Typed Sets as a Basis for Object-Oriented Database Schemas*

Herman Balsters[1], Rolf A. de By[1], Roberto Zicari[2]

[1] Computer Science Department, University of Twente,
P.O. Box 217, 7500 AE Enschede,
The Netherlands
[2] Fachbereich Informatik, Johann Wolfgang Goethe-Universität,
Robert Mayerstraße 11–15,
D–6000 Frankfurt am Main,
Germany

Abstract. The object-oriented data model TM is a language that is based on the formal theory of FM, a typed language with object-oriented features such as attributes and methods in the presence of subtyping. The general (typed) set constructs of FM allow one to deal with (database) constraints in TM.
The paper describes the theory of FM, and discusses the role that set expressions may play in conceptual database schemas. Special attention is paid to the treatment of constraints, and a three-step specification approach is proposed. This approach results in the formal notion of database universe stated as an FM expression.
Keywords: object-oriented databases, constraint specification, sets, type theory.

1 Introduction

The language TM is a (high-level) object-oriented data model that has been developed at the University of Twente and the Politecnico di Milano. It contains the basic elements that one would expect from a state-of-the-art OO model, but with *important new features*, two of which we will discuss in this paper:

1. predicative description of sets (predicative sets as complex values), and
2. static constraints of different granularity (object level, class level, database level).

The language TM is a language for describing conceptual schemas of object-oriented databases. As it stands, TM is not a database programming language, like for instance Galileo [3] or that of O_2 [21], although the provisions to make

* This work has been partially funded by the Commission of The European Communities under the ESPRIT R&D programme: project 2443 STRETCH. Our E-mail addresses are, resp.: balsters@cs.utwente.nl, deby@cs.utwente.nl, zicari@informatik.uni-frankfurt.de.

it one are fairly straightforward. The strength of TM stems from its richness as a specification language and its formal, type-theoretic, background.

The expressiveness of the language is best characterized by a list of its typical features. It has

- complex objects, formed from arbitrarily nested records, variant records, sets, and lists[3];
- object identities (oid's);
- multiple inheritance;
- method and method inheritance;
- composition links that allow to refer from one object type to another; and
- static type-checking.

The paper does not present the full syntax of TM, but instead presents significant examples to illustrate the main features of the model.

The language TM is not unique in that it is based on a formal theory; other notable examples are IQL [1], Iris [22], LIFE [2], Logres [11], Machiavelli [24, 25], and F-logic [19]. However, we know of no other language that is formally capable of dealing with arbitrary set expressions and (their) powertypes (or set types) in the context of subtyping and multiple inheritance. In fact, TM is a syntactically sugared version of its underlying Formal Model, FM. Aspects of FM are treated in Sect. 3. The aim of this paper is to illustrate the importance of sets for conceptual schemas, and, indeed, for object-oriented database systems in general.

The essential novelty of TM is that it allows arbitrary set expressions as well-typed expressions. Like existing systems such as ORION [7, 20], O_2 [21] and GemStone [15], TM allows to handle expressions that denote enumerated sets. Moreover, it also allows to use set expressions that are formed by *set comprehension*. Such expressions take the form $\{ x : \sigma \mid \phi(x) \}$, where $\phi(x)$ is a boolean expression in which the variable x may occur. A set expression of this form is called a *predicative set*. Together with the (well-founded) introduction of logical constructs it allows to formally deal with such necessary elements in conceptual schemas as constraints and the collection of allowed database states.

In TM, set expressions are either enumerated sets or predicative sets. If the elements of some set are of type σ then that set is of type $\mathbb{P}\sigma$, the powertype of σ as defined in Sect. 3.3. In systems like O_2 and ORION, this type is denoted by $\{ \sigma \}$. Like all other expressions, set expressions are typed within a context of subtyping, which means that expressions may have many types.

The topic of this paper concerns general set constructs and (static) constraints in an object-oriented model, both at the level of a TM specification and at the level of its formal FM counterpart. Dynamic aspects such as method specification and method invocation, updates, and dynamic constraints will not be discussed. Dynamical issues are, of course, important, but treatment of these

[3] In the rest of the paper we will not deal with variant records and lists for reasons of a clean exposition.

issues is beyond the scope of this paper. We refer to [33] for full coverage of TM and FM supplied with methods and views.

The TM language is currently being implemented as a high-level database interface. With respect to the topic of this paper, it is important to remark which problems arise for implementing constraints, and predicative sets in general. Constraint specifications can be used in many ways, for instance, to define the notion of transaction safety, or to deduce properties of the database application. Our first prototype generates testing code from the constraint specification to determine whether the constraint is satisfied. We are currently investigating less naive and more efficient ways to do so, following lines as were set out by [26, 28].

This paper is organized as follows. A small example application, involving (the recurring) employees, managers, secretaries, and departments, is described in Sect. 2. The formal theory of FM, which is based on the Cardelli type system [13], is described in more detail in Sect. 3 and in the Appendix. We refrain from offering denotational semantics for this extended Cardelli type system solely for lack of space; we refer the interested reader to [5, 6, 32]. To define the mathematical meaning of TM in terms of FM we translate the structural concepts of TM into elements of FM where the mathematical meaning is precisely given. This translation is described in Sect. 4.

2 Motivation with Examples

We will demonstrate the new features of the TM language by means of an example. We do not describe in this paper the full syntax of the language but rather present significant examples in order to illustrate the main features of the underlying model.

In this section we show two features of the TM language, namely the possibility to:

- use predicative descriptions of sets and
- express static constraints with different granularity (i.e., for objects in isolation, class extensions, and database states).

2.1 Predicative Sets

Most of the theoretical proposals and system implementations of object-oriented data models are restricted to the possibility of defining only enumerated sets [1, 8, 9, 21, 24]. Besides enumerated sets, we allow the definition of predicative sets in TM.

Example 1. Let us consider the following three class declarations (which also show the possibility to define inheritance and composition links in TM):

```
Class Employee with extension EMP
  attributes
      name    : string
```

```
      age    : integer
      spouse : Employee
      gender : string
end  Employee

Class Manager ISA Employee with extension MAN
  attributes
      friends : ℙEmployee
end  Manager

Class Secretary ISA Employee with extension SEC
  attributes
      boss : Manager
 end  Secretary □
```

The three classes are related to each other by an inheritance relationship. Before defining the predicative sets feature, we first discuss our notion of inheritance. TM's inheritance is based on an extension of Cardelli's subtyping relation $\leq$ [13]. In this paper, it is extended to powertypes (powertypes start with the type constructor $\mathbb{P}$). A formal definition of the notion of subtyping in TM can be found in Sect. 3.

We give an example of a predicative set in the context of the declarations given above. Consider the following **Manager** object with attributes **name**, **age**, **spouse**, **gender** and **friends**:

```
( name="Martin", age=40, spouse=Martin's_Wife, gender="Male",
  friends={x : Employee | (x·name = "Black" or x·name = "Smith") and
                          x·age < 40} ),
```

where **Martin's_Wife** is some **Employee** object.

What is relevant here is the definition of the values of the set of friends of Martin which is given by a predicate without enumerating all values of the set.

In the example x is a member of the extension of class **Employee**. This means that the set of friends is a subset of the existing employees stored in the database. That is, Martin's friends are solely those employees (stored in the database) with name Black or Smith and who are younger than 40.

Another interesting point here is that whenever the extension associated to class **Employee** changes (e.g. by addition/deletion of employees in it) the set of friends of Martin is automatically derived by the definition of the associated predicate and its definition need not be changed. That is, the set-valued attribute **friends** is a derived attribute and depends on the actual state of the extension of class **Employee**. Such attributes can be compared to the approach taken in [31], where query statements can be used as attribute values.

In general, TM's predicative sets can be defined using a full first-order logic with typed variables and special boolean expressions. The general format of a predicative set is

$$\{ x : \sigma \mid \phi(x) \} ,$$

where $\phi(x)$ is a boolean expression, and σ is a class name or a type. If σ is a class name, x is implicitly taken to range over σ's class extension.

The boolean expressions of TM are either

- *constants:* **true** and **false**
- *logical formulas*, built up as follows:
  - if e is a boolean expression, then so is **not** e
  - if e and e' are boolean expressions, then so are (e **or** e'), (e **and** e'), (e **implies** e')
  - if e is a boolean expression, x is a variable and σ is a type, then so are **forall** $x : \sigma \bullet (e)$ and **exists** $x : \sigma \bullet (e)$
- *special boolean expressions:* if e and e' are TM-expressions then so are the expressions (e **isa** e'), ($e = e'$), (e **in** e'), (e **sin** e'), and (e **subset** e').

The special boolean expressions are treated in more detail in Sect. 3, but we offer a few words of comment here so that the reader can get an idea of what is intended by these constructs.

- ($e = e'$): this expression holds if both expressions e and e' evaluate to the same value.
- (e **isa** e'): this expression holds if e is a specialization of e'. An example of such a (well-typed and true) boolean expression is

  ⟨age=3, name="Jones", salary=5000⟩ isa ⟨age=3, name="Jones"⟩ ,

  since there are two respective types of these (record-)expressions that are in the subtype relation and, furthermore, the first expression is indeed a specialization of the first one.
- (e **in** e'): this expression holds if the element e evaluates to a member of the evaluation of the set e'.
- (e **sin** e'): this expression holds if there exists an element e'' such that both (e **isa** e'') and (e'' **in** e') evaluate to true. In other words, e can be considered a specialization of an element of e'.
- (e **subset** e'): this expression holds if the evaluation of the set e is contained in the evaluation of the set e'.

In the remainder of this section we will offer some examples employing these special boolean expressions. We note that we have also used some arithmetical relations (like $\leq$ and $>$) in these examples. Strictly speaking, we should have also listed these relations above when discussing our logical formulas, but we will tacitly assume that reader knows how to incorporate such constructs in the language.

Example 2. Suppose we want to modify the original predicative definition of the set of friends of Martin, such that now all Martin's friends have to be married with an employee younger than 35. We will not repeat the other attribute values, and focus only on that of `friends`:

```
friends={x : Employee | (x·name = "Black" or x·name = "Smith") and
                        x·age < 40 and
                        exists y:Employee • (x·spouse = y and y·age < 35)}
```

The additional condition (**exists** y:**Employee** • (x·**spouse** = y)), now implies that all of Martin's friends are married to employees younger than 35. □

We give another example of a predicative set definition in TM.

Example 3. Let us consider again Martin's friends (recall that Martin is a manager), and suppose we want an additional condition which imposes that his friends are all married to secretaries. In TM we write:

```
friends= {x : Employee | (x·name = "Black"  or x·name = "Smith")  and
                         x·age < 40 and
                         exists y:Secretary • (y isa x·spouse and y·age < 35)} □
```

The last condition contains the special boolean expression (y **isa** x·**spouse**) between two typed expressions: x·**spouse** of type **Employee** and y, which is of type **Secretary**. This **isa**-expression tests whether the value of y, viewed as an **Employee**, is equal to that of a spouse (who is an employee). Informally this means comparing the common attrinutes of a secretary object and an employee object to decide whether they are equal. This possibility of "viewing" objects at different levels of abstraction according to their position in the type hierarchy is called *object context* in TM.

To be correct, the above definition would require some additional constraints on the gender of spouses (e.g., the gender of two married employees must be different). We will introduce them later in this section, when introducing the TM mechanism to specify constraints.

We conclude this informal introduction to TM's predicative sets giving a last example.

Example 4. We want to add yet another condition to the set of friends of Martin: every spouse of Martin's friends (who are all married to secretaries by previous conditions) is also a friend of a manager. We write:

```
friends= {x : Employee | (x·name = "Black" or x·name = "Smith") and
                         x·age < 40 and
                         exists y:Secretary • (y isa x·spouse and y·age < 35 and
                               exists z:Manager • y sin z·friends)} □
```

The last condition in the above predicate contains the boolean expression y **sin** z·**friends**). Informally speaking this expression implies the object y (our secretarial spouse) to be at least in one manager's set of friends.

Just as '=' has an accompanying notion in **isa** to deal with subtypes, **in** has **sin** as an analogous notion. This means that e **sin** e' holds whenever e **isa** e'' holds for some e'' for which e'' **in** e'.

We turn now our attention to the mechanism of specifing constraints in TM.

2.2 Constraints

In TM it is possible to incorporate static constraints at three different levels:

- Objects,
- Class extensions, and
- Database states.

In this paper, we are interested in the specification of constraints rather than in their evaluation or enforcement. We will illustrate each type of constraint by means of an example.

Object and Class Constraints. Constraints can be associated to objects (i.e., they should hold for *each* individual object of a certain class; we call them *object constraints*), and to class extensions (i.e., they should hold for each extension of a certain class; we call them *class constraints*). Both object and class constraints are defined within the corresponding TM class definition. Our language makes a syntactical distinction between the two kinds of constraints because of notational conciseness: object constraints need not be explicitly (universally) quantified over all possible objects. These features are demonstrated by an example.

Example 5. Consider the class hierarchy as defined in Example 1. Suppose we want to define some object and class constraints for the class Employee as follows:
Object constraints (added to the definition of class **Employee**) :

- "the gender of an employee is either Male or Female". In TM we write:

  c_1 : (**gender** = "**Male**" **or** **gender** = "**Female**")

- "Married employees have different gender":

  c_2 : (**spouse**·**gender**≠**gender**)

- "For each employee x the spouse of the spouse of x is x":

  c_3 : (**spouse**·**spouse** = **self**) □

An object constraint is always implicitly quantified over the objects of the class, e.g. the class **Employee** in the above example. Validation of object constraints can be performed for one object at a time. We remark that the example constraints are very simple; arbitrarily complex constraints are actually allowed.

Object constraints defined in a class are also inherited by the objects of its subclasses. These "inherited" constraints will be in conjunction with the constraints defined for each subclass. For an example of object constraint inheritance we refer to Sect. 4.

Typical class constraints (though not all[4]), in contrast to object constraints, involve two quantifications over the objects of that class. To validate such a constraint the least one needs is the class extension as a whole.

Example 6. We specify a class constraint for the class **Employee**:

[4] Class constraints *not* involving two explicit quantifications are, for instance, those that use aggregate functions like *sum*, *count*, etc.

- Class constraints are also added to the definition of class Employee:
  - "If two male employees have different names, also their respective spouses will have different names" (This is a rule by Dutch law, as the woman gets the husband's name.):
    c_4 : **forall** x_1, x_2 •
    ((x_1·name≠ x_2·name **and** x_1·gender= "Male" **and** x_2·gender= "Male")
    implies x_1·spouse·name≠ x_2·spouse·name)

The complete specification of the class Employee thus becomes:

Class Employee **with extension** EMP
attributes
name : string
age : integer
spouse : Employee
gender : string
object constraints
c_1 : gender= "Male" **or** gender= "Female"
c_2 : spouse·gender≠ gender
c_3 : spouse·spouse= **self**
class constraints
c_4 : **forall** x_1, x_2 •
((x_1·name≠ x_2·name **and** x_1·gender= "Male" **and** x_2·gender= "Male")
implies x_1·spouse·name≠ x_2·spouse·name)
end Employee □

Database Constraints. In TM it is also possible to define constraints between different classes. This is expressed by the so-called TM database constraints. Database constraints are defined in a separate section after the definition of the classes. Typical database constraints (though not all) involve quantifications over the objects of at least two different classes.

Example 7. For example, we may specify the following constraints:

- Database constraints (defined after the class definitions):
  - "A secretary and a manager cannot be the same employee" (i.e., the intersection of the Secretary and Manager extensions is empty):
    c_5 : **forall** x : Manager **forall** y : Secretary •
    (**not exists** z : Employee • x **isa** z **and** y **isa** z)
  - "Each manager has at least one secretary":
    c_6 : **forall** x : Manager **exists** y : Secretary • y·boss= x

3 Formal Language Features

The formal semantics of the TM language are based on those of FM, a formal model that is extensively described in [5, 6, 32]. The semantics of TM is precisely

determined by describing a mapping from TM expressions to FM expressions. This mapping is described in Sect. 4. Here, we focus on FM.

FM is an extension of the theory described in [13, 14]. The extension consists of introducing set constructs and logical formalism. A major aim is to incorporate a general set construct into the language involving first-order logical predicates; the format of such a set is $\{ x : \sigma \mid \phi(x) \}$.

The intention of this section is to give an outline of syntactical aspects of the type theory of FM; for further details and semantical issues we refer to [5, 6]. Semantical issues are, of course, important, but (solely due to reasons of lack of space) we refer the interested reader to the two articles mentioned above for a full treatment of the semantical part of our theory. For readers interested in other issues regarding subtyping in type theory we refer to [12, 16, 19, 25, 30, 34].

3.1 Cardelli Type Theory

In this section we give a brief summary of the system described in [13, 14]. For the base system, we restrict ourselves to basic types and record types. Full details of syntactical and proof-theoretical aspects can be found in the Appendix.

Expressions and Types. Types in our simplified version of the Cardelli system are either

1. *basic types*: such as `integer, real, bool, string`, or
2. *record types*: such as `⟨age:integer, name:string⟩`, and `⟨address:string, date:⟨day:integer,month:integer,year:integer⟩⟩`.

Record types have field names (*labels* or, as you wish, *attribute names*), and type components; records are permutation invariant under field components. Note that types can have a complex structure.

Expressions in our version of Cardelli type theory are either

1. *variables*: such as $x_{\texttt{integer}}, y_{\texttt{string}}, z^{\langle \texttt{name:string, sal:real} \rangle}$ or
2. *constants*: such as $1_{\texttt{integer}}, 2.0_{\texttt{real}}, \texttt{True}_{\texttt{bool}}$, or
3. *records*: such as `⟨name="john", age=17⟩`, or
4. *projections*: such as `⟨name="john", sal=1.2⟩·name`.

We note that variables and constants are typed. Type subscripts in variables and constants are usually dropped, for purposes of readability, when it is clear from the context what the type of the variable or constant is. Expressions of the form $e{\cdot}a$, where e is supposed to be some record expression, denote the projection of the record e on one of its attributes a.

All (correct) expressions in our language are typed according to (inductively defined) typing rules. If an expression e has type σ then this is written as $e : \sigma$.

Example 8. For example, the expression

`⟨age=3,name="john",date=⟨day=27,month=2,year=1991⟩⟩`

is typed as

`⟨age:integer,name:string,date:⟨day:integer,month:integer,year:integer⟩⟩`. □

Subtyping. The set of types T is equipped with a subtyping relation $\leq$. We speak of subtyping when there exists a partial order $\leq$ on the set of types, and the typing rules are extended to the effect that an expression e that has type σ may occur at a position where an expression of supertype τ of σ is expected. Or, in other words,

$$e : \sigma,\ \sigma \leq \tau \Rightarrow \quad e : \tau \ .$$

Subtyping represents specialization/generalization characteristics of expressions and types.

Example 9. We have

$$\texttt{(age:integer, name:string, address:string)} \leq \texttt{(age:integer, name:string)} \ ,$$

because the former type has all of the properties (here reflected by the (suggestive) attribute names age and name) of the latter type, but also has an extra property, namely that it has the additional field component tagged address. Hence any expression of the first type can also be typed as the second type. □

The subtyping relation introduces polymorphism into the language, in the sense that expressions can now have more than just one type. We shall now describe a way of attaching a *unique* type to a correctly typed expression by means of so-called *minimal typing.*

Minimal Typing. As we have seen above, subtyping allows expressions to have many types. It turns out, however, that every correctly typed expression has a unique so-called *minimal type* [5, 27]. An expression e has minimal type τ, if $e : \tau$ and there is no type σ such that $e : \sigma$ and $\sigma \leq \tau$. If σ is the minimal type of e then we write $e :: \sigma$. Another important property of minimal typing is that if $e :: \sigma$ and $e : \tau$, for some type τ, then it is always the case that $\sigma \leq \tau$. The minimal type of an expression, say e, conveys the most detailed type information that can be attached to e; any other type that can be attached to e will be a generalization (supertype) of the minimal type and will thus have filtered out certain information regarding e's components. We will make frequent use of minimal typing in our language, since it is very handy to have a unique type available for expressions in the presence of subtyping polymorphism. We note that this language, as well as the extended language with predicates and sets, is statically type checkable [5, 6, 13]. Exact typing rules are given in the Appendix.

3.2 Adding Predicates – The Logical Formalism

The actual extension of the language that we would like to achieve, is that we wish to incorporate predicatively described sets, of which the general format is $\{\ x : \sigma \mid \phi(x)\ \}$, where $\phi(x)$ is a boolean expression. It is the purpose of this section to describe what the expressions $\phi(x)$ look like.

Let e, e' be boolean expressions and σ be some type. Then so are $\neg\,(e), (e \Rightarrow e')$ and $\forall\, x : \sigma \bullet (e)$ boolean expressions, with the obvious intuitive meanings. (We

note that the other propositional connectives like $\wedge$, $\vee$ and $\Leftrightarrow$, as well as the existential quantifier $\exists$, can be defined in terms of the above mentioned constructs in the usual way.)

Furthermore, we have boolean constructs of the form $(e = e')$ and $(e \lhd e')$. The boolean construct $(e = e')$ is used to test whether the expressions e, e' are *equal*, and the second construct is used to test whether expression e is a *specialization* of the expression e'. We shall now discuss these two constructs in more detail.

The expression $(e = e')$ is correctly typed if the expressions e, e' are correctly typed and, moreover, both expressions have exactly the same typing possibilities. This is the same as saying that the minimal types of e, e' are the same.

The expression $(e \lhd e')$ is correctly typed if the expressions e, e' are correctly typed and, moreover, expression e has at least the same typing possibilities as expression e'. This can also be expressed by saying that the minimal type of e is a subtype of the minimal type of e'.

Example 10. The expression

```
⟨age=3, name="john", registered=True⟩◁⟨name="john", registered=False⟩ ,
```

is correctly typed because the first record has a minimal type that is smaller than the minimal type of the second record. The expression evaluates, however, to *false*, because the left hand side subexpression is not a specialization of the right hand side. □

3.3 Adding Powertypes and Sets

Set Membership and Subtyping. If σ is a type then $\mathbb{P}\sigma$ denotes the powertype of σ. Intuitively, a powertype $\mathbb{P}\sigma$ denotes the collection of all sets of expressions e such that e satisfies $e : \sigma$. Note that a powertype as well as elements thereof can be infinite, depending on the particular type. The powertype constructor resembles the construction of the powerset $\mathcal{P}(V)$ of a set V in ordinary set theory. An expression e in our language is called a *set* if it has a powertype as its type; i.e. $e : \mathbb{P}\sigma$, for some type σ. We stress here that a set in our theory is an *expression* and not a type; i.e. we add to the set of types special types called powertypes, and, in addition, we add to the set of expressions special expressions called sets. Powertypes obey the following rule regarding subtyping: if $\sigma \leq \tau$ then $\mathbb{P}\sigma \leq \mathbb{P}\tau$. Typing rules for sets are given, for reasons of simplicity, in terms of minimal typing.

Set membership is denoted by $\in$ and indicates that an element occurs, modulo the $=$-relation, in some set. We have the following rule for typing of set-membership

$$(e \in e') :: \texttt{bool}, \text{ whenever } e' :: \mathbb{P}\sigma \text{ and } e :: \sigma \ .$$

We also have a *specialized* version of set membership, denoted by ε, indicating that an element occurs, modulo the $\lhd$-relation, in some set. We have the following typing rule for specialized set membership

$$(e \,\varepsilon\, e') :: \texttt{bool}, \text{ whenever } e' :: \mathbb{P}\sigma', e :: \sigma \text{ and } \sigma \leq \sigma' \ .$$

Intuitively, $e \,\varepsilon\, e'$ holds if for some e'' we have $e \lhd e''$ and $e'' \in e'$.

Consider, as an example, the constants `1.0` and `2.0` of type **real** and the **integer** constant `2`, and postulate that **integer** $\leq$ **real**. We then have that $2.0 \in \{1.0,\ 2.0\}$ is correctly typed and evaluates to true, whereas the expression $2 \in \{1.0,\ 2.0\}$ cannot even be evaluated for the simple reason that it is wrongly typed. It does hold, however, that $2\,\varepsilon\,\{1.0,\ 2.0\}$, since $2 \lhd 2.0$.

Enumerative and Predicative Sets. The first set construct that we have is *set enumeration*. To get a clean typing of enumerated set terms, all of the form $\{e_1, \ldots, e_m\}$, we will only allow expressions $e_1, \ldots, e_m$ that have exactly the same typing possibilities; i.e. the expressions $e_1, \ldots, e_m$ must have the same minimal type. We have the following rule for the typing of enumerated sets:

$$\{e_1, \ldots, e_m\} :: \mathbb{P}\sigma, \text{ whenever } e_k :: \sigma, \text{ for all } k\,(1 \leq k \leq m) \ .$$

This rule might seem like a rather severe restriction, but we have some very good reasons for doing so, which we now explain by means of an example.

Example 11. First, consider an alleged enumerated set like $\{1,\ 2.0\}$, in which an integer and a real element occur. As in ordinary set theory, we would like this set to be in the $=$-relation with the predicatively described set $\{x : \sigma \mid x = 1 \vee x = 2.0\}$, where we still have to decide on the choice of the type σ. Obviously, we have only two choices for σ: it is either **integer** or **real**. But no matter which of the types is chosen for σ, we always get a type inconsistency, which a quick look at the two components $x = 1$ and $x = 2.0$ in the latter predicative set reveals. Since the minimal type of x can only be either **integer** or **real**, one of the two components will be incorrectly typed, hence yielding the type inconsistency. (Note that if the integer `1` in the set $\{1,\ 2.0\}$ is replaced by the real number `1.0` then we would not have such a type inconsistency.) □

We now describe *predicative sets*. A predicative set is of the form $\{x : \sigma \mid e\}$, where e is some boolean expression possibly containing the variable x. Intuitively, such a set denotes the collection of all those expressions of type σ satisfying the predicate e.
For example, $\{x : \mathbf{real} \mid (x = 1.0) \vee (x = 2.0)\}$ is the set corresponding to the expression $\{1.0,\ 2.0\}$. We have the following rule for the typing of predicative sets:

$$\{x : \sigma \mid e\} :: \mathbb{P}\sigma, \text{ whenever } e :: \mathbf{bool} \ .$$

We now come back to some remarks made above concerning the (rather severe) typing of enumerated sets. In order to make our theory of sets intuitively acceptable, we would, of course, want to have the enumerated set $\{1.0,\ 2.0\}$ to be in the $=$-relation with the predicative set $\{x : \mathbf{real} \mid (x = 1.0) \vee (x = 2.0)\}$. (Intuitively these two constructs describe exactly the same set). Now, the only way to achieve this is to assure that the minimal types of these two sets are exactly the same. It is for this reason, and our definition of set membership, that we have chosen for our strict rule of only allowing expressions, occurring in some enumerated set, that have exactly the same minimal type.

Union, Intersection and Set Difference. Given two sets e and e', we can also form the union, intersection and difference of these two sets.

Consider the two, very simple, sets {1} and {2.0}. If we denote the union of these two sets by {1} ∪ {2.0}, then we, intuitively, would expect this union to be in the =-relation with the set {1,2.0}. But then, we get erroneous results: the latter, enumerated, set is not correctly typed since it contains both an integer and a real element, while both elements should have had the same minimal type. So, somehow, the union of the two mentioned sets should not be allowed. It would be allowed, however, either if the first set had been the real set {1.0}, or if the second had been the integer set {2}. This suggests that of two sets the union can be taken only if those sets have the same minimal type, and this is indeed the choice that we will make when taking the union of two sets: $e \cup e'$ is correctly typed if e and e' have the same minimal powertype. We emphasize that this choice is not at all arbitrary; complete coverage of all aspects regarding this matter is, however, beyond the scope of this paper. For full details the reader is referred to [6] and [32] . Analogous arguments pertain to taking set intersection and set difference.

We now state our rules for the typing of union, intersection, and difference of sets.

$$(e \cup e'),\ (e \cap e'),\ (e - e') \ ::\ \mathbb{P}\sigma,\ \text{whenever } e, e' :: \mathbb{P}\sigma \ .$$

We mention, finally, that we also have a subset relation between sets, denoted by $(e \subseteq e')$, with the following typing rule

$$(e \subseteq e') :: \mathit{bool},\ \text{whenever } e, e' :: \mathbb{P}\sigma \text{ (for some type } \sigma) \ .$$

(We note that we can define a specialized form $\sqsubseteq$ of the subset relation $\subseteq$, analogous to $\in$ and ε: $e \sqsubseteq e'$ holds if, for some e'', it holds that $e \triangleleft e''$ and $e'' \subseteq e'$.)

4 Mapping Higher-order OO Concepts

4.1 Introductory Remarks

The purpose of the present section is to show how to give a formal meaning to a database schema specified in TM. We will do so by supplying a mapping of TM constructs to FM constructs. As FM is a language with a complete type theory (as well as a formal semantics) we will, in the end, have supplied a semantics for TM.

Providing the precise meaning of a database schema means to characterize the allowed database states and the allowed operations on such states. Operations are not the topic of this paper. A full translation of the example of Sect. 2 is not given here; the interested reader is referred to [4].

4.2 Three-level Methodology

The methodology that we adopt here to describe the set of allowed states of our database is essentially that of [10], where it was used to give a set-theoretic foundation of relational databases.

The methodology consists of three levels of description, each of which makes use of the earlier level(s), if any. At level 0 we start off by describing the object types associated with the classes of interest to the database schema. At this level we also describe the set of allowed (possible) objects of those types. Actually, at each level we will describe a type, and a set of instances of that type. This set describes a further restriction on the possible instances by taking constraints into account. In the following, by *class extension* of a certain class we will mean a collection of objects of the class' associated type.

The three levels that are used in our methodology are

0. the **object level**, in which the object types of interest are described *as well as*, for each object type, the set of allowed objects of that type,
1. the **class extension level**, in which the set of allowed class extensions for each class is described, and
2. the **database level**, in which the set of allowed database states is described.

Each of the above mentioned sets of allowed instances will be a set expression in FM. For a class `C` we will, at the object level, identify `C`'s object type, which we call γ. Then, we define the expression `C_Universe` :: $\mathbb{P}\gamma$ such that `C_Universe` is the set of allowed objects of class `C`. At the class extension level, we will proceed with identifying `C_ClassUniverse` :: $\mathbb{P}\mathbb{P}\gamma$ as the set of allowed class extensions for class `C`. An element of `C_ClassUniverse` is thus a possible class extension of class `C`. At the database level, finally, we will define `DatabaseUniverse` as the collection of allowed database states. We have chosen to view a database state as a record, such that its attributes denote class extension names. As a consequence, a typical type for `DatabaseUniverse` is

$$\texttt{DatabaseUniverse} :: \mathbb{P}(\texttt{C}_1 : \mathbb{P}\gamma_1, \ldots, \texttt{C}_n : \mathbb{P}\gamma_n) \ ,$$

where $\texttt{C}_1$ up to $\texttt{C}_n$ denote the classes in the database schema. Note that the type of `DatabaseUniverse` is determined to be a minimal type; hence, the database cannot contain more class extensions than $\texttt{C}_1$ up to $\texttt{C}_n$. Similar observations need to be made about the variables in the table below.

We give the following tabular overview of our specification strategy.

Next, we should also be more precise as to the forms of the definitions of the sets of allowed instances. To that end, we give definition schemes for each kind of set occurring in the three levels. The class universe of a class `C` has a rather straightforward definition scheme:

$$\texttt{C\_Universe} = \{x : \gamma \mid \phi(x)\} \ .$$

The predicate $\phi(x)$ determines which objects of type γ are allowed objects; $\phi(x)$ is typically the predicate to describe attribute and object constraints.

Table 1. Overview of specification strategy

level	*type*	*allowed instances*
level 0	γ	C_Universe :: $\mathbb{P}\gamma$
level 1	$\mathbb{P}\gamma$	C_ClassUniverse :: $\mathbb{P}\mathbb{P}\gamma$
level 2	$\langle \mathtt{C}_1 : \mathbb{P}\gamma_1, \ldots, \mathtt{C}_n : \mathbb{P}\gamma_n \rangle$	DatabaseUniverse::$\mathbb{P}\langle \mathtt{C}_1 : \mathbb{P}\gamma_1, \ldots, \mathtt{C}_n : \mathbb{P}\gamma_n \rangle$

The set of allowed class extensions for class C can now schematically be defined as

$$\mathtt{C\_ClassUniverse} = \{\ X : \mathbb{P}\gamma \mid X \subseteq \mathtt{C\_Universe} \wedge \phi'(X)\ \}\ .$$

Each class extension X should obviously contain allowed instances for C only, and should thus be a subset of C_Universe. The predicate $\phi'(X)$ is be used to state further constraints on the class extension like, for instance, the requirement that at least ten objects should be in any extension of this class. In $\phi'(X)$ the class constraints are described. The expression DatabaseUniverse should finally be defined:

$$\begin{aligned}&\mathtt{DatabaseUniverse} = \\ &\quad \{\ DB : \langle \mathtt{C}_1 : \mathbb{P}\gamma_1, \ldots, \mathtt{C}_n : \mathbb{P}\gamma_n \rangle \mid \\ &\qquad \textstyle\bigwedge_{i=1}^{n} DB \cdot \mathtt{C}_i \in \mathtt{C}_i\mathtt{\_ClassUniverse} \wedge \Phi(DB)\ \}\ .\end{aligned}$$

By the generalized conjunction, this definition first of all requires each class extension in an allowed database state DB to be an allowed class extension. Furthermore, it may pose additional requirements on the database state by means of $\Phi(DB)$. This predicate is the place for all remaining structural constraints like, for instance, referential integrity between distinct class extensions. We refer the reader to Sect. 4.3 for specific examples.

4.3 Modelling of TM concepts in FM.

The class concept. Most of TM's constructs can be mapped straightforwardly to FM constructs. In general, a type associated with some class of TM will become a type in FM. TM's object constraints, class constraints, and database constraints correspond (or are mapped) to the three levels just discussed.

The TM features of Class, ISA-relationship, persistency, implicit quantification, and advanced attribute specification can all be dealt with in our approach as we map from *class specifications* in TM to *(set) expressions* in FM, thereby giving an indirect semantics of those class specifications.

Recursive data types and self. To deal with the TM features of recursive data types and the `self` concept in FM we postulate the existence of the *basic type* `oid` (see Sect. 3.1). The type `oid` is meant to capture the notion of *object identity* [18]. This type is not visibly a part of TM, but it rather allows to 'mathematically implement' TM's features in FM.

This implementation comes rather naturally by augmenting each associated type of a class with the `id` attribute of type `oid`. The obvious idea about this attribute is that it will uniquely identify an object in the class extension and that it is immune to user modifications.

We therefore have the following (FM) types associated with the classes of the examples in Sect. 2. Note that we show here the first step of the object level:

Example 12. Translation of types associated with classes.

```
employee  = ⟨id:oid, name:string, age:integer, spouse:oid, gender:string⟩
manager   = ⟨id:oid, name:string, age:integer, spouse:oid, gender:string,
             friends:ℙoid⟩
secretary = ⟨id:oid, name:string, age:integer, spouse:oid, gender:string,
             boss:oid⟩ □
```

As is illustrated here, TM's recursive type structures are broken down in non-recursive structures by changing the type of recursive attributes, like `spouse`, to `oid`. In addition, it is required that the `id`-values are unique and referential integrity holds. In other words, the `spouse`-values should be existing `id`-values in the relevant class extension. To deal with the recursive attribute `spouse:Employee` of that class the following changes have been made:

1. `employee` is augmented with the `id:oid` attribute,
2. `employee`'s `spouse` attribute is changed to be of type `oid`, and
3. allowed class extensions (i.e., each element of `Employee_ClassUniverse`) obey:
   (a) `id`-uniqueness, and
   (b) referential integrity with respect to the `spouse` attribute.

Let us first show how the above requirements on the class extension are dealt with. We complete the object level as far as employees are concerned:

Example 13. Translation of set of allowed employee objects; declaration and definition.

```
Employee_Universe :: ℙemployee
Employee_Universe = {x:employee | x·gender = "Male" ∨ x·gender = "Female"}  (c1)
```

□

Now we can deal with the two requirements at the class extension level. First, the set of allowed employee class extensions is declared, and then it is defined:

Example 14. Translation of `Employee` class extension; declaration and definition.

$$
\begin{array}{l}
\texttt{Employee\_ClassUniverse} :: \mathbb{P}\mathbb{P}\texttt{employee} \\
\texttt{Employee\_ClassUniverse} = \\
\quad \{X : \mathbb{P}\texttt{employee} \mid \\
\qquad X \subseteq \texttt{Employee\_Universe} \wedge \qquad \text{(object level)} \\
\qquad \forall x : \texttt{employee} \bullet \exists y : \texttt{employee} \bullet (x \in X \Rightarrow (y \in X \wedge x \cdot \texttt{spouse} = y \cdot \texttt{id})) \wedge \\
\qquad\qquad \text{(referential integrity)} \\
\qquad \forall x : \texttt{employee} \bullet \forall y : \texttt{employee} \bullet (x \in X \wedge y \in X) \Rightarrow \\
\qquad\quad ((x \cdot \texttt{id} = y \cdot \texttt{id} \Rightarrow x = y) \wedge \qquad \text{(oid uniqueness)} \\
\qquad\quad (x \cdot \texttt{spouse} = y \cdot \texttt{id} \Rightarrow x \cdot \texttt{gender} \neq y \cdot \texttt{gender}) \wedge \qquad (c_2') \\
\qquad\quad (x \cdot \texttt{spouse} = y \cdot \texttt{id} \Rightarrow x \cdot \texttt{id} = y \cdot \texttt{spouse}) \wedge \qquad (c_3') \\
\qquad\quad (\forall x' : \texttt{employee} \bullet \forall y' : \texttt{employee} \bullet \\
\qquad\quad (x \in X \wedge y \in X \wedge x' \in X \wedge y' \in X \ \wedge x' \cdot \texttt{id} = x \cdot \texttt{spouse} \ \wedge y' \cdot \texttt{id} = y \cdot \texttt{spouse}) \\
\qquad\quad\ \Rightarrow x' \cdot \texttt{name} \neq y' \cdot \texttt{name})) \qquad (c_4) \\
\qquad\quad \} \ \square
\end{array}
$$

The set of allowed employee objects, i.e. **Employee_Universe**, should typically deal with the object constraints of the class **Employee**. The constraint c_1 is dealt with in the above, but the other two (i.e. c_2 and c_3) are *not*. The reason for this is that these two constraints make use of TM's specific feature of recursive data types, and although they are proper object constraints in TM, they should be considered class constraints in FM. This is why they occur in a slightly altered version as c_2' and c_3' in the definition of **Employee_ClassUniverse**. Thus, we see that constraints that may conceptually be at TM's object level can be at FM's class extension level. In fact, it is even possible that one of TM's object constraints will be translated to a database constraint in FM. We will in the sequel call such a situation a *level shift*. In case of a level shift, quantifiers that were implicit in TM need to be made explicit.

The **self** concept becomes a trivial notion now. It simply denotes the **id** value of the object at hand. Thus, the constraint

$$c_3 : (\text{spouse} \cdot \text{spouse} = \textbf{self})$$

will become a formula of the form

$$c_3 : (\text{spouse} \cdot \text{spouse} = \text{id}) \ .$$

However, by the introduction of object identities we have also introduced a form of indirection, since no longer can we use an expression of the form **spouse·spouse** in the context of an object of the class **employee**. This is because **spouse** is no longer an expression of some record type (where we may apply attribute selection), but now it is of type **oid**. To obtain the **spouse** of one's **spouse** we now need to (conceptually) query the class extension. This form of indirection is also provided for in the translation.

ISA-relationships and inheritance. A word should also be said about the translation of **ISA**-relationships. This translation concerns aspects of inheritance. The aspects that are relevant in the present context are

1. inheritance of attributes,
2. inheritance of object constraints, and
3. the fact that the subclass extension can be seen as a subset of the class extension at any time.

We remark that inheritance of class constraints should *not* take place. To understand why this is the case, consider the example constraint that employees should *on average* earn more than some specific amount. This typical class constraint may well hold for the class of managers, but need not hold for that of secretaries. Thus, class constraints should not be inherited.

Object constraints should be inherited. The reason for this is that all variables that range over the class extension are universally quantified, and thus the constraint could be inherited. Actually, the translation takes care of its inheritance regardless of whether it has been explicitly specified. The constraint for ISA-inclusion in the definition of **DatabaseUniverse** is the reason for this.

The inheritance of attributes is straightforward: all attributes are inherited and sometimes their types may become more specific types as indicated in the subclass definition. This has already been illustrated in Example 12.

The inheritance of object constraints can be performed directly when there is no level shift. It takes place from the object universe of the supertype to that of the subtype. The type over which the implicit variable ranges needs to be adjusted. See for an example the translation of c_1 in the sets **Employee_Universe** and **Manager_Universe**:

Example 15. Translation of set of allowed manager objects.

Manager_Universe :: $\mathbb{P}$**manager**
Manager_Universe $= \{\ x$:**manager** $\mid x\cdot$**gender** $=$ "**Male**" $\vee\ x\cdot$**gender** $=$ "**Female**" $\}$
(c_1 inherited) □

In case of a level shift of an object constraint, it is also inherited but now only the type over which the original implicit variable ranged needs adjustment, and none of the other types.

As has been discussed, object constraints (including those that get a level shift (c_2' and c_3')) should be inherited. However, to express them properly for the **manager** and **secretary** class universes, we need to refer to the class universe of **employee**, and then by definition the constraints become a database constraint. For instance, the c_3 version for **manager** should read something like "The employee that is the spouse of some manager, should have that manager as her/his spouse". Consequently, the constraints c_2 and c_3 are getting *two* level shifts. They take effect at the database level, which is illustrated below.

Example 16. Translation continued; definition of the database universe.

DatabaseUniverse :: $\mathbb{P}$(**EMP**:$\mathbb{P}$**employee**, **MAN**:$\mathbb{P}$**manager**, **SEC**:$\mathbb{P}$**secretary**)
DatabaseUniverse =
{ **DB** : (**EMP**:$\mathbb{P}$**employee**, **MAN**:$\mathbb{P}$**manager**, **SEC**:$\mathbb{P}$**secretary**) |
DB·**EMP** $\in$ **Employee_ClassUniverse** $\wedge$

$$\mathbf{DB\text{-}MAN} \in \mathbf{Manager\_ClassUniverse} \wedge$$
$$\mathbf{DB\text{-}SEC} \in \mathbf{Secretary\_ClassUniverse} \wedge \qquad \text{(class level)}$$
$$\forall x{:}\mathbf{manager} \bullet \exists y{:}\mathbf{employee} \bullet (x \in \mathbf{DB\text{-}MAN} \Rightarrow (y \in \mathbf{DB\text{-}EMP} \wedge x \lhd y)) \wedge$$
$$\forall x{:}\mathbf{secretary} \bullet \exists y{:}\mathbf{employee} \bullet (x \in \mathbf{DB\text{-}SEC} \Rightarrow (y \in \mathbf{DB\text{-}EMP} \wedge x \lhd y)) \wedge$$
$$\text{(ISA-inclusions)}$$
$$\forall x{:}\mathbf{manager} \bullet \forall y{:}\mathbf{oid} \bullet$$
$$(x \in \mathbf{DB\text{-}MAN} \wedge y \in x\cdot\mathbf{friends}) \Rightarrow$$
$$\exists z{:}\mathbf{employee} \bullet (z \in \mathbf{DB\text{-}EMP} \wedge z\cdot\mathbf{id} = y) \wedge$$
$$\forall x{:}\mathbf{secretary} \bullet \exists y{:}\mathbf{manager} \bullet$$
$$(x \in \mathbf{DB\text{-}SEC} \Rightarrow (y \in \mathbf{DB\text{-}MAN} \wedge y\cdot\mathbf{id} = x\cdot\mathbf{boss})) \wedge$$
$$\text{(referential integrities)}$$
$$\forall x{:}\mathbf{manager} \bullet \forall y{:}\mathbf{employee} \bullet$$
$$(x \in \mathbf{DB\text{-}MAN} \wedge y \in \mathbf{DB\text{-}EMP}) \Rightarrow$$
$$(x\cdot\mathbf{spouse} = y\cdot\mathbf{id} \Rightarrow x\cdot\mathbf{gender} \neq y\cdot\mathbf{gender}) \wedge \qquad (c_2'' \text{ for } \mathbf{manager})$$
$$\forall x{:}\mathbf{secretary} \bullet \forall y{:}\mathbf{employee} \bullet$$
$$(x \in \mathbf{DB\text{-}SEC} \wedge y \in \mathbf{DB\text{-}EMP}) \Rightarrow$$
$$(x\cdot\mathbf{spouse} = y\cdot\mathbf{id} \Rightarrow x\cdot\mathbf{gender} \neq y\cdot\mathbf{gender}) \wedge \qquad (c_2'' \text{ for } \mathbf{secretary})$$
$$\forall x{:}\mathbf{manager} \bullet \forall y{:}\mathbf{employee} \bullet$$
$$(x \in \mathbf{DB\text{-}MAN} \wedge y \in \mathbf{DB\text{-}EMP}) \Rightarrow (x\cdot\mathbf{spouse} = y\cdot\mathbf{id} \Rightarrow x\cdot\mathbf{id} = y\cdot\mathbf{spouse}) \wedge$$
$$(c_3'' \text{ for } \mathbf{manager})$$
$$\forall x{:}\mathbf{secretary} \bullet \forall y{:}\mathbf{employee} \bullet$$
$$(x \in \mathbf{DB\text{-}SEC} \wedge y \in \mathbf{DB\text{-}EMP}) \Rightarrow (x\cdot\mathbf{spouse} = y\cdot\mathbf{id} \Rightarrow x\cdot\mathbf{id} = y\cdot\mathbf{spouse}) \wedge$$
$$(c_3'' \text{ for } \mathbf{secretary})$$
$$\forall x{:}\mathbf{manager} \bullet \forall y{:}\mathbf{secretary} \bullet$$
$$(x \in \mathbf{DB\text{-}MAN} \wedge y \in \mathbf{DB\text{-}SEC}) \Rightarrow \neg \exists z{:}\mathbf{employee} \bullet (z \in \mathbf{DB\text{-}EMP} \wedge x \lhd z \wedge y \lhd z) \wedge$$
$$(c_5)$$
$$\forall x{:}\mathbf{manager} \bullet \exists y{:}\mathbf{secretary} \bullet x \in \mathbf{DB\text{-}MAN} \Rightarrow (y \in \mathbf{DB\text{-}SEC} \wedge x\cdot\mathbf{id} = y\cdot\mathbf{boss})$$
$$(c_6)$$
$$\} \ \square$$

5 Conclusions and future work

In this paper, the language TM, which is used to describe conceptual schemas of object-oriented databases, is discussed. The strength of the language stems from its conceptual richness and its firm basis in a type theory that allows subtyping and multiple inheritance. One particular feature of TM, its (typed) predicative sets, are introduced and their (elementary) role in giving a formal database description is illustrated.

TM can be compared to other languages described in the literature, for instance Machiavellli [24, 25]. Machiavelli does not fully support Cardelli and Wegner's view of inheritance [12], and the authors of Machiavelli themselves also consider this somewhat unsatisfactory. Another drawback is that Machiavelli only allows for enumerated sets as set-valued objects.

We have only discussed TM's features for handling constraints in this paper. Views and methods can also be defined in TM [33]. Other research underway concerns both implementation and theoretical aspects of the language. One specific problem, which is well-known from the area of program synthesis, is that our

language allows the specification of so-called non-constructive predicative sets. Such sets do not allow a straightforward translation to program code, typically because of the use of quantifiers. The use of techniques from the area of program synthesis in our context will be the topic of another paper.

We are currently investigating the following issues in the context of TM:

- a TM-based DBMS prototype,
- a logical query language for TM-databases,
- further conceptual primitives for dealing with synchronization,
- the specification of general dynamic constraints,
- application development on top of TM, and
- the problem of schema updates, [7, 8, 29, 36, 37, 38], and that of object updates [35] in the presence of TM's three-level constraints.

6 Acknowledgements

We would like to express our thanks to Peter Apers, René Bal and Chris de Vreeze for their remarks made on an earlier version of this paper.

References

1. Object identity as a query language primitive, S. Abiteboul & P.C. Kanellakis, Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989, J. Clifford, B. Lindsay & D. Maier (eds.), ACM Press, New York, NY, 1989, pp. 159–173.
2. H. Aït-Kaci, An Algebraic Semantics Approach to the Effective Resolution of Type Equations, Theoretical Computer Science 45, 1986, pp. 293–351.
3. A. Albano, L. Cardelli & R. Orsini, GALILEO: A strongly-typed, interactive conceptual language, ACM Transactions on Database Systems 10, 2, June, 1985, pp. 230–260.
4. H. Balsters, R.A. de By & R. Zicari, Sets and Constraints in an Object-Oriented Data Model, Technical Report INF–90–75, University of Twente, Enschede, December, 1990, ISSN 0923–1714.
5. H. Balsters & M.M. Fokkinga, Subtyping can have a simple semantics, Theoretical Computer Science 87, September, 1991, pp. 81–96.
6. H. Balsters & C.C. de Vreeze, A semantics of object-oriented sets, The Third International Workshop on Database Programming Languages: Bulk Types & Persistent Data (DBPL–3), August 27–30, 1991, Nafplion, Greece, P. Kanellakis & J.W. Schmidt (eds.), Morgan Kaufmann Publishers, San Mateo, CA, 1991, pp. 201–217.
7. J. Banerjee, H.T. Chou, J.F. Garza, W. Kim et al. Data model issues for object-oriented applications, ACM Transactions on Office Information Systems 5, 1, January, 1987, pp. 3–26.
8. J. Banerjee et al., Semantics and implementation of schema evolution in object-oriented databases, Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA, May 27–29, 1987, U. Dayal & I. Traiger (eds.), ACM Press, New York, NY, 1987, pp. 311–322.

9. D. Beech, A Foundation for Evolution from Relational to Object Databases, Advances in Database Technology — EDBT '88, J. W. Schmidt, S. Ceri & M. Missikoff (eds.), Springer-Verlag, New York–Heidelberg–Berlin, 1988, pp. 251–270.
10. E. O. de Brock, Database Models and Retrieval Languages, Technische Hogeschool Eindhoven, Eindhoven, 1984, Ph.D. Thesis.
11. F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca & R. Zicari, Integrating object-oriented data modeling with a rule-based programming paradigm, Proceedings of ACM-SIGMOD 1990 International Conference on Management of Data, Atlantic City, NJ, May 23–25, 1990, H. Garcia-Molina & H. V. Jagadish (eds.), ACM Press, New York, NY, 1990, pp. 225–236.
12. L. Cardelli & P. Wegner, On understanding types, data abstraction, and polymorphism, Computing Surveys 17, 4, December, 1985, pp. 471–522.
13. L. Cardelli, A semantics of multiple inheritance, in: Semantics of Data Types, G. Kahn, D. B. Macqueen & G. Plotkin (eds.), Lecture Notes in Computer Science #173, Springer-Verlag, New York–Heidelberg–Berlin, 1984, pp. 51–67.
14. L. Cardelli, A semantics of multiple inheritance, Information and Computation 76, 1988, pp. 138–164.
15. G. Copeland & D. Maier, Making Smalltalk a Database System, Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston, MA, June 18–21, 1984, B. Yormark (ed.), ACM, New York, NY, 1984, pp. 316–325.
16. Y. C. Fuh & P. Mishra, Type inference with subtypes, Proceedings Second European Symposium on Programming (ESOP88), H. Ganzinger (ed.), Lecture Notes in Computer Science #300, Springer-Verlag, New York–Heidelberg–Berlin, 1988, pp. 94–114.
17. S. E. Hudson & R. King, Cactis: a self-adaptive, concurrent implementation of an object-oriented database management system, ACM Transactions on Database Systems 14, 3, 1989, pp. 291–321.
18. S. N. Khoshafian & G. P. Copeland, Object identity, Proceedings of the First International Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA86), N. Meyrowitz (ed.), Portland, Oregon, 1986, pp. 406–416.
19. M. Kifer & G. Lausen, F-logic: a higher-order language for reasoning about objects, inheritance, and scheme, Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989, J. Clifford, B. Lindsay & D. Maier (eds.), ACM Press, New York, NY, 1989, pp. 134–146.
20. W. Kim et al., Integrating an object-oriented programming system with a database system, Proceedings of the Second International Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA88), San Diego, CA, September, 1988.
21. C. Lécluse & P. Richard, The O_2 database programming language, Proceedings of Fifteenth International Conference on Very Large Data Bases, Amsterdam, The Netherlands, August 22–25, 1989, P. M. G. Apers & G. Wiederhold (eds.), Morgan Kaufmann Publishers, Palo Alto, CA, 1989, pp. 411–422.
22. P. Lyngbaek & V. Vianu, Mapping a semantic database model to the relational model, Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA, May 27–29, 1987, U. Dayal & I. Traiger, ACM Press, New York, NY, pp. 132–142.
23. D. Maier, A logic for objects, Workshop on Foundations of Deductive Databases and Logic Programming, Washington, DC, August, 1986, pp. 6–26.

24. A. Ohori, P. Buneman & V. Breazu-Tannen, Database programming in Machiavelli — a polymorphic language with static type inference, Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989, J. Clifford, B. Lindsay & D. Maier (eds.), ACM Press, New York, NY, pp. 46–57.

25. A. Ohori, Semantics of types for database objects, Theoretical Computer Science 76, 1, October, 1990, pp. 53–92.

26. X. Qian, The Deductive Synthesis of Database Transactions, Ph.D. Thesis, Department of Computer Science, Stanford University, Stanford, CA, November, 1989.

27. J. C. Reynolds, Three Approaches to Type Structure, in: Mathematical Foundations of Software Development, H. Ehrig et al. (eds.), Springer-Verlag, New York–Heidelberg–Berlin, 1985, Lecture Notes in Computer Science # 185, pp. 97–138.

28. T. Sheard & D. Stemple, Automatic verification of database transaction safety, ACM Transactions on Database Systems 14, 3, September, 1989, pp. 322–368.

29. A. H. Skarra & S. B. Zdonik, Type Evolution in an Object-Oriented Database, in: Research Directions in Object-Oriented Programming, MIT Press series in Computer Systems, B. D. Shriver & P. Wegner (eds.), MIT Press, Cambridge, MA, 1987, pp. 393–416.

30. R. Stansifer, Type inference with subtypes, Proceedings Fifteenth Annual ACM Principles of Programming Languages (POPL88), 1988, pp. 88–97.

31. M. Stonebraker, E. Anderson, E. Hanson & B. Rubenstein, QUEL as a data type, Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data, Boston, MA, June 18–21, 1984, B. Yormark (ed.), ACM, New York, NY, pp. 208–214.

32. C. C. de Vreeze, Extending the Semantics of Subtyping, accommodating Database Maintenance Operations, Universiteit Twente, Enschede, The Netherlands, Doctoraal verslag, August, 1989.

33. C. C. de Vreeze, Formalization of inheritance of methods in an object-oriented data model, Technical Report, INF–90–76, December, 1990, University of Twente, Enschede.

34. M. Wand, Complete type inference for simple objects, Proceedings Second Annual Symposium on Logic in Computer Science (LICS87), 1987, pp. 37–44.

35. S. B. Zdonik, Can objects change type? Can type objects change?, Proceedings of the Workshop on Database Programming Languages, Roscoff, France, September, 1987, (extended abstract), pp. 193–200.

36. R. Zicari, Primitives for schema updates in an object-oriented database system: A proposal, X3/SPARC/DBSSG OODB Task Group Workshop on Standardization of Object Database Systems, Ottawa, Canada, October 23, 1990.

37. R. Zicari, A framework for schema updates in an object-oriented database system, Proceedings of Seventh International Conference on Data Engineering, Kobe, April 8–12, 1991.

38. R. Zicari, A framework for schema updates in an object-oriented database system, in: Building an Object-oriented Database System–The Story of O_2, F. Bancilhon, C. Delobel & P. Kanellakis (eds.), Morgan Kaufmann Publishers, San Mateo, CA, 1992, pp. 146–182.

A Appendix

In the following appendix we give the syntax of FM, as discussed in the main body of the text. Throughout this appendix, T denotes the set of types and E denotes the set of expressions. We let σ and τ vary over T, and let e vary over E. C denotes a set of constants, V a set of variables. Also, we let c vary over C and x over V. The symbol L, finally, denotes a collection of labels (or attribute names). We let a and b vary over L. A set of basic types B, including for instance `integer`, `real`, and `string`, is postulated.

We have used the following abbreviations in our meta-syntax: $\mathbf{E}\sigma \in T(\ldots)$ for "there exists a $\sigma \in T$ such that ...", $\mathbf{A}\sigma \in T(\ldots)$ for "for all $\sigma \in T$ ...", $\longrightarrow$ for "implies", and $\leftrightarrow$ for "if and only if".

1. Types and expressions

$T ::= B \mid \langle L:T, \ldots, L:T \rangle \mid \mathbb{P}T$

basic types, record types, power types

$E ::= V_T \mid C_T \mid \langle L=E, \ldots, L=E \rangle \mid E{\cdot}L \mid \{E, \ldots, E\} \mid \{V_T | E\} \mid (E=E) \mid (E \lhd E) \mid (E \in E) \mid (E \varepsilon E) \mid (E \subseteq E) \mid (E \sqsubseteq E) \mid \neg(E) \mid (E \Rightarrow E) \mid \forall V_T \bullet (E)$

typed variables, typed constants, records, record projection, enumerated sets, predicatively described sets, extra boolean constructs, and logical expressions

2. Typing rules

(a) $x_\tau : \tau$

(b) $c_\tau : \tau$

(c) $e_1 : \tau_1, \ldots, e_m : \tau_m \longrightarrow \langle a_1 = e_1, \ldots, a_m = e_m \rangle : \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$

(d) $\langle a_1 = e_1, \ldots, a_m = e_m \rangle : \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle \longrightarrow \langle a_1 = e_1, \ldots, a_m = e_m \rangle{\cdot}a_j : \tau_j$

(e) $e_1 : \tau,\ \mathbf{A}\sigma \in T((\mathbf{E}j \in [1\,..\,m](e_j : \sigma)) \longrightarrow \mathbf{A}i \in [1\,..\,m](e_i : \sigma)) \longrightarrow \{e_1, \ldots, e_m\} : \mathbb{P}\tau$

(f) $e : \mathtt{bool} \longrightarrow \{x_\tau | e\} : \mathbb{P}\tau$

(g) $\mathbf{E}\tau \in T(e : \tau), \mathbf{A}\sigma \in T(e : \sigma \leftrightarrow e' : \sigma) \longrightarrow (e = e') : \mathtt{bool}$

(h) $\mathbf{E}\tau \in T(e : \tau), \mathbf{A}\sigma \in T(e : \sigma \longrightarrow e' : \sigma) \longrightarrow (e' \lhd e) : \mathtt{bool}$

(i) $\mathbf{E}\tau \in T(e' : \mathbb{P}\tau), \mathbf{A}\sigma \in T(e' : \mathbb{P}\sigma \leftrightarrow e : \sigma) \longrightarrow (e \in e') : \mathtt{bool}$

(j) $\mathbf{E}\tau \in T(e' : \mathbb{P}\tau), \mathbf{A}\sigma \in T(e' : \mathbb{P}\sigma \longrightarrow e : \sigma) \longrightarrow (e \varepsilon e') : \mathtt{bool}$

(k) $\mathbf{E}\tau \in T(e' : \mathbb{P}\tau), \mathbf{A}\sigma \in T(e' : \mathbb{P}\sigma \leftrightarrow e : \mathbb{P}\sigma) \longrightarrow (e \subseteq e') : \mathtt{bool}$

(l) $\mathbf{E}\tau \in T(e' : \mathbb{P}\tau), \mathbf{A}\sigma \in T(e' : \mathbb{P}\sigma \longrightarrow e : \mathbb{P}\sigma) \longrightarrow (e \sqsubseteq e') : \mathtt{bool}$

(m) $e : \mathtt{bool}, e' : \mathtt{bool} \longrightarrow \neg(e) : \mathtt{bool}, (e \Rightarrow e') : \mathtt{bool}, \forall x_\tau \bullet (e) : \mathtt{bool}$

3. Subtyping rules

Postulate a partial order $\leq_B$ on B.

The relation $\leq$ is defined by

(a) $\beta \leq_B \beta' \longrightarrow \beta \leq \beta'$

(b) $\tau_1 \leq \tau_1', \ldots, \tau_n \leq \tau_n' \longrightarrow \langle a_1 : \tau_1, \ldots, a_n : \tau_n, b_1 : \sigma_1, \ldots, b_m : \sigma_m \rangle \leq \langle a_1 : \tau_1', \ldots, a_n : \tau_n' \rangle \ \ (m, n \geq 0)$

(c) $\tau \leq \tau' \longrightarrow \mathbb{P}\tau \leq \mathbb{P}\tau'$

4. Minimal typing rules

(a) $x_\tau :: \tau$

(b) $c_\tau :: \tau$

(c) $e_1 :: \tau, \ldots, e_m :: \tau_m \longrightarrow \langle a_1 = e_1, \ldots, a_m = e_m \rangle :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$

(d) $e :: \langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle \longrightarrow e{\cdot}a_i :: \tau_i \ (1 \leq i \leq n, n \geq 1)$

(e) $e : \mathtt{bool} \longrightarrow e :: \mathtt{bool}$

(f) $e_1 :: \tau, \ldots, e_m :: \tau \longrightarrow \{e_1, \ldots, e_m\} :: \mathbb{P}\tau$

(g) $\{x_\tau | e\} :: \mathbb{P}\tau$

(h) $e :: \tau,\ e' :: \tau \longrightarrow (e = e') :: \mathtt{bool}$

(i) $e :: \tau,\ e' :: \tau',\ \tau \leq \tau' \longrightarrow (e \lhd e') :: \mathtt{bool}$

(j) $e :: \tau, e' :: \mathbb{P}\tau \longrightarrow (e \in e') :: \mathtt{bool}$

(k) $e :: \tau,\ e' :: \mathbb{P}\tau',\ \tau \leq \tau' \longrightarrow (e \varepsilon e') :: \mathtt{bool}$

(l) $e, e' :: \mathbb{P}\tau \longrightarrow (e \subseteq e') :: \mathtt{bool}$

(m) $e :: \mathbb{P}\tau,\ e' :: \mathbb{P}\tau',\ \tau \leq \tau' \longrightarrow (e \sqsubseteq e') :: \mathtt{bool}$

(n) $e :: \mathtt{bool},\ e' :: \mathtt{bool} \longrightarrow \neg(e) :: \mathtt{bool},\ (e \Rightarrow e') :: \mathtt{bool},\ \forall x_\tau \bullet (e) :: \mathtt{bool}$