string},\ \mathit{citizenship} : \mathbf{string},\\
&\quad \mathit{gender} : \mathbf{string},\ \mathit{acts\_in} : \{\mathit{Movie}\},\\
&\quad \mathit{award} : \{\mathit{Award\_type}\},\ \mathit{directs} : \{\mathit{Movie}\}]
\end{aligned}
$$

The use of type names here is purely syntactic. We would obtain the same schema if we replaced, for instance, *Person_type* with the value of this type.

Observe that $\sigma(\mathit{Director}) \leq \sigma(\mathit{Person})$ and $\sigma(\mathit{Actor}) \leq \sigma(\mathit{Person})$, etc.

The Structural Semantics of a Class Hierarchy

We now describe how values can be associated with the classes and types of a class hierarchy. Because the values in an OODB instance may include OIDs, the semantics of classes and types must be defined simultaneously. The basis for these definitions is the notion of OID assignment, which assigns a set of OIDs to each class.

Definition 21.2.5 Let $(C, \sigma, \prec)$ be a (well-formed) class hierarchy. An *OID assignment* is a function π mapping each name in C to a disjoint finite set of OIDs. Given OID assignment π, the *disjoint extension* of c is $\pi(c)$, and the *extension* of c, denoted $\pi^*(c)$, is $\cup\{\pi(c') \mid c' \in C, c' \prec c\}$.

If π is an OID assignment, then $\pi^*(c') \subseteq \pi^*(c)$ whenever $c' \prec c$. This should be understood as a formalization of the fact that an object of a subclass c' may be viewed also as an object of a superclass c of c'. From the perspective of typing, this suggests that operations that are type correct for members of c are also type correct for members of c'.

Unlike the case for many semantic data models, the definition of OID assignment for OODB schemas implies that extensions of classes of an ISA hierarchy without common subclasses are necessarily disjoint. In particular, extensions of all leaf classes of the hierarchy are disjoint (see Exercise 21.2). This is a simplifying assumption that makes it easier to associate objects to classes. There is a unique class to whose disjoint extension each object belongs.

The semantics for types is now defined relative to a class hierarchy $(C, \sigma, \prec)$ and an OID assignment π. Let $O = \cup\{\pi(c) \mid c \in C\}$, and define $\pi(\mathbf{any}) = O$. The *disjoint interpretation* of a type τ, denoted $dom(\tau)$, is given by

(a) for each atomic type τ, $dom(\tau)$ is the usual interpretation of that type;

(b) $dom(\mathbf{any})$ is $\mathbf{val}(O)$;

(c) for each $c \in C$, $dom(c) = \pi^*(c) \cup \{nil\}$;

(d) $dom(\{\tau\}) = \{\{v_1, \ldots, v_n\} \mid n \geq 0, \text{ and } v_i \in dom(\tau), i \in [1, n]\}$; and

(e) $dom([A_1 : \tau_1, \ldots, A_k : \tau_k]) = \{[A_1 : v_1, \ldots, A_k : v_k] \mid v_i \in dom(\tau_i), i \in [1, k]\}$.

Remark 21.2.6 In the preceding interpretation, the type determines precisely the structure of a value of that type. It is interesting to replace (e) by

$$
\begin{array}{ll}
 & dom([A_1 : \tau_1, \ldots, A_k : \tau_k]) = \\
(\mathrm{e}') & \{[A_1 : v_1, \ldots, A_k : v_k, A_{k+1} : v_{k+1}, \ldots A_l : v_l] \mid \\
 & v_i \in dom(\tau_i), i \in [1, k], v_j \in \mathbf{val}(O), j \in [k+1, l]\}.
\end{array}
$$

Under this alternative interpretation, for each τ, τ' in $\mathbf{types}(C)$, if $\tau' \leq \tau$ then $dom(\tau') \subseteq dom(\tau)$. This is why this is sometimes called the *domain-inclusion semantics*. From a data model viewpoint, this presents the disadvantage that in a correctly typed database instance, a tuple may have a field that is not even mentioned in the database schema. For this reason, we do not adopt the domain-inclusion semantics here. On the other hand, from a linguistic viewpoint it may be useful to adopt this more liberal semantics in languages to allow variables denoting tuples with more attributes than necessary. ■

Adding Behavior

The final ingredient of the generic OODB model is *methods*. A method has three components:

(a) a *name*

(b) a *signature*

(c) an *implementation* (or *body*).

There is no problem in specifying the names and signatures of methods in an OODB schema. To specify the implementation of methods, a language for methods is needed. We do not consider specific languages in the generic OODB model. Therefore only names and signatures of methods are specified at the schema level in this model. In Section 21.4, we shall consider several languages for methods and shall therefore be able to add the implementation of methods to the schema.

Without specifying the implementation of methods, the generic OODB model specifies their semantics (i.e., the effect of each method in the context of a given instance). This effect, which is a function over the domains of the types corresponding to the signature of the method, is therefore specified at the instance level.

We assume the existence of an infinite set **meth** of method names. Let $(C, \sigma, \prec)$ be a class hierarchy. For method name m, a *signature* of m is an expression of the form $m : c \times \tau_1 \times \cdots \times \tau_{n-1} \to \tau_n$, where c is a class name in C and each τ_i is a type over C. This signature is associated with the class c; we say that method *m applies* to objects of class c and to objects of classes that inherit m from c. It is common for the same method name to have different signatures in connection with different classes. (Some restrictions shall be specified later.) The notion of signature here generalizes the one typically found in object-oriented programming languages, because we permit the τ_i's to be types rather than only classes.

It is easiest to describe the notions of overloading, method inheritance, and dynamic binding in terms of an example. Consider the methods defined in the schema of Fig. 21.1. All three share the name *get_name*. The signatures are given by

$$\begin{aligned}&\mathit{get\_name} : \mathit{Person} \rightarrow \textbf{string}\\&\mathit{get\_name} : \mathit{Director} \rightarrow \textbf{string}\\&\mathit{get\_name} : \mathit{Actor\_Director} \rightarrow \textbf{string}\end{aligned}$$

Note that *get_name* has different implementations for these classes; this is an example of *overloading* of a method name.

Recall that *Actor* is a subclass of *Person*. According to the informal discussion, if *get_name* applies to elements of *Person*, then it should also apply to members of *Actor*. Indeed, in the object-oriented paradigm, if a method m is defined for a class c but not for a subclass c' of c (and it is not defined anywhere else along a path from c' to c), then the definition of m for c' is *inherited* from c. In particular, the signature of m on c' is identical to the one of m for c, except that the first c is replaced by c'. The implementation of m for c' is identical to that for c. In the schema of Fig. 21.1, the signature of *get_name* for *Actor* is

$$\mathit{get\_name} : \mathit{Actor} \rightarrow \textbf{string}$$

and the implementation is identical to the one for *Person*. The determination of the correct method implementation to use for a given method name m and class c is called *method resolution*; the selected implementation is called the *resolution* of m for c.

Suppose that π is an OID assignment, that $oid25$ is in the extension $\pi^*(\mathit{Person})$ of *Person*, and that *get_name* is called on $oid25$. What implementation of *get_name* will be used? In our OODB model we shall use *dynamic binding* (also called *late binding*, or *value-dependent binding*). This means that the specific implementation chosen for *get_name* on $oid25$ depends on the most specific class that $oid25$ belongs to, that is, the class c such that $oid25 \in \pi(c)$.

(An alternative to dynamic binding is *static binding*, or *context-dependent binding*. Under this discipline, the implementation used for *get_name* depends on the type associated with the variable holding $oid25$ at the point in program where *get_name* is invoked. This can be determined at compile time, and so static binding is generally much cheaper than dynamic binding. In the language C++, the default is static binding, but dynamic binding can be obtained by using the keyword **virtual** when specifying the method.)

Consider a call $m(o, v_1, \ldots, v_{n-1})$ to method m. This is often termed a *message*, and o is termed the *receiver*. As described here, the implementation of m associated with this message depends exclusively on the class of o. To emphasize the importance of the receiver for finding the actual implementation, in some languages the message is denoted $o \rightarrow m[v_1, \ldots v_{n-1}]$. In some object-oriented programming languages, such as CommonLoops (an object-oriented extension of LISP), the implementation depends on

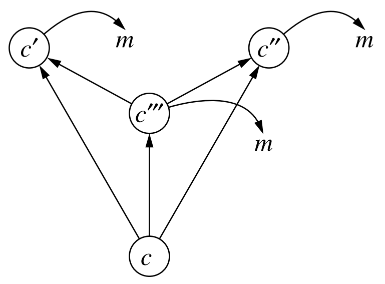

Figure 21.2: Unambiguous definition

all of the parameters of the call, not just the first. This is also the approach of the method schemas introduced in Section 21.4.

The set of methods applicable to an object is called the *interface* of the object. As noted in the informal description of OODB models, in most cases objects are accessed only via their interface; this philosophy is called *encapsulation*.

As part of an OODB schema, a set M of method signatures is associated to a class hierarchy $(C, \sigma, \prec)$. Note that a signature $m : c \times \tau_1 \times \cdots \times \tau_{n-1} \rightarrow \tau_n$ can be viewed as giving a particular meaning to m for class c, at least at a syntactic level. Because of inheritance, a meaning for method m need not be given explicitly for each class of C nor even for subclasses of a class for which m has been given a meaning. However, we make two restrictions on the family of method signatures: The set M is *well formed* if it obeys the following two rules:

Unambiguity: If c is a subclass of c' and c'' and there is a definition of m for c' and c'', then there is a definition of m for a subclass of c' and c'' that is either c itself, or a superclass of c. (See Fig. 21.2.)

Covariance[4]*:* If $m : c \times \tau_1 \times \cdots \times \tau_n \rightarrow \tau$ and $m : c' \times \tau'_1 \times \cdots \times \tau'_m \rightarrow \tau'$ are two definitions and $c \prec c'$, then $n = m$ for each i, $\tau_i \leq \tau'_i$ and $\tau \leq \tau'$.

The first rule prevents ambiguity resulting from the presence of two method implementations both applicable for the same object. A primary motivation for the second rule is intuitive: We expect the argument and result types of a method on a subclass to be more refined than those of the method on a superclass. This also simplifies the writing of type-correct programs, although type checking leads to difficulties even in the presence of the covariance assumption (see Section 21.4).

Database Schemas and Instances

We conclude this section by presenting the definitions of schemas and instances in the generic OODB model. An important subtlety here will be the role of OIDs in instances

[4] In type theory, *contravariance* is used instead. Contravariance is the proper notion when functions are passed as arguments, which is not the case here.

as placeholders; as will be seen, the specific OIDs present in an instance are essentially irrelevant.

As indicated earlier, a schema describes the structure of the data that is stored in a database, including the types associated with classes and the ISA hierarchy and the signature of methods (i.e., the interfaces provided for objects in each class).

In many practical OODBs, it has been found convenient to allow storage of complex values that are not associated with any objects and that can be accessed directly using some name. This also allows us to subsume gracefully the capabilities of value-based models, such as relations and complex values. It also facilitates writing queries. To reflect this feature, we allow a similar mechanism in schemas and instances. Thus schemas may include a set of value names with associated types. Instances assign values of appropriate type to the names. Method implementations, external programming languages, and query languages may all use these names (to refer to their current values) or a class name (to refer to the set of objects currently residing in that class). In this manner, named values and class names are analogous to relation names in the relational model and to complex value relation names in the complex value model.

In the schema of Fig. 21.1, examples of named values are *Pariscope* (holding a set of triples); *Persons_I_like*, *Actors_I_like*, and *Actors_you_like* (referring to sets of person objects and actor objects; and, finally, *My_favorite_director* (referring to an individual object as opposed to a set). These names can be used explicitly in method implementations and in external query and programming languages.

We now have the following:

Definition 21.2.7 A *schema* is a 5-tuple $\mathbf{S} = (C, \sigma, \prec, M, G)$ where

- G is a set of *names* disjoint from C;
- σ is a mapping from $C \cup G$ to $\mathbf{types}(C)$;
- $(C, \sigma, \prec)$ is a well-formed class hierarchy[5]; and
- M is a well-formed set of method signatures for $(C, \sigma, \prec)$.

An instance of an OODB schema populates the classes with OIDs, assigns values to these OIDs, gives meaning to the other persistent names, and assigns semantics to method signatures. The semantics of method signatures are mappings describing their effect. From a practical viewpoint, the population of the classes, the values of objects, and the values of names are kept extensionally; whereas the semantics of the methods are specified by pieces of code (intensionally). However, we ignore the code of methods for the time being.

Definition 21.2.8 An *instance* of schema $(C, \sigma, \prec, M, G)$ is a 4-tuple $\mathbf{I} = (\pi, \nu, \gamma, \mu)$, where

(a) π is an OID assignment (and let $O = \cup\{\pi(c) \mid c \in C\}$);

(b) ν maps each OID in O to a value in $\mathbf{val}(O)$ of correct type [i.e., for each c and $o \in \pi(c)$, $\nu(o) \in dom(\sigma(c))$];

[5] By abuse of notation, we use here and later σ instead of $\sigma|_C$.

(c) γ associates to each name in G of type τ a value in $dom(\tau)$;

(d) μ assigns semantics to method names in agreement with the method signatures in M. More specifically, for each signature $m : c \times \vec{\alpha} \to \tau$,

$$\mu(m : c \times \vec{\alpha} \to \tau) : dom(c \times \vec{\alpha}) \to dom(\tau);$$

that is, $\mu(m : c \times \vec{\alpha} \to \tau)$ is a partial function from $dom(c \times \vec{\alpha})$ to $dom(\tau)$.

Recall that a method m can occur with different signatures in the same schema. The mapping μ can assign different semantics to each signature of m. The function $\mu(m : c \times \vec{\alpha} \to \tau)$ is only relevant on objects associated with c and subclasses of c for which m is not redefined.

In the preceding definitions, the assignment of semantics to method signatures is included in the instance. As will be seen in Section 21.4, if method implementations are included in the schema, they induce the semantics of methods at the instance level (this is determined by the semantics of the particular programming language used in the implementation).

Intuitively, it is generally assumed that elements of the atomic domains have universally understood meaning. In contrast, the actual OIDs used in an instance are not relevant. They serve essentially as placeholders; it is only their relationship with other OIDs and constants that matters. This arises in the practical perspective in two ways. First, in most practical systems, OIDs cannot be explicitly created, examined, or manipulated. Second, in some object-oriented systems, the actual OIDs used in a physical instance may change over the course of time (e.g., as a result of garbage collection or reclustering of objects).

To capture this aspect of OIDs in the formal model, we introduce the notion of OID isomorphism. Two instances **I**, **J** are *OID isomorphic*, denoted $\mathbf{I} \equiv_{OID} \mathbf{J}$, if there exists a bijection on $\mathbf{dom} \cup \mathbf{obj}$ that maps **obj** to **obj**, is the identity on **dom**, and transforms **I** into **J**. To be precise, the term *object-oriented instance* should refer to an equivalence class under OID isomorphism of instances as defined earlier. However, it is usually more convenient to work with representatives of these equivalence classes, so we follow that convention here.

REMARK 21.2.9 In the model just described, a class encompasses two aspects:

1. at the schema level, the class definition (its type and method signatures); and
2. at the instance level, the class extension (the set of objects currently in the class).

It has been argued that one should not associate explicit class extensions with classes. To see the disadvantage of class extensions, consider object deletion. To be removed from the database, an object has to be deleted *explicitly* from its class extension. This is not convenient in some cases. For instance, suppose that the database contains a class *Polygon* and polygons are used only in figures. When a polygon is no longer used in any figure of the current database, it is no longer of interest and should be deleted. We would like this deletion to be implicit. (Otherwise the user of the database would have to search all possible places in which a reference to a polygon may occur to be able to delete a polygon.)

To capture this, some OODBs use an integrity constraint, which states that *every object should be accessible from some named value.*

This integrity constraint is enforced by an automatic deletion of all objects that become unreachable from the named values. In the polygon example, this approach would allow defining the class *Polygon*, thus specifying the structure and methods proper to polygons. However, the members of class *Polygon* would only be those polygons that are currently relevant. Relevance is determined by membership in (or accessibility from) the named values (e.g., *My-Figures, Your-Figures*) that refer to polygons. From a technical viewpoint, this involves techniques such as garbage collection.

In these OODBs, the set of objects in a class is not directly accessible. For this reason, the corresponding models are sometimes called models without class extension. Of course, it is always possible, given a schema, to compute the class extensions or to adapt object creation in a given class to maintain explicitly a named value containing that class extension. In these OODBs, the named values are also said to be roots of persistence, because the persistence of an object is dependent on its accessibility from these named values. ■

21.3 Languages for OODB Queries

This section briefly introduces several languages for querying OODBs. These queries are formulated against the database as a whole; unlike methods, they are not associated with specific classes. In the next section, we will consider languages intended to provide implementations for methods.

In describing the OODB query languages, we emphasize how OODB features are incorporated into them. The first language is an extension of the calculus for complex values, which incorporates such object-oriented components as OIDs, different notions of equality, and method calls. The second is an extension of the *while* language, initially introduced in Chapter 14. Of primary interest here is the introduction of techniques for creating new OIDs as part of a query. At this point we examine the notion of completeness for OODB access languages. We also briefly look at a language introducing a logic-based approach to object creation. Finally, we mention a practical language, O_2SQL. This is a variant of SQL for OODBs that provides elegant object-oriented features.

Although the languages discussed in this section do provide the ability to call methods and incorporate the results into the query processing and answer, we focus primarily on access to the extensional structural portion of the OODB. The intensional portion, provided by the methods, is considered in the following section. Also, we largely ignore the important issue of typing for queries and programs written in these languages. The issue of typing is considered, in the context of languages for methods, in the next section.

An Object-Oriented Calculus

The object-oriented calculus presented here is a straightforward generalization of the complex value calculus of Chapter 20, extended to incorporate objects, different notions of equality, and methods.

Let $(C, \sigma, \prec, M, G)$ be an OODB schema, and let us ignore the object-oriented features for a moment. Each name in G can be viewed as a complex value; it is straightforward to generalize the complex value calculus to operate on the values referred to by G. (The fact that in the complex value model all relations are sets whereas some names in G might refer to nonset values requires only a minor modification of the language.)

Let us now consider objects. OIDs may be viewed as elements of a specific sort. If viewed in isolation from their associated values, this suggests that the only primitive available for comparing OIDs is equality. Recall from the schema of Fig. 21.1 the names *Actors_I_like* and *Actors_you_like*. The query[6]

(21.1) $\exists x, y(x \in Actors\_I\_like \wedge y \in Actors\_you\_like \wedge x = y)$

asks whether there is an actor we both like. To obtain the names of such actors, we need to introduce dereferencing, a mechanism to obtain the value of an object. Dereferencing is denoted by $\uparrow$. The following query yields the names of actors we both like:

(21.2) $\{y \mid \exists x(x \in Actors\_I\_like \wedge x \in Actors\_you\_like \wedge x \uparrow .name = y)\}$

In the previous query, $x \uparrow$ denotes the value of x, in this case, a tuple with four fields. The dot notation (.) is used as before to obtain the value of specific fields.

In query (21.1), we tested two objects for equality, essentially testing whether they had the same OID. Although it does not increase the expressive power of the language, it is customary to introduce an alternative test for equality, called *value equality*. This tests whether the values of two objects are equal regardless of whether their OIDs are distinct. To illustrate, consider the three objects having *Actor_type*:

$$\begin{aligned}(&oid50, [name: \text{“Martin”}, citizenship: \text{“French”}, gender: \text{“male”},\\ &\qquad award: \{\,\}, acts\_in: \{oid33\}])\\ (&oid51, [name: \text{“Martin”}, citizenship: \text{“French”}, gender: \text{“male”},\\ &\qquad award: \{\,\}, acts\_in: \{oid33\}])\\ (&oid52, [name: \text{“Martin”}, citizenship: \text{“French”}, gender: \text{“male”},\\ &\qquad award: \{\,\}, acts\_in: \{oid34\}])\end{aligned}$$

Then *oid*50 and *oid*51 are value equal, whereas *oid*50 and *oid*52 are not. Yet another form of equality is *deep equality*. If *oid*33 and *oid*34 are value equal, then *oid*50 and *oid*52 are deep equal. Intuitively, two objects are deep equal if the (possibly infinite) trees obtained by recursively replacing each object by its value are equal. The infinite trees that we obtain are called the *expansions*. They present some regularity; they are regular trees (see Exercise 21.10).

The notion of deep equality highlights a major difference between value-based and object-based models. In a value-based model (such as the relational or complex value

[6] In this example, if *name* is a key for *Actor*, then one can easily obtain an equivalent query not using object equality; this may not be possible if there is no key for *Actor*.

models), the database can be thought of as a collection of (finite) trees. The connections between trees arise as a result of the contents of atomic fields. That is, they are implicit (e.g., the same string may appear twice). In the object-oriented world, a database instance can be thought of as graph. Paths in the database are more explicit. That is, one may view an $(oid, value)$ pair as a form of logical pointer and a path as a sequence of pointer dereferencing.

This graph-based perspective leads naturally to a navigational form of data access (e.g., using a sequence such as $o \uparrow .director \uparrow .citizenship$ to find the citizenship of the director of a given movie object o). This has led some to view object-oriented models as less declarative than value-based models such as the relational model. This is inaccurate, because declarativeness is more a property of access languages than models. Indeed, the calculus for OODBs described here illustrates that a highly declarative language can be developed for the OODB model.

We conclude the discussion of the object-oriented calculus by incorporating methods. For this discussion, it is irrelevant how the methods are specified or evaluated; this evaluation is external to the query. The query simply uses the method invocations as oracles. Method resolution uses dynamic binding. The value of an expression of the form $m(t_1, \ldots, t_n)$ under a given variable assignment ν is obtained by evaluating (externally) the implementation of m for the class of $\nu(t_1)$ on input $\nu(t_1, \ldots, t_n)$. In this context, it is assumed that m has no side-effects. Although not defined formally here, the following illustrates the incorporation of methods into the calculus:

(21.3) $\{y \mid \exists x(x \in Persons\_I\_like \wedge y = get\_name(x))\}$

If the set *Persons_I_like* contains Bergman and Liv Ullman, the answer would be

$$\{\text{“Ms. Ullman”}, \text{“Liv Ullman”}\}$$

The use of method names within the calculus raises a number of interesting typing and safety issues that will not be addressed here.

Object Creation and Completeness

Relational queries take relational instances as input and produce relational instances as output. The preceding calculus fails to provide the analogous capability because the output of a calculus query is a set of values or objects. Two features are needed for a query language to produce the full-fledged structural portion of an object-oriented instance: the ability to create OIDs, and the ability to populate a family of named values (rather than producing a single set).

We first introduce an extension of the *while* language of Chapter 14 that incorporates both of these capabilities. This language leads naturally to a discussion of completeness of OODB access languages. After this we mention a second approach to object creation that stems from the perspective of logic programming.

The extension of *while* introduced here is denoted $while_{obj}$. It will create new OIDs in a manner reminiscent of how the language $while_{new}$ of Chapter 18 invented new constants.

The language $while_{obj}$ incorporates object-oriented features such as dereferencing and method calls, as in the calculus. To illustrate, we present a $while_{obj}$ program that collects all actors reachable from an actor I like—Liv Ullman. In this query, *v_movies* and *v_directors* serve as variables, and *reachable* serves as a new name that will hold the output.

```
$reachable := \{x \mid x \in Actors\_I\_like \wedge x \uparrow .name = \text{“Liv Ullman”}\}$;
$v\_movies := \{\ \}$; $v\_directors := \{\ \}$;
while change do
  begin
  $reachable := reachable \cup \{x \mid \exists y(y \in v\_movies \wedge x \in y \uparrow .actors)\}$;
  $v\_directors := v\_directors$
    $\cup \{x \mid \exists y(y \in v\_movies \wedge x \in y \uparrow .director)\}$;
  $v\_movies := v\_movies$
    $\cup \{x \mid \exists y(y \in reachable \wedge x \in y \uparrow .acts\_in)\}$
    $\cup \{x \mid \exists y(y \in v\_directors \wedge x \in y \uparrow .directs)\}$;
  end;
```

We now introduce object creation. The operator **new** works as follows. It takes as input a set of values (or objects) and produces one new OID for each value in the set. As a simple example, suppose that we want to objectify the quadruples in the named value *Pariscope* of the schema of Fig. 21.1. This may be accomplished with the commands

```
add_class Pariscope_obj
  type tuple (theater : Theater, time : string, price : integer, movie : Movie);
Pariscope_obj := new(Pariscope)
```

Of course, the **new** operator can be used in conjunction with arbitrary expressions that yield a set of values, not just a named value.

The **new** operator used here is closely related to the *new* operator of the language $while_{new}$ of Chapter 18. Given that $while_{obj}$ has iteration and the ability to create new OIDs, it is natural to ask about the expressive power of this language. To set the stage, we introduce the following analogue of the notion of (computable) query, which mimics the one of Chapter 18. The definition focuses on the structural portion of the OODB model; methods are excluded from consideration.

Definition 21.3.1 Let $\mathbf{R}$ and $\mathbf{S}$ be two OODB schemas with no method signatures. A *determinate query* is a relation Q from $inst(\mathbf{R})$ to $inst(\mathbf{S})$ such that

(a) Q is computable;

(b) (Genericity) if $\langle \mathbf{I}, \mathbf{J}\rangle \in Q$ and ρ is a one-to-one mapping on constants, then $\langle \rho(\mathbf{I}), \rho(\mathbf{J})\rangle \in Q$;

(c) (Functionality) if $\langle \mathbf{I}, \mathbf{J}\rangle \in Q$, and $\langle \mathbf{I}, \mathbf{J}'\rangle \in Q$, then $\mathbf{J}$ and $\mathbf{J}'$ are OID isomorphic; and

(d) (Well defined) if $\langle \mathbf{I}, \mathbf{J}\rangle \in Q$ and $\langle \mathbf{I}', \mathbf{J}'\rangle$ is OID isomorphic to $\langle \mathbf{I}, \mathbf{J}\rangle$, then $\langle \mathbf{I}', \mathbf{J}'\rangle \in Q$.

A language is *determinate complete* (for OODBs) if it expresses exactly the determinate queries.

The essential difference between the preceding definition and the definition of determinate query in Chapter 18 is that here only OIDs can be created, not constants. Parts (c) and (d) of the definition ensure that a determinate query Q can be viewed as a *function* from OID equivalence classes of instances over $\mathbf{R}$ to OID equivalence classes of instances over $\mathbf{S}$. So OIDs serve two purposes here: (1) They are used to compute in the same way that invented values were used to break the polynomial space barrier; and (2) they are now essential components of the data structure and in particular of the result. With respect to (2), an important aspect is that we are not concerned with the actual value of the OIDs, which motivates the use of the equivalence relation. (Two results are viewed as identical if they are the same up to the renaming of the OIDs.)

Like $while_{new}$, $while_{obj}$ is not determinate complete. There is an elegant characterization of the determinate queries expressible in $while_{obj}$. This result, which we state next, uses a *local* characterization of input-output pairs of $while_{obj}$ programs. That characterization is in the spirit of the notion of BP-completeness, relating input-output pairs of relational calculus queries (see Exercise 16.11). For each input-output pair $\langle I, J\rangle$, the characterization of $while_{obj}$ queries requires a simple connection between the automorphism group of I and that of J. For an instance K, let $Aut(K)$ denote the set of automorphisms of K. For a pair of instances K, K', $Aut(\langle K, K'\rangle)$ denotes the bijections on $\mathbf{adom}(K \cup K')$ that are automorphisms of both K and K'.

Theorem 21.3.2 A determinate query q is expressible in $while_{obj}$ iff for each input-output pair $\langle I, J\rangle$ in q there exists a mapping h from $Aut(I)$ to $Aut(\langle I, J\rangle)$ such that for each $\tau, \mu \in Aut(I)$,

(i) τ and $h(\tau)$ coincide on I;

(ii) $h(\tau \circ \mu) = h(\tau) \circ h(\mu)$; and

(iii) $h(\mathbf{id}_I) = \mathbf{id}_{\langle I,J\rangle}$.

The "only if" part of the theorem is proven by an extension of the trace technique developed in the proof of Theorem 18.2.5 (Exercise 21.14). The "if" part is considerably more complex and is based on a group-theoretic argument.

A mapping h just shown is called an *extension homomorphism* from $Aut(I)$ to $Aut(\langle I, J\rangle)$. To see an example of the usefulness of this characterization, consider the query q in Fig. 21.3. Recall that q was shown as not expressible in the language $while_{new}$ by Theorem 18.2.5. The language $while_{obj}$ is more powerful than $while_{new}$, so in principle it may be able to express that query. However, we show that this is not the case, so $while_{obj}$ is not determinate complete.

Proposition 21.3.3 Query q (of Fig. 21.3) is not expressible in $while_{obj}$.

Proof Let $\langle I, J\rangle$ be the input-output pair of Fig. 21.3. The proof is by contradiction. Suppose there is a $while_{obj}$ query that produces J on input I. By Theorem 21.3.2, there is an extension homomorphism h from $Aut(I)$ to $Aut(\langle I, J\rangle)$. Let μ be the automorphism of I exchanging a and b. Note that $\mu^{-1} = \mu$, so $\mu \circ \mu = \mathbf{id}_I$. Consider $h(\mu)(\psi_0)$. Clearly, $h(\mu)(\psi_0) \in \{\psi_1, \psi_3\}$. Suppose $h(\mu)(\psi_0) = \psi_1$ (the other case is similar). Then clearly,

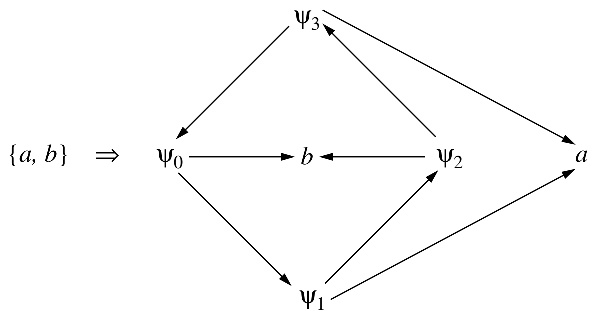

Figure 21.3: A query not expressible in *while*$_{obj}$

$h(\mu)(\psi_1) = \psi_2$. Consider now $h(\mu \circ \mu)(\psi_0)$. We have, on one hand,

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= (h(\mu) \circ h(\mu))(\psi_0) \\ &= h(\mu)(\psi_1) \\ &= \psi_2 \end{aligned}$$

and on the other hand

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= h(\mathbf{id}_I)(\psi_0) \\ &= \mathbf{id}_{\langle I,J\rangle}(\psi_0) \\ &= \psi_0, \end{aligned}$$

which is a contradiction because $\psi_0 \neq \psi_2$. So q is not expressible in *while*$_{obj}$. ■

It is possible to obtain a language expressing all determinate queries by adding to *while*$_{obj}$ a *choose* operator that allows the selection (nondeterministically but in a determinate manner) of one object out of a set of objects that are isomorphic (see Exercise 18.14). However, this is a highly complex construct because it requires the ability to check for isomorphism of graphs. The search for simpler, local constructs that yield a determinate-complete language is an active area of research.

A Logic-Based Approach to Object Creation

We now briefly introduce an alternative approach for creating OIDs that stems from the perspective of datalog and logic programming. Suppose that a new OID is to be created for each pair $\langle t, m\rangle$, where movie m is playing at theater t according to the current value of *Pariscope*. Consider the following dataloglike rule:

1. $\mathit{create\_tm\_object}(x, t, m) \leftarrow \mathit{Pariscope}(t, s, m)$

Note that x occurs in the rule head but not in the body, so the rule is not safe. Intuitively, we would like to attach semantics to this rule so that a new OID is associated to x for each distinct pair of (t, m) values. Using the symbol $\exists!$ to mean "exists a unique," the following versions of (1) intuitively captures the semantics.

2. $\forall t \forall m \exists! x \forall s[\mathit{create\_tm\_object}(x, t, m) \leftarrow \mathit{Pariscope}(t, s, m)]$
3. $\forall t \forall m \exists! x[\mathit{create\_tm\_object}(x, t, m) \leftarrow \exists s(\mathit{Pariscope}(t, s, m))]$

This suggests that Skolem functions might be used. Specifically, let f_{tm} be a function symbol associated with the predicate *create_tm_object*. We rewrite (2) as

$$\forall t \forall m \forall s[\mathit{create\_tm\_object}(f_{tm}(t, m), t, m) \leftarrow \mathit{Pariscope}(t, s, m)]$$

or, leaving off the universal quantifiers as traditional in datalog,

4. $\mathit{create\_tm\_object}(f_{tm}(t, m), t, m) \leftarrow \mathit{Pariscope}(t, s, m)$

Under this approach, the Skolem terms resulting from rule (4) are to be interpreted as new, distinct OIDs. Under some formulations of the approach, syntactic objects such as $f_{tm}(\mathit{oid}7, \mathit{oid}22)$ (where $\mathit{oid}7$ is the OID of some theater and $\mathit{oid}22$ the OID of some movie) serve explicitly as OIDs. Under other formulations, such syntactic objects are viewed as placeholders during an intermediate stage of query evaluation and are (nondeterministically) replaced by distinct new OIDs in the final stage of query evaluation (see Exercise 21.13).

The latter approach to OID creation, incorporated into complex value datalog extended to include also OID dereferencing, yields a language equivalent to while_{obj}. As with while_{obj}, this language is not determinate complete.

A Practical Language for OODBs

We briefly illustrate some object-oriented features of the language O$_2$SQL, which was introduced in Section 20.8. Several examples are presented there, that show how O$_2$SQL can be used to access and construct deeply nested complex values. We now indicate how the use of objects and methods is incorporated into the language. It is interesting to note that methods and nested complex values are elegantly combined in this language, which has the appearance of SQL but is essentially based on the functional programming paradigm.

For this example, we again assume the complex value *Films* of Fig. 20.2, but we assume that *Age* is a method defined for the class *Person* (and thus for *Director*).

```
select  tuple (f.Director, f.Director.Age)
from    f in Films
where f.Director not in flatten select m.Actors
                                from  g in Films,
                                      m in g.Movies
                                where g.Director = "Hitchcock"
```

(Recall that here the inner *select-from-where* clause returns a set of sets of actors. The keyword **flatten** has the effect of forming the union of these sets to yield a set of actors.)

21.4 Languages for Methods

So far, we have used an abstraction of methods (their signature) and ignored their implementations. In this section, we present two abstract programming languages for specifying method implementations. Method implementations will be included in the specification of methods in OODB schemas. In studying these languages, we emphasize two important issues: type safety and expressive power. This focus largely motivates our choice of languages and the particular abstractions considered.

The first language is an imperative programming language. The second, method schemas, is representative of a functional style of database access. In the first language, we will gather a number of features present in practical object-oriented database languages (e.g., side-effect, iteration, conditionals). We will see that with these features, we get (as could be expected) completeness, and we pay the obvious price for it: the undecidability of many questions, such as type safety. With method schemas, we focus on the essence of inheritance and methods. We voluntarily consider a limited language. We see that the undecidability of type safety is a consequence of recursion in method calls. (We obtain decidability in the restricted case of monadic methods.) With respect to expressiveness, we present a surprising characterization of QPTIME in terms of a simple language with methods.

For both languages, we study type safety and expressive power. We begin by discussing briefly the meaning of these notions in our context, and then we present the two languages and the results.

An OODB schema **S** (with method implementations assigned to signatures) is type safe if for each instance **I** of **S** and each syntactically correct method call on **I**, the execution of this method does not result in a runtime type error (an illegal method call). When the imperative programming language is used in method implementations, type safety is undecidable. (It is possible, however, to obtain decidable sufficient conditions for type safety.) For method schemas, type safety remains undecidable. Surprisingly, type safety is decidable for monadic method schemas.

To evaluate the expressive power of OODB schemas using a particular language for method implementation, a common approach is to simulate relational queries and then ask what family of relational queries can be simulated. If OID creation is permitted, then all computable relational queries can be simulated using the imperative language. The expressive power of imperative methods without OID creation depends on the complex types permitted in OODB schemas. We also present a result for the expressive power of method schemas, showing that the family of method schemas using an ordered domain of atomic elements expresses exactly QPTIME.

A Model with Imperative Methods

To consider the issue of type safety in a general context, we present the *imperative (OODB) model*, which incorporates imperative method implementations. This model simplifies the OODB model presented earlier by assuming that the type of each class is a tuple of values and OIDs. However, a schema in this model will include an assignment of implementations to method signatures.

The syntax for method implementations is

par: $u_1, \ldots, u_n$;
var: $x_1, \ldots, x_l$;
body: $s_1; \ldots; s_q$;
return x_1

where the u_i's are parameters ($n \geq 1$), the x_j's are internal variables ($l \geq 1$), and for each $p \in [1, q]$, s_p is a statement of one of the following forms (where w, y, z range over parameters and internal variables):

Basic operations

(i) $w :=$ *self*.
(ii) $w :=$ *self.a* for some field name a.
(iii) $w := y$.
(iv) $w := m(y, \ldots, z)$, for some method name m.
(v) *self.a* $:= w$, for some field name a.

Class operations

(vi) $w :=$ **new**(c), where c is a class.
(vii) **delete**(c, w), where c is a class.
(viii) **for each** w **in** c **do** $s'_1; \ldots; s'_t$ **end**, where c is a class and $s'_1, \ldots, s'_t$ are statements having forms from this list.

Conditional

(ix) **if** $y\theta z$ **then** s, where θ is $=$ or $\neq$ and s is a statement having a form in this list except for the conditional.

It is assumed that all internal variables are initialized before used to some default value depending on their type. The intended semantics for the forms other than (viii) should be clear. (Here *clear* does not mean "easy to implement." In particular, object deletion is complex because all references to this object have to be deleted.) The looping construct executes for each element of the extension (not disjoint extension) of class c. The execution of the loop is viewed as nondeterministic, in the sense that the particular ordering used for the elements of c is not guaranteed by the implementation. In general, we focus on OODB schemas in which different orders of execution of the loops yield OID-equivalent results (note, however, that this property is undecidable, so it must be ensured by the programmer).

An imperative *schema* is a 6-tuple $\mathbf{S} = (C, \sigma, \prec, M, G, \mu)$, where $(C, \sigma, \prec, M, G)$ is a schema as before; where the range of σ is tuples of atomic and class types; and where μ is an assignment of implementations to signatures. The notion of *instance* for this model is defined in the natural fashion.

It is straightforward to develop operational semantics for this model, where the execution of a given method call might be *successful*, *nonterminating*, or *aborted* (as the result of a runtime type error) (Exercise 21.15a).

Type Safety in the Imperative Model There are two ways that a runtime type error can arise: (1) if the type of the result of an execution of method m does not lie within the type specified by the relevant method signature of m; or (2) if a method is called on a tuple of parameters that does not satisfy the domain part of the appropriate signature of m. We assume that the range of all method signatures is **any**, and thus we focus on case (2).

A schema $\mathbf{S}$ is *type safe* if for each instance over $\mathbf{S}$ and each $m(o, v_1, \dots, v_n)$ method call that satisfies the signature of m associated with the class of o, execution of this call is either successful or nonterminating.

Given a Turing machine M, it is easy to develop a schema S in this model that can simulate the operation of M on a suitable encoding of an input tape (Exercise 21.15c). This shows that such schemas are computationally powerful and implies the usual undecidability results. With regard to type safety, it is easy to verify the following (Exercise 21.16):

Proposition 21.4.1 It is undecidable, given an imperative schema $\mathbf{S}$, whether $\mathbf{S}$ is type safe. This remains true, even if in method implementations conditional statements and the **new** operator are prohibited and all methods are monadic (i.e., have only one argument).

A similar argument can be used to show that it is undecidable whether a given method terminates on all inputs. Finally, a method m' on class c' is *reachable* from method m on class c in OODB schema $\mathbf{S}$ if there is some instance $\mathbf{I}$ of $\mathbf{S}$ and some tuple $o, v_1, \dots$ with o in c such that the execution of $m(o, v_1, \dots)$ leads to a call of m' on some object in c'. Reachability is also undecidable for imperative schemas.

Expressive Power of the Imperative Model

As discussed earlier, we measure the expressive power of OODB schemas in terms of the relational queries they can simulate. A relational schema $\mathbf{R} = \{R_1, \dots, R_n\}$ is *simulated by* an OODB schema $\mathbf{S}$ of this model if there are leaf classes $c_1, \dots, c_n$ in $\mathbf{S}$, where the number of attributes of c_i is the arity of R_i for $i \in [1, n]$ and where the type of each of these attributes is atomic. We focus on instances in which no null values appear for such attributes. Let $\mathbf{R}$ be a relational schema and $\mathbf{S}$ be an OODB schema that simulates $\mathbf{R}$. An instance $\mathbf{I}$ of $\mathbf{R}$ is *simulated* by instance $\mathbf{J}$ of $\mathbf{S}$ if for each tuple $\vec{v} \in \mathbf{I}(R_i)$ there is exactly one object o in the extension of c_i such that the value associated with o is $\vec{v}$ and all other classes of $\mathbf{S}$ are empty. Following this spirit, it is straightforward to define what it means for a method call in schema $\mathbf{S}$ to simulate a relational query from $\mathbf{R}$ to relation schema R.

We consider only schema $\mathbf{S}$ for which different orders of evaluation of the looping construct yield the same final result (i.e., generic mappings). We now have the following (see Exercise 21.20):

Theorem 21.4.2 The family of generic queries corresponding to imperative schemas coincides with the family of all relational queries.

The preceding result relies on the presence of the **new** operator. It is natural to ask about the expressive power of imperative schemas that do not support **new**. As discussed in Exercise 21.21, the expressive power depends on the complex types permitted for objects.

Note also that imperative schemas can express all determinate queries. This uses the nondeterminism of the **for each** construct. Naturally, nondeterministic queries that are not determinate can also be expressed.

Method Schemas

We now present an abstract model for side-effect-free methods, called method schemas. In this model, we focus almost exclusively on methods and their implementations. Two kinds of methods are distinguished: base and composite. The base methods do not have implementations: Their semantics is specified explicitly at the instance level. The implementations of composite methods consist of a composition of other methods.

We now introduce method schemas. In the next definition, we make the simplifying assumption that there are no named values (only class names) in database schemas. In fact, data is only stored in base methods. In the following, $\sigma_{[\,]}$ denotes the type assignment $\sigma_{[\,]}(c) = [\,]$ for every class c. Because the type assignment provides no information in method schemas (it is always $\sigma_{[\,]}$), this assignment is not explicitly specified in the schemas.

Definition 21.4.3 A *method schema* is a 5-tuple $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$, where $(C, \sigma_{[\,]}, \prec)$ is a well-formed class hierarchy, $M_{base} \cup M_{comp}$ is a well-formed set of method signatures for $(C, \sigma_{[\,]}, \prec)$, and

- no method name occurs in both M_{base} and M_{comp};
- each method signature in M_{comp} is of the form $m : c_1, \ldots, c_n \rightarrow$ **any** (method signatures for M_{base} are unrestricted, i.e., can have any class as range);
- μ is an assignment of implementations to the method signatures of M_{comp}, as follows: For a signature $m : c_1, \ldots, c_n \rightarrow$ **any** in M_{comp}, $\mu(m : c_1, \ldots, c_n \rightarrow$ **any**$)$ is a term obtained by composing methods in M_{base} and M_{comp}.

An example of an implementation for a method $m : c_1, c_2 \rightarrow$ **any** is

$$m(x, y) \equiv m_1(m_2(x), m_1(x, y)).$$

The semantics of methods is defined in the obvious way. For instance, to compute $m(o, o')$, one computes first $o_1 = m_2(o)$ and then $o_2 = m_1(o, o')$; the result is $m_1(o_1, o_2)$. The range of composite methods is left unspecified (it is **any**) because it is determined by the domain and the method implementation as a composition of methods. Because the range of composite methods is always **any**, we will sometimes only specify their domain.

Let $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$ be a method schema. An *instance* of $\mathbf{S}$ is a pair $\mathbf{I} = (\pi, \nu)$, where π is an OID assignment for $(C, \prec)$ and where ν assigns a semantics to the base methods. Note the difference from the imperative schemas of the previous section, where π together with the method implementations was sufficient to determine the semantics of methods. In contrast, the semantics of the base methods must be specified in instances of method schemas.

Inheritance of method implementations for method schemas is defined slightly differently from that for the OODB model given earlier. Specifically, given an n-ary method m and invocation $m(o_1, \ldots, o_n)$, where o_i is in disjoint class c_i for $i \in [1, n]$, the implementation for m is inherited from the implementation of signature $m : c'_1, \ldots, c'_n \rightarrow c'$, where this is the unique signature that is pointwise least above $c_1, \ldots, c_n$. [Otherwise m is undefined on input $(o_1, \ldots, o_n)$.]

An important special case is when methods take just *one* argument. Method schemas using only such methods are called *monadic*. To emphasize the difference, unrestricted method schemas are sometimes called *polyadic*.

Example 21.4.4 Consider the following monadic method schema. The classes in the schema are

$$\begin{aligned} &\textit{class } c \\ &\textit{class } c' \prec c \end{aligned}$$

The base method signatures are

$$\begin{aligned} &\textit{method } m_1 : c \rightarrow c' \\ &\textit{method } m_2 : c \rightarrow c \\ &\textit{method } m_2 : c' \rightarrow c' \\ &\textit{method } m_3 : c' \rightarrow c \end{aligned}$$

The composite method definitions are

$$\begin{aligned} &\textit{method } m : c = m_2(m_2(m_1(x))) \\ &\textit{method } m' : c = m_3(m'(m_2(x))) \\ &\textit{method } m' : c' = m_1(x) \end{aligned}$$

Note that m' is recursive and that calls to m' on elements in c' break the recursion.

Type Safety for Method Schemas As before, a method schema **S** is *type safe* if for each instance **I** of **S** no method call on **I** leads to a runtime type error.

The following example demonstrates that the schema of Example 21.4.4 is not type safe. Note how the interpretation ν for base methods can be viewed as an assignment of values for objects.

Example 21.4.5 Recall the method schema of Example 21.4.4. An instance of this is $\mathbf{I} = (\pi, \nu)$, where[7]

$$\begin{aligned} \pi(c) &= \{p, q\} \\ \pi(c') &= \{r\} \end{aligned}$$

[7] We write $\nu(m_1)(p)$ rather than $\nu(m_1, c)(p)$ to simplify the presentation.

and

$$\begin{array}{lll} \nu(m_1)(p) = r & \nu(m_2)(p) = q & \\ \nu(m_1)(q)l = \bot & \nu(m_2)(q) = r & \nu(m_3)(r) = p. \\ \nu(m_1)(r) = r & \nu(m_2)(r) = r & \end{array}$$

Consider the execution of $m(p)$. This calls for the computation of $m_2(m_2(m_1(p))) = m_2(m_2(r)) = r$. Thus the execution is successful. On the other hand, $m'(p)$ leads to a runtime type error: $m'(p) = m_3(m'(m_2(p))) = m_3(m'(q)) = m_3(m_3(m'(m_2(q)))) = m_3(m_3(m'(r))) = m_3(m_3(m_1(r))) = m_3(m_3(r)) = m_3(p)$, which is undefined and raises a runtime type error. Thus the schema is not type safe.

It turns out that type safety of method schemas permitting polyadic methods is undecidable (Exercise 21.19). Interestingly, type safety is decidable for monadic method schemas. We now sketch the proof of this result.

Theorem 21.4.6 It is decidable in polynomial time whether a monadic method schema is type safe.

Crux Let $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$ be a monadic method schema. We construct a context-free grammar (see Chapter 2) that captures possible executions of a method call over all instances of **S**. The grammar is $G_{\mathbf{S}} = (V_n, V_t, A, P)$, where the set V_t of terminals is the set of base method names (denoted N_{base}) along with the symbols $\{\langle error\rangle, \langle ignore\rangle\}$, and the set V_n of nonterminals includes start symbol A and

$$\{[c, m, c'] \mid c, c' \text{ are classes, and } m \text{ is a method name}\}$$

The set P of production rules includes

(i) $A \to [c, m, c']$, if m is a composite method name and it is defined at c or a superclass of c.

(ii) $[c, m, c'] \to \langle error\rangle$, if m is not defined at c or a superclass of c.

(iii) $[c, m, c'] \to m$, if m is a base method name, the resolution of m for c is $m : c_1 \to c_2$, and $c' \prec c_2$. (Note that $c' = c_2$ is just a particular case.)

(iv) $[c, m, c] \to \epsilon$, if m is a composite method name and the resolution of m for c is the identity mapping.

(v) $[c, m, c_n] \to [c, m_1, c_1][c_1, m_2, c_2] \ldots [c_{n-1}, m_n, c_n]$, if m is a composite method, m on c resolves to a method with implementation $m_n(m_{n-1}(\ldots(m_2(m_1(x)))\ldots))$, and $c_1, \ldots, c_n$ are arbitrary classes.

(vi) $[c, m, c'] \to \langle ignore\rangle$, for all classes c, c' and method names m.

Given a successful execution of a method call $m(o)$, it is easy to construct a word in $L(G_{\mathbf{S}})$ of the form $m_1 \ldots m_n$, where the m_i's list the sequence of base methods called during the execution. On the other hand, if the execution of $m(o)$ leads to a runtime error, a word of the form $m_1 \ldots m_i\langle error\rangle \ldots$ can be formed. The terminal $\langle ignore\rangle$ can be used in cases where a nonterminal $[c, m, c']$ arises, such that m is a base method name and c' is outside the range of m for c. The productions of type (vi) are permitted for all nonterminals $[c, m, c']$, although they are needed only for some of them.

It can be shown that **S** is type safe iff

$$L(G_{\mathbf{S}}) \cap N^*_{base} \langle error \rangle V^*_t = \emptyset.$$

Because it can be tested if the intersection of a context-free language with a regular language is empty, the preceding provides an algorithm for checking type safety. However, a modification of the grammar $G_{\mathbf{S}}$ is needed to obtain the polynomial time test (see Exercise 21.18). ■

Expressive Power of Method Schemas We now argue that method schemas (with order) simulate precisely the relational queries in QPTIME. The object-oriented features are not central here: The same result can be shown for functional data models without such features.

As for imperative schemas, we show that method schemas can simulate relational queries. The encoding of these queries assumes an ordered domain, as is traditional in the world of functional programming.

A relational database is encoded as follows:

(a) a class **elem** contains objects representing the elements of the domain, and it has **zero** as a subclass containing a unique element, say 0;

(b) a function *pred*, which is included as a base method,[8] provides the predecessor function over **elem** ∪ **zero** [$pred(0)$ is, for instance, 0]; a base method 0 returns the least element and another base method N the largest object in **elem**;

(c) to have the Booleans, we think of 0 as the value *false* and all objects in **elem** as representations of *true*;

(d) an n-ary relation R is represented by an n-ary base method m_R of signature m_R : **elem**, ..., **elem** → **elem**, the characteristic function of R. [For a tuple t, $m_R(t)$ is *true* iff t is in R.]

Next we represent queries by composite methods. A query q is computed by method m_q if $m_q(t)$ is true (not in **zero**) iff t is in the answer to query q.

The following illustrates how to compute with this simple language.

Example 21.4.7 Consider relation R with $R = \{R(1, 1), R(1, 2)\}$. The class **zero** is populated with the object 0 and the class **elem** with 1, 2. The base method *pred* is defined by $pred(2) = 1, pred(0) = pred(1) = 0$. The base method m_R is defined by $m_R(1, 1) = m_R(1, 2) = 1$ and $m_R(x, y) = 0$ otherwise.

---

[8] The function *pred* is a functional analog of the relation *succ*, which we have assumed is available in every ordered database (a successor function could also have been used).

Recall that each object in class **elem** is viewed as *true* and object 0 as *false*. We can code the Boolean function *and* as follows:

$$\begin{array}{lll} \text{for } x, y \text{ in} & \textbf{zero}, \textbf{zero} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{zero} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{zero}, \textbf{elem} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{elem} & and(x, y) \equiv N. \end{array}$$

The other standard Boolean functions can be coded similarly. We can code the intersection between two binary relations R and S with $and(m_R(x, y), m_S(x, y))$. As a last example, the projection of a binary relation R over the first coordinate can be coded by a method $\pi_{R,1}$ defined by

$$\pi_{R,1} \equiv m(x, N),$$

where m is given by

$$\begin{array}{lll} \text{for } x, y \text{ in} & \textbf{elem}, \textbf{zero} & m(x, y) \equiv m_R(x, y) \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{elem} & m(x, y) \equiv or(m_R(x, y), m(x, pred(y))). \end{array}$$

We now state the following:

Theorem 21.4.8 Method schemas over ordered databases express exactly QPTIME.

Crux As indicated in the preceding example, we can construct composite methods for the Boolean operations *and*, *or*, and *not*. For each k, we can also construct k k-ary functions $pred_k^i$ for $i \in [1, k]$ that compute for each k tuple u the k components of the predecessor (in lexicographical ordering) of u. Indeed, we can simulate an arbitrary relational operation and more generally an arbitrary inflationary fixpoint. To see this, consider the transitive closure query. It is computed with a method tc defined (informally) as follows. Intuitively, a method $tc(x, y)$ asks, "Is $\langle x, y\rangle$ in the transitive closure?" Execution of $tc(x, y)$ first calls a method $m_1(x, y, N)$, whose intuitive meaning is "Is there a path of length N from x to y?" This will be computed by asking whether there is a path of length $N - 1$ (a recursive call to m_1), etc. This can be generalized to a construction that simulates an arbitrary inflationary fixpoint query. Because the underlying domain is ordered, we have captured all QPTIME queries. The converse follows from the fact that there are only polynomially many possible method calls in the context of a given instance, and each method call in this model can be answered in QPTIME. Moreover, loops in method calls can be detected in polynomial time; calls giving rise to loops are assumed to output some designated special value. (See Exercise 21.25.) ■

We have presented an object-oriented approach in the applicative programming style. There exists another important family of functional languages based on typed λ calculi. It is possible to consider database languages in this family as well. These calculi present additional advantages, such as being able to treat functions as objects and to use higher-order functions (i.e., functions whose arguments are functions).

21.5 Further Issues for OODBs

As mentioned at the beginning of this chapter, the area of OODB is relatively young and active. Much research is needed to understand OODBs as well as we understand relational databases. A difficulty (and richness) is that there is still no well-accepted model. We conclude this chapter with a brief look at some current research issues for OODBs. These fall into two broad categories: advanced modeling features and dynamic aspects.

Advanced Modeling Features

This is not an exhaustive list of new features but a sample of some that are being studied:

Views: Views are intended to increase the flexibility of database systems, and it is natural to extend the notion of relational view to the OODB framework. However, unlike relational views, OODB views might redefine the behavior of objects in addition to restructuring their associated types. There are also significant issues raised by the presence of OIDs. For example, to maintain incrementally a materialized view with created OIDs, the linkage between the base data and the created OIDs must be maintained. Furthermore, if the view is virtual, then how should virtual OIDs be specified and manipulated?

Object roles: The same entity may be involved in several roles. For instance, a director may also be an actor. It is costly, if not infeasible, to forecast all cases in which this may happen. Although not as important in object-oriented programming, in OODBs it would be useful to permit the same object to live in several classes (a departure from the disjoint OID assignment from which we started) and at least conceptually maintain distinct repositories, one for each role. This feature is present in some semantic data models; in the object-oriented context, it raises a number of interesting typing issues.

Schema design: Schema design techniques (e.g., based on dependencies and normal forms) have emerged for the relational model (see Chapter 11). Although the richer model in the OODB provides greater flexibility in selecting a schema, there is a concomitant need for richer tools to facilitate schema design. The scope of schema design is enlarged in the OODB context because of the interaction of methods within a schema and application software for the schema.

Querying the schema: In many cases, information may be hidden in an OODB schema. Suppose, for example, that movies were assigned categories such as “drama,” “western,” “suspense,” etc. In the relational model, this information would typically be represented using a new column in the *Movies* relation. A query such as “list all categories of movie that Bergman directed” is easily answered. In an OODB, the category information might be represented using different subclasses of the *Movie* class. Answering this query now requires the ability of the query language to return class names, a feature not present in most current systems.

Classification: A related problem concerns how, given an OODB schema, to classify new data for this schema. This may arise when constructing a view, when merging two databases, or when transforming a relational database into an OODB one by objectifying tuples. The issue of classification, also called taxonomic reasoning, has a long history in the field of knowledge representation in artificial intelligence, and some research in this direction has been performed for semantic and object-oriented databases.

Incorporating deductive capabilities: The logic-programming paradigm has offered a tremendous enhancement of the relational model by providing an elegant and (in many cases) intuitively appealing framework for expressing a broad family of queries. For the past several years, researchers have been developing hybrids of the logic-programming and object-oriented paradigms. Although it is very different in some ways (because the OO paradigm has fundamentally imperative aspects), the perspective of logic programming provides alternative approaches to data access and object creation.

Abstract data types: As mentioned earlier, OODB systems come equipped with several constructors, such as set, list, or bag. It is also interesting to be able to extend the language and the system with application-specific data types. This involves language and typing issues, such as how to gracefully incorporate access mechanisms for the new types into an existing language. It also involves system issues, such as how to introduce appropriate indexing techniques for the new type.

Dynamic Issues

The semantics of updates in relational systems is simple: Perform the update if the result complies with the dependencies of the schema. In an OODB, the issue is somewhat trickier. For instance, can we allow the deletion of an object if this object is referred to somewhere in the database (the dangling reference problem)? This is prohibited in some systems, whereas other systems will accept the deletion and just mark the object as dead. Semantically, this results in viewing all references to this object as nil.

Another issue is object migration. It is easy to modify the value of an object. But changing the status of an object is more complicated. For example, a person in the database may act in a movie and overnight be turned into an actor. In object-oriented programming languages, objects are often not allowed to change classes. Although such limitations also exist in most present OODBs, object migration is an important feature that is needed in many database applications. One approach, followed by some semantic data models, is to permit objects to be associated with multiple classes or roles and also permit them to migrate to different classes over time. This raises fundamental issues with regard to typing. For example, how do we treat a reference to the manager of a department (that should be of type *Employee*) when he or she leaves the company and is turned into a "normal" person?

Finally, as with the relational model, we need to consider evolution of the schema itself. The OODB context is much richer than the relational, because there are many more kinds of changes to consider: the class hierarchy, the type of a class, additions or deletions of methods, etc.

Bibliographic Notes

Collections of papers on object-oriented databases can be found in [BDK92, KL89, ZM90]. The main characteristics of object-oriented database systems are described in [ABD$^+$89]. An influential discussion of some foundational issues around the OODB paradigm is [Bee90]. An important survey of subtyping and inheritance from the perspective of programming languages, including the notion of domain-inclusion semantics, is [CW85].

Object-oriented databases are, of course, closely related to object-oriented programming languages. The first of these is Smalltalk [GR83], and C++ [Str91] is fast becoming the most widely used object-oriented programming language. Several commercial OODBs are essentially persistent versions of C++. Several object-oriented extensions of Lisp have been proposed; the article [B$^+$86] introduces a rich extension called CommonLoops and surveys several others.

There have been a number of approaches to provide a formal foundation [AK89, Bee90, HTY89, KLW93] for OODBs. We can also cite as precursors attempts to formalize semantic data models [AH87] and object-based models [KV84, HY84]. Recent graph-oriented models, although they do not stress object orientation, are similar in spirit (e.g., [GPG90]).

The generic OODB model used here is directly inspired from the IQL model [AK89] and that of O_2 [BDK92, LRV88]. The model and results on imperative method implementations are inspired by [HTY89, HS89a]. A similar model of imperative method implementation, which avoids nondeterminism and introduces a parallel execution model, is developed in [DV91]. Method schemas and Theorem 21.4.6 are from [AKRW92]; the functional perspective and Theorem 21.4.8 are from [HKR93].

OIDs have been part of many data models. For example, they are called *surrogates* in [Cod79], *l-values* in [KV93a], or *object identifiers* in [AH87]. The notion of object and the various forms of equalities among objects form the topic of [KC86]. Type inheritance and multiple inheritance are studied in [CW85, Car88].

Since [KV84], various languages for models with objects have been proposed in the various paradigms: calculus, algebra, rule based, or functional. Besides standard features found in database languages without objects, the new primitives are centered around object creation. Language-theoretic issues related to object creation were first considered in the context of IQL [AK89]. Object creation is an essential component of IQL and is the main reason for the completeness of the language. The need for a primitive in the style of copy elimination to obtain determinate completeness was first noticed in [AK89]. The IQL language is rule based with an inflationary fixpoint semantics in the style of datalog$^\neg$ of Chapter 14.

The logic-based perspective on object creation based on Skolem was first informally discussed in [Mai86] and refined variously in [CW89a, HY90, KLW93, KW89]. In particular, F-logic [KLW93] considers a different approach to inheritance. In our framework, the classification of objects is explicit; in particular, when an object is created, it is within an explicit class. In [KLW93], data organization is also specified by rules and thus may depend on the properties of the objects involved. For instance, reasoning about the hierarchy becomes part of the program.

Algebraic and imperative approaches to object creation are developed in [Day89].

Since then, object creation has been the center of much interesting research (e.g., [DV93, HS89b, HY92, VandBG92, VandBGAG92, VandB93]). The characterization of queries expressible in $while_{obj}$ (Theorem 21.3.2) is from [VandBG92]; this extends a previous result from [AP92]. The proof of Proposition 21.3.3 is also from [VandBGAG92]. In [VandBGAG92, VandB93], it is argued that the notion of determinate query may not be the most appropriate one for the object-based context, and alternative notions, such as semideterministic queries, are discussed. A tractable construct yielding a determinate-complete language is exhibited in [DV93]. However, the construct proposed there is global in nature and is involved. The search for simpler and more natural local constructs continues.

As mentioned earlier, the OODB calculus and algebra presented here are mostly variations of languages for non object-based models and, in particular, of the languages for complex values of Chapter 20. There have been several proposals of SQL extensions. In particular, as indicated in Section 21.3, O_2SQL [BCD89] retains the flavor of SQL but incorporates object orientation by adopting an elegant functional programming style. This approach has been advanced as a standard in [Cat94].

Functional approaches to databases have been considered rather early but attracted only modest interest in the past [BFN82, Shi81]. The functional approach has become more popular recently, both because of the success of object-oriented databases and due to recent results of complex objects and types emphasizing the functional models [BTBN92, BTBW92]. The use of a typed functional language similar to λ calculus as a formalism to express queries is adapted from [HKM93]. Characterizations of QPTIME in functional terms are from [HKM93, LM93]. The work in [AKRW92, HKM93, HKR93] provides interesting bridges between (object-oriented) databases and well-developed themes in computer science: applicative program schemas [Cou90, Gre75] and typed λ calculi [Chu41, Bar84, Bar63].

This chapter presented both imperative and functional perspectives on OODB methods. A different approach (based on rules and datalog with negation) has been used in [ALUW93] to provide semantics to a number of variations of schemas with methods. The connection between methods and rule-based languages is also considered in [DV91].

Views for OODBs are considered in [AB91, Day89, HY90, KKS92, KLW93]. The merging of OODBs is considered in [WHW90]. Incremental maintenance of materialized object-oriented views is considered in [Cha94]. The notion of object roles, or sharing objects between classes, is found in some semantic data models [AH87, HK87] and in recent research on OODBs [ABGO93, RS91]. A query language that incorporates access to an OODB schema is presented in [KKS92]. Classification has been central to the field of knowledge representation in artificial intelligence, based on the central notion of taxonomic reasoning (e.g., see [BGL85, MB92], which stem from the KL-ONE framework of [BS85]); this approach has been carried to the context of OODBs in, for example, [BB92, BS93, BBMR89, DD89]. Deductive object-oriented database is the topic of a conference (namely, the *Intl. Conf. on Deductive and Object-Oriented Databases*). Properties of object migration between classes in a hierarchy are studied in [DS91, SZ89, Su92].

Exercises

Exercise 21.1 Construct an instance for the schema of Fig. 21.1 that corresponds to the **CINEMA** instance of Chapter 3.

Exercise 21.2 Suppose that the class *Actor_Director* were removed from the schema of Fig. 21.1. Verify that in this case there is no OID assignment for the schema such that there is an actor who is also a director.

Exercise 21.3 Design an OODB schema for a bibliography database with articles, book chapters, etc. Use inheritance where possible.

Exercise 21.4 Exhibit a class hierarchy that is not well formed.

Exercise 21.5 Add methods to the schema of Fig. 21.1 so that the resulting family of methods violates rules *unambiguous* and *covariance*.

Exercise 21.6 Show that testing whether $\mathbf{I} \equiv_{OID} \mathbf{J}$ is in NP and at least at hard as the graph isomorphism problem (i.e., testing whether two graphs are isomorphic).

Exercise 21.7 Give an algorithm for testing value equality. What is the data complexity of your algorithm?

Exercise 21.8 In this exercise, we consider various forms of equality. Value equality as discussed in the text is denoted $=_1$. Two objects o, o' are 2-value equal, denoted $o =_2 o'$, if replacing each object in $\nu(o)$ and $\nu(o')$ by its value yields values that are equal. The relations $=_i$ for each i are defined similarly. Show that for each i, $=_{i+1}$ refines $=_i$. Let n be a positive integer. Give a schema and an instance over this schema such that for each i in $[1, n]$, $=_i$ and $=_{i+1}$ are different.

Exercise 21.9 Design a database schema to represent information about persons, including males and females with names and husbands and wives. Exhibit a cyclic instance of the schema and an object o that has an infinite expansion. Describe the infinite tree representing the expansion of o.

★ **Exercise 21.10** Consider a database instance **I** over a schema **S**. For each o in **I**, let $expand(o)$ be the (possibly infinite) tree obtained by replacing each object by its value recursively. Show that $expand(o)$ is a *regular* tree (i.e., that it has a finite number of distinct subtrees). Derive from this observation an algorithm for testing deep equality of objects.

Exercise 21.11 In this exercise, we consider the schema **S** with a single class c that has type $\sigma(c) = [A : c, B : \textbf{string}]$. Exhibit an instance **I** over **S** and two distinct objects in **I** that have the same expansion. Exhibit two distinct instances over **S** with the same set of object expansions.

Exercise 21.12 Sketch an extension of the complex value algebra to provide an algebraic simulation of the calculus of Section 21.3. Give algebraic versions of the queries of that section.

♠ **Exercise 21.13** Recall the approach to creating OIDs by extending datalog to use Skolem function symbols. Consider the following programs:

$$\begin{array}{ll}
T(f_1(x, y), x) \leftarrow S(x, y) & T(f_3(x, y), x) \leftarrow S(x, y) \\
T(f_2(x, y), x) \leftarrow S(x, y) & T(f_3(y, x), x) \leftarrow S(x, y) \\
T(f_1(x, y), y) \leftarrow S(x, y), S(y, x) & T(f_4(x, y), x) \leftarrow S(x, y), S(y, x) \\
\qquad\qquad P & \qquad\qquad Q
\end{array}$$

(a) Two programs P_1, P_2 involving Skolem terms such as the foregoing are *exposed equivalent*, denoted $P_1 \sim_{exp} P_2$, if for each input instance **I** having no OIDs, $P_1(\mathbf{I}) = P_2(\mathbf{J})$. Show that $P \sim_{exp} Q$ does not hold.

(b) Following the ILOG languages [HY92], given an instance **J** possibly with Skolem terms, an *obscured* version of **J** is an instance $\mathbf{J}'$ obtained from **J** by replacing each distinct nonatomic Skolem term with a new OID, where multiple occurrences of a given Skolem term are replaced by the same OID. (Intuitively, this corresponds to hiding the history of how each OID was created.) Two programs P_1, P_2 are *obscured equivalent*, denoted $P_1 \sim_{obs} P_2$, if for each input instance **I** having no OIDs, if $\mathbf{J}_1$ is an obscured version of $P_1(\mathbf{I})$ and $\mathbf{J}_2$ is an obscured version of $P_2(\mathbf{I})$, then $\mathbf{J}_1 \equiv_{OID} \mathbf{J}_2$. Show that $P \sim_{obs} Q$.

(c) Let P and Q be two nonrecursive datalog programs, possibly with Skolem terms in rule heads. Prove that it is decidable whether $P \sim_{exp} Q$. *Hint:* Use the technique for testing containment of unions of conjunctive queries (see Chapter 4).

★(d) A nonrecursive datalog program with Skolem terms in rule heads has *isolated OID invention* if in each target relation at most one column can include nonatomic Skolem terms (OID). Give a decision procedure for testing whether two such programs are obscured equivalent. (Decidability of obscured equivalence of arbitrary nonrecursive datalog programs with Skolem terms in rule heads remains open.)

♠**Exercise 21.14** [VandBGAG92] Prove the "only if" part of Theorem 21.3.2. *Hint:* Associate traces to new object id's, similar to the proof of Theorem 18.2.5. The extension homomorphism is obtained via the natural extension to traces of automorphisms of the input.

Exercise 21.15 [HTY89]

(a) Define an operational semantics for the imperative model introduced in Section 21.4.

(b) Describe how a method in this model can simulate a *while*loop of arbitrary length. *Hint:* Use a class c with associated type **tuple**$(a : c, \ldots)$, and let $c' \prec c$. Construct the implementation of method m on c so that on input o if the loop is to continue, then it creates a new object o' in c, sets $o.a = o'$, and calls m on o'. To terminate the loop, create o' in c', and define m on c' appropriately.

(c) Show how the computation of a Turing machine can be simulated by this model.

Exercise 21.16 Prove Proposition 21.4.1. *Hint:* Use a reduction from the PCP problem, similar in spirit to the one used in the proof of Theorem 6.3.1. The effect of conditionals can be simulated by putting objects in different classes and using dynamic binding.

Exercise 21.17 Describe how monadic method schemas can be simulated in the imperative model.

Exercise 21.18 [AKRW92]

(a) Verify that the grammar $G_\mathbf{S}$ described in the proof of Theorem 21.4.6 has the stated property.

(b) How big is $G_\mathbf{S}$ in terms of **S**?

(c) Find a variation of $G_\mathbf{S}$ that has size polynomial in the size of **S**. *Hint:* Break production rules having form (v) into several rules, thereby reducing the overall size of the grammar.

(d) Complete the proof of the theorem.

Exercise 21.19 [AKRW92]

★ (a) Show that it is undecidable whether a polyadic method schema is type safe. *Hint:* You might use undecidability results for program schemas (see Bibliographic Notes), or you might use a reduction from the PCP.

★ (b) A schema is *recursion free* if there are no two methods m, m' such that m occurs in some code for m' and conversely. Show that type safety is decidable for recursion-free method schemas.

Exercise 21.20

(a) Complete the formal definition of an imperative schema simulating a relational query.

(b) Prove Theorem 21.4.2.

♠ **Exercise 21.21**

(a) Suppose that the imperative model were extended to include types for classes that have one level of the set construct (so tuple of set of tuple of atomic of class types is permitted) and that the looping construct is extended to the sets occurring in these types. Assume that the **new** command is not permitted. Prove that the family of relational queries that this model can simulate is QPSPACE. *Hint:* Intuitively, because the looping operates object at a time, it permits the construction of a nondeterministic ordering of the database.

(b) Suppose that n levels of set nesting are permitted in the types of classes. Show that this simulates QEXP^{n-1}SPACE.

Exercise 21.22

(a) Describe how the form of method inheritance used for polyadic method schemas can be simulated using the originally presented form of method inheritance, which is based only on the class of the first argument.

(b) Suppose that a base method m_R in an instance of a polyadic method schema is used to simulate an n-ary relation R. In a simulation of this situation by an instance of a conventional OODB schema, how many OIDs are present in the class on which m_R is simulated?

Exercise 21.23 Show how to encode or, not, and $equal$ using method schemas.

Exercise 21.24 Show how to encode $pred_k^i$ and the join operation using method schemas.

♠ **Exercise 21.25** [HKR93] Prove Theorem 21.4.8. *Hint:* Show first that method schemas can simulate relational algebra and then inflationary fixpoint. For the fixpoint, you might want to use $pred_k$. For the other direction, you might want to simulate method schemas over ordered databases by inflationary fixpoint.