Object-Orientation Axiomatised by Dynamic Logic

Alfons Kemper[*] *Guido Moerkotte*[+] *Klaus Peithner*[*]

[*]Lehrstuhl für Informatik III
RWTH Aachen
Ahornstraße 55
D–5100 Aachen, F. R. G.
kemper|peithner@informatik.rwth-aachen.de

[+]Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten 5
D–7500 Karlsruhe, F. R. G.
moer@ira.uka.de

Abstract

The field of object-oriented databases is suffering from the lack of a—widely agreed upon—formal model. In this paper we develop the basics of a formal model for an object-oriented data model (kernel). The formalization is based on *dynamic logic*: combining *many-sorted algebras* with *states* and *state transitions*. The many-sorted algebras are utilized to model the static aspects, i.e., the value-oriented part of the object model. The state transitions account for the dynamic behavior of the object base due to object modifications. The main dynamic features that we incorporate in the formal model comprise (object) attribute assignment, object creation, object deletion, and set object manipulation, i.e., removing and inserting elements.

The virtues of the proposed model are demonstrated on three sample applications. Dangling references may lead a given program to non-terminate—we will show this fact. Second, a desirable property of a program will be verified. Last, the semantics of a simple declarative language for our object model will be specified. Utilizing the formal model we are able to prove the correctness of a particular query evaluation plan—called the "nested loop" evaluation.

1 Introduction

Object-oriented database systems seem to be maturing, witnessed by the fact that researchers—more or less—agree on the essential features of object-oriented data models [6, 29]. However, there is still no widely agreed upon formalization of object-orientation on the horizon—despite the existence of several implementations. There are serious problems resulting from the lack of such a formal model.

Without a formal foundation there can be no formal specification of the semantics of the object-oriented data model(s). Compared to the relational model—where a rigorous formal model existed before implementation efforts were undertaken—this is even more serious for object-oriented databases since they normally involve a full object-oriented programming language—thus the behavioral part plays a central role. Since experience with "simple" programming languages—as Pascal and Modula—has already revealed that it is impossible to unambiguously define their semantics verbally even more problems are to be expected with the far more complex object-oriented programming languages. Only a rigorous formal specification of the features of the object-oriented model(s) can ultimately lead to the definition of a standard. In the database area a thoroughly defined standard seems to be a prerequisite for user acceptance.

Moreover, without a formal model several implementation aspects of object base systems lack a profound basis. One aspect—that is dealt with in this presentation—is the definition of query languages. Usually, they are translated into an algebraic expression. This translation should be proven correct which, however, can only be accomplished within the framework of a formal model. Other important aspects to be formalized are update semantics, optimization of query evaluation plans, and typing issues. Last not least, a formal model as the one proposed in this paper can ultimately be envisioned as the basis for (automatic) program verification.

There exist first approaches towards a formalization of object-oriented data models. These works can be classified into three categories:

1. formal work which tries to cover the structural part of object-oriented databases, e.g., [8, 23], including the specification of algebras most of

which have their origins in the work on nested relations (NF^2), e.g., [25, 26, 28];

2. formal, logic programming based models, e.g., [4, 10, 13, 20, 21, 22]; and

3. informal work on object-oriented databases discussing features a formal model should exhibit, e.g., [9, 24].

The major challenge in formalizing object-oriented data models stems from the destructive/updating nature of object bases—which none of the above mentioned approaches handles satisfactorily. Abiteboul observes in [1] that both, the model theoretic and the proof theoretic approaches fall short of capturing the dynamic aspects of object bases to account for the creation, deletion and modification of objects. Beeri [9] suggests λ-calculus for specifying the behavioral component of object bases; however, we examine an alternative approach based on dynamic logic.

Both, Abiteboul [1] and Beeri [9] hint that *state-based* models may be worthwhile investigating as a basis for formalizing the semantics of object models. However, to our knowledge nobody has pursued this clue so far. In this paper we propose such a state-oriented model based on *dynamic logic* [12]. The dynamic logic framework enables the formalization of dynamic aspects by having two different levels of reasoning: the *internal* logic level which reflects the logic applied within the machine and the *external* logic which enables reasoning about programs (see also [12] for a detailed discussion of this issue). Since the external level is also presented within a formal logic it allows to formally reason about programs, and hence, about updates. Eventually, the development of a proof theory would even allow to have computer support for the reasoning about programs. This could ultimately lead to an automatic program verification component which is capable of proving certain sentences about a given program.

The proposed model can be seen as a step towards a complete formal model. It can be divided into two major parts. The first part describes the *static* aspects including the structural part and the evaluation of expressions, while the second part describes the *dynamic* aspects such as control, assignments, object creation and deletion, and set manipulation. The structural

part of our model allows to define tuple-structured and set-structured objects. Additionally, we incorporate the relational model in our formal framework for two reasons mainly: one, to be downward compatible and, two, to exploit the large body of knowledge existing in the relational context. We do not incorporate complex objects (in [9] called structured values) since they are equivalent to objects (with identity) if naming of complex objects is possible. We still enforce several restrictions since we wanted to keep the first attempt towards a formal model as easy as possible: our model includes neither lists nor arrays. Neither operations nor functions may have side effects. Instead, modifications of the (state of the) object base can only be achieved by applying specific commands defined within our command language. Further, we assume that all types are predefined. Thus, we cannot (yet) formally reason about schema updates.

The remainder of the paper is organized as follows. Section 2 introduces the basic (syntax) definitions—namely the *sorts*, the *algebra*, and the *object-oriented database language*. A running example illustrates the definitions. Section 3 defines the notion of *model*. The introduction of the model is divided into two major parts: first, semantics of the static aspects is specified and, second, the dynamic aspects are treated. We then give some applications of our formal model. In Section 4 with the help of two sample programs a non-desirable property—called non-termination—and a desirable property—specified by a formula—will be derived. Further, we specify the semantics of a particular query language and its "*nested loop*" evaluation. A theorem shows the correctness of the nested loop evaluation. Section 5 concludes the paper.

2 Basic Definitions

In this section we lay out the fundamentals of our formal model. There exist two kinds of elements: *values* and *objects*. Values never change; they are "god given" [9]. Objects are identifiable, e.g., by an object identifier, and have values associated with them. The identity of an object never changes while the value association is subject to change. An object identifier as such is nothing more than a specially interpreted value.[1]

[1] In fact there may be no reason not to have any value to be an object identifier.

On the value side we distinguish base values and composite values. In our model, object identifiers are base values (This is not always the case, see e.g., [20]). As composite values we have finite sets of base values, finite tuples of base values and finite sets of tuples of base values (i.e., relations). However, since tuples and sets may be represented by an *object identifier OID* [19] our model only needs "flat" structures, i.e., there are no tuples of tuples, tuples of sets, or sets of sets. The expressive power is still the same as if we permit such structures [14].

The base values together with the sets of base values comprise the *base part* of our model, tuples and relations form the *relational part*, and the *object part*—which provides for handling of tuple- and set-structured objects—completes the fundamentals of the model (see Figure 1). Together, the base and the relational part comprise the *value part* of the formal model. Between the three parts of the model there exist (partial) mappings to transfer from one representation to another. The idea to incorporate the relational model and to provide mappings from the relational part to the object part and vice versa was first proposed in [5].

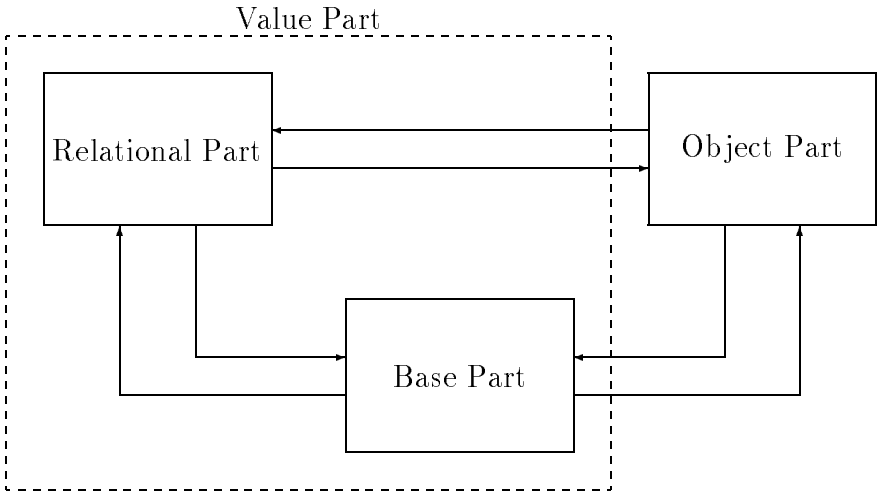

Figure 1: The Fundamentals of the Model

Why did we incorporate the relational model? There are different reasons for doing so (see also [9]):

1. Being downward compatible to relational database systems.

2. Being upward compatible to deductive database systems. This argument was also pointed out in [5]. It will become increasingly important due to the tendency of unifying object-oriented and deductive databases (witness, e.g., [2, 4, 7, 11, 20, 21]).

3. Being able to exploit the vast amount of formal work carried out in the context of the relational model—over more than two decades.

4. Many queries naturally return relations and not objects (see also [27] on this).

5. Allowing formal reasoning with new (relation-based) optimization techniques, e.g., access support relations [16, 17] and generalized materialization relations [15].

For us, these arguments seemed cogent enough to incorporate the relational model into our formal model of object-oriented databases.

2.1 Sorts

Within this subsection we start specifying the value part of our model. We assume that the set of *attribute symbols* $Attr := \{A_1, A_2, \ldots\}$ and the finite set of *type symbols* T with $ANY \in T$ are given. Associated with T there exists a subtype relationship, i.e., a transitive partial order $\leq_T$ on T permitting multiple inheritance. The only restriction we impose on $\leq_T$ is that for any $z \in T$, $z \leq_T ANY$ holds.

Definition 1 (Sort Space) *The set T is given; the* sort space *Z is defined as*

$$Z := BS \cup TS \cup SS \cup RS$$

where the subsets are defined as follows

1. $BS := \{\mathbf{BOOL}\} \cup T$
2. $TS := \{\langle A_1 : z_1, \ldots, A_n : z_n \rangle | n \in \mathbf{N_0}, A_i \in Attr \text{ pairwise distinct}, z_i \in BS\}$
3. $SS := \{\mathbf{\{}z\mathbf{\}} | z \in BS\}$
4. $RS := \{\mathbf{\{}z\mathbf{\}} | z \in TS\}$

We use $\langle\rangle$ and $\mathbf{\{\}}$ instead of [] and {} to emphasize that this is only a syntactical notion for sorts with no relation to actual domains. This way we avoid to confuse, e.g., tuple sorts with real tuples. The elements of Z are called *sorts* and are denoted by z (or z', z'') possibly with an index. The abbreviations BS, TS, SS, and RS stand for base sort, tuple sort, set sort, and relation sort, respectively. For simplicity, we restricted the base sorts to $\{\mathbf{BOOL}\} \cup T$ so far, but obviously one should incorporate **INT**, **STR**, etc.

A member of Z is associated with each type symbol by the mapping $\eta : T \to Z$. For a type sort $z \in T$, $\eta(z)$ is called the structural representation of z. We define $\eta(ANY) := \langle\rangle$. Another useful notation is the one of concatenation of tuple sorts which defines for $z = \langle A_1 : z_1, \ldots, A_n : z_n\rangle$ and $z' = \langle A'_1 : z'_1, \ldots, A'_m : z'_m\rangle$ with no attributes in common the concatenation as $z \circ z' := \langle A_1 : z_1, \ldots, A_n : z_n, A'_1 : z'_1, \ldots, A'_m : z'_m\rangle$.

2.2 Running Example

In this subsection we will introduce an example schema, called *Company*. The sorts will be used throughout the paper to illustrate our definitions. The sort/type definitions are as follows:

$$\begin{aligned} Attr &:= \{Backups, Mgr, Name, Profit, Salary, WorksIn\} \\ T &:= \{ANY, DepType, EmpType, EmpSetType, MgrType\} \\ BS &:= \{\mathbf{BOOL}, \mathbf{INT}, \mathbf{STR}\} \cup T \end{aligned}$$

Let $MgrType \leq_T EmpType$. Then, the type symbols should have the following structural representation:

$$\begin{aligned} \eta(DepType) &:= \langle Name : \mathbf{STR}, Mgr : MgrType, Profit : \mathbf{INT}\rangle \\ \eta(EmpType) &:= \langle Name : \mathbf{STR}, WorksIn : DepType, Salary : \mathbf{INT}\rangle \\ \eta(EmpSetType) &:= \mathbf{\{}EmpType\mathbf{\}} \\ \eta(MgrType) &:= \langle Name : \mathbf{STR}, WorksIn : DepType, Salary : \mathbf{INT}, \\ &\qquad Backups : EmpSetType\rangle \end{aligned}$$

Together with an example algebra the schema is illustrated in Figure 2.

2.3 Algebra

Using the notion of algebra, semantics is associated with the value part in a standard manner [12].

Definition 2 (Algebra) *Let Z be a sort space. A Z-*algebra $\mathcal{A}$ *is given by a structure $((\mathcal{A}_z)_{z\in Z}, \omega)$ where*

1. *$(\mathcal{A}_z)_{z\in Z}$ is a family of sets indexed by elements of Z called carrier sets of $\mathcal{A}$; we use the following carrier sets: $\mathcal{A}_{\mathbf{BOOL}} := \{\mathbf{1}, \mathbf{0}\}$, $\mathcal{A}_z := \mathcal{OID}_z$ for each $z \in T$, $\mathcal{A}_{\langle A_1:z_1,\ldots,A_n:z_n\rangle} := \{[A_1 : a_1, \ldots, A_n : a_n] | a_i \in \mathcal{A}_{z_i}$ for $1 \leq i \leq n\}$, $\mathcal{A}_{\langle\rangle} := \{[]\}$, and $\mathcal{A}_{\{z\}} = \mathcal{P}^{fin}(\mathcal{A}_z)$. $\mathcal{OID}_z$ denotes a (denumerably infinite) set of object identifiers and $\mathcal{P}^{fin}$ is an operator for powersets (of finite subsets).*
2. *for $z, z' \in T$ we require that $\mathcal{A}_{z'} \subseteq \mathcal{A}_z$ whenever $z' \leq_T z$. Further, we require that if $\mathcal{A}_z \cap \mathcal{A}_{z'} \neq \emptyset$ then $z \leq_T z'$ or $z' \leq_T z$. For each $z, z' \in T$ with $z' \leq_T z$ and $z' \neq z$ we require that $\mathcal{A}_z \setminus \mathcal{A}_{z'}$ is a denumerably infinite set; and*
3. *ω is a special symbol accounting for errors.*

Let $z' <_T z$ be for $z' \leq_T z$ *and* $z' \neq z$ and let $\mathcal{A}_{z!}$ be for $\mathcal{A}_z \setminus \bigcup_{z'<_T z} \mathcal{A}_{z'}$. For $z \in T$ the elements of the set $\mathcal{A}_{z!}$ will play the role of object identifiers for objects of type z. Condition 2. ensures that (the identifiers of) subtype objects are included in the carrier of the supertype. Further, let $\mathcal{A}_z^+$ be for $\mathcal{A}_z \cup \{\omega\}$.

For our example the carrier elements are signified in Figure 2.

2.4 Languages and their Syntactic Categories

Analogously to [12], we define the notion of a programming language L which is a structure $L = (Z, X)$. Z is a set of sorts known from Definition 1 and X is a family $(X_z)_{z\in Z}$ of sets of variables. The elements of X_z will often be denoted by x_z, x'_z, etc. For a language L we define the syntactic categories $Bxp(L)$, $Axp(L)$, $Pxp(L)$, $Cmd(L)$, and $Fma(L)$. How the several categories hang together is illustrated in Figure 3: there are two views—the internal and the external view on the machine. The former depends on the current state of the machine. There, a many-sorted algebra is defined and is extended

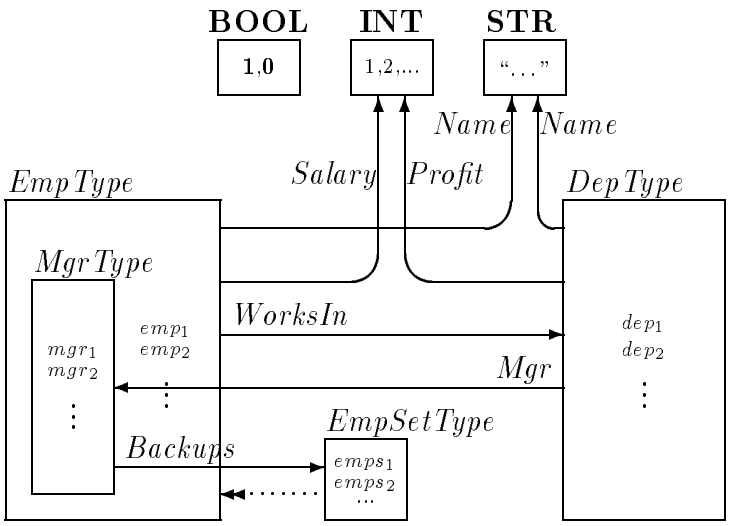

Figure 2: An Example Schema with Carrier Elements

to a set of algebraic expressions $Axp(L)$. The boolean expressions $Bxp(L)$ and the path expressions $Pxp(L)$ are subsets of this set. State transitions are invoked by the command category $Cmd(L)$. If we want to describe the states of the machine from outside we have to use the language of formulae $Fma(L)$. Formulae make assertions about one or several states and describe how the states are connected by commands. The boolean expressions which represent the internal logic on the one hand are also parts of the external view on the other hand.

Boolean expressions describe elementary predicates. They denote elements of the "truth value" data type $\{\mathbf{1}, \mathbf{0}\}^+$.

Definition 3 (Boolean Expressions) *The set* $Bxp(L)$ *of* boolean expressions *is defined as follows:*

1. $false \in Bxp(L)$,
2. *if* $z \in Z$, $\upsilon, \tau \in Axp_z(L)$, *and* $rel \in \{==, <, >\}$ *then* $(\upsilon\ rel\ \tau) \in Bxp(L)$, *and*

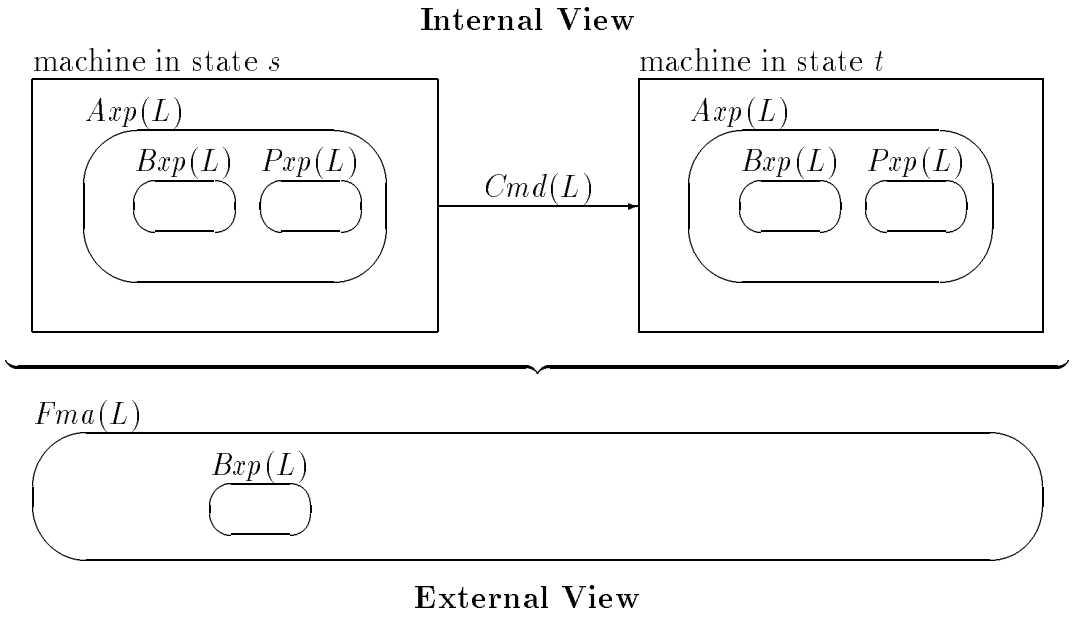

Figure 3: The Categories of the Programming Language L

3. *if* $z' \in BS \cup TS$, $z = \{z'\}$, $v \in Axp_{z'}(L)$, *and* $\tau \in Axp_z(L)$ *then* $(v \text{ in } \tau) \in Bxp(L)$.

Additionally, we introduce the abbreviation *true* for (*false* == *false*). As usual, we define the logical connectives *and*, *or*, *not*, and *implies*.

Path expressions are a particularity of an object-oriented database system and thus, they have a special significance.

Definition 4 (Path Expressions) *The set* $Pxp(L) := \bigcup_{z \in Z} Pxp_z(L)$ *denotes the* path expressions *of the language L defined as follows: for each* $z \in T$ *with* $\eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle$ *and* $v \in Axp_z(L)$ *there is* $v.A_i \in Pxp_{z_i}(L)$ *(for* $1 \leq i \leq n$*)*[2].

Algebraic expressions are required for getting data from the object base. They cannot change the object base. The set $Axp(L)$ of the algebraic expressions is subdivided into subsets $Axp_z(L)$. Thus, each algebraic expression is typed, i.e., it has as many types z as it belongs to sets $Axp_z(L)$.

Definition 5 (Algebraic Expressions) *Let L be a programming language. The set* $Axp(L) := \bigcup_{z \in Z} Axp_z(L)$ *of* algebraic expressions *is inductively defined as follows:*

1. *for all* $z \in Z$: $\mathcal{A}_z \subseteq Axp_z(L)$, $X_z \subseteq Axp_z(L)$, $Pxp_z(L) \subseteq Axp_z(L)$, *and* $Bxp(L) \subseteq Axp_{\mathbf{BOOL}}(L)$.
   Carrier elements, variables, path expressions, and boolean expressions are algebraic expressions.
2. *if* $z \in T$: $\text{ext}(z) \in Axp_{\{z\}}(L)$.
   This expression yields the extension of type z. An extension is the set of all objects of a given type existing in the object base.

There are some rules for constructing and destructing tuple values.

3. *if* $z \in BS$, $A \in Attr$, *and* $v \in Axp_z(L)$: $[A : v] \in Axp_{\langle A:z \rangle}(L)$.
4. *if* $z \in TS$, $z = \langle A_1 : z_1, \ldots, A_n : z_n \rangle$, *and* $v \in Axp_z(L)$: $v.A_i \in Axp_{z_i}(L)$ *(for* $1 \leq i \leq n$*)*.

[2]Note, that as we will see later v could itself be a path expression—thus leading to arbitrarily long path expressions.

5. *if* $z, z' \in TS$ *with no attributes in common,* $v \in Axp_z(L)$*, and* $v' \in Axp_{z'}(L)$*:* $v \circ v' \in Axp_{z \circ z'}(L)$.

There are some rules for constructing set values and "fetching" elements from sets.

6. *if* $z \in BS \cup TS$ *and* $v \in Axp_z(L)$*:* $\{v\} \in Axp_{\{z\}}(L)$.
7. *if* $z \in T$ *with* $\eta(z) = \{z'\}$ *or* $z \in SS \cup RS$*, and* $v \in Axp_z(L)$*:* $v.fetch \in Axp_{z'}(L)$.
8. *if* $v_1, v_2 \in Axp_z(L)$ *for* $z \in SS \cup RS$*:* v_1 *union* $v_2 \in Axp_z(L)$.
9. *if* $v_1, v_2 \in Axp_z(L)$ *for* $z \in SS \cup RS$*:* v_1 *setminus* $v_2 \in Axp_z(L)$.

There are three well-known operators from the relational model.

10. *if* $\epsilon \in Bxp(L)$*,* $v \in Axp_z(L)$ *for some* $z = \{z'\} \in SS \cup RS$*, and* $x \in X_{z'}$*:* $\sigma_{\lambda x(\epsilon)}(v) \in Axp_z(L)$.
11. *if* $z, z' \in BS \cup TS$*,* $v \in Axp_{\{z'\}}(L)$*,* $x \in X_{z'}$*, and* $\tau \in Axp_z(L)$*:* $\pi_{\lambda x(\tau)}(v) \in Axp_{\{z\}}(L)$.
12. *if* $v_1 \in Axp_{z_1}(L)$*,* $v_2 \in Axp_{z_2}(L)$ *with* $z_i = \{z_i'\}$*,* $z_i' \in TS$*, where* z_1' *and* z_2' *have no attribute in common,* $\epsilon \in Bxp(L)$*, and* $x \in X_{\{z_1' \circ z_2'\}}$*:* $(v_1 \bowtie_{\lambda x(\epsilon)} v_2) \in Axp_{\{z_1' \circ z_2'\}}(L)$.

There are two casting operations—also called dereferences [3, 28]—which transform an object described by an OID expression to a value expression. These casting operations are overloaded, i.e., they have the same operator symbol κ.

13. *if* $z \in T$ *with* $\eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle$ *and* $v \in Axp_z(L)$ *then* $\kappa(v)$ *is an element of* $Axp_{\langle A_1 : z_1, \ldots, A_n : z_n \rangle}(L)$*. In this case the operator* κ *casts a tuple-structured object into a tuple value; and if* $z \in T$*,* $z' \in BS \cup TS$ *with* $\eta(z) = \{z'\}$ *and* $v \in Axp_z(L)$ *then* $\kappa(v)$ *is an element of* $Axp_{\{z'\}}(L)$*. In this case the operator* κ *casts a set- or a relation-structured object into a set value or relation, respectively.*

There is one more construction rule somewhat outside of the rules we have introduced so far.

14. *if* $z, z' \in T$ *with* $z' \leq_T z$: $Axp_{z'}(L) \subseteq Axp_z(L)$. *This inclusion results from condition 2. of Definition 2 (see also item 1. of Definition 6).*

Within our running example we can, e.g., build the following algebraic expressions:

$$\begin{aligned}
x_{MgrType} &\in Axp_{MgrType}(L) \subseteq Axp_{EmpType}(L) \\
x_{MgrType}.WorksIn &\in Pxp_{DepType}(L) \subseteq Axp_{DepType}(L) \\
x_{MgrType}.WorksIn.Mgr &\in Pxp_{MgrType}(L) \subseteq Axp_{EmpType}(L) \\
\{[Mgr : x_{MgrType}, Emps : x_{EmpSetType}]\} &\in Axp_{\{\langle Mgr:MgrType, Emps:EmpSetType\rangle\}}(L) \\
(emp_1 \ in \ \kappa(x_{MgrType}.Backups)) &\in Bxp(L)
\end{aligned}$$

Commands involve state transitions—also called side effects. Later we will give a model which specifies these side effects, thus associating commands with state transitions.

Definition 6 (Commands) *Let L be a programming language. The set of commands $Cmd(L)$ is defined by the following induction closure conditions:*

1. *if* $z \in Z$, $x_z \in X_z$, *and* $v \in Axp_z(L)$ *then*

$$x_z := v \quad \in Cmd(L)$$

Assignment[3] *is allowed to variables. Note, if* $v \in Axp_{z'}(L)$ *and* $z' \leq_T z$ *holds then* $v \in Axp_z(L)$ *holds too and the assignment is legal. But otherwise if* $x_{z'} \in X_{z'}$ *and* $z' \leq_T z$ *holds then* $x_{z'} \in X_z$ *does not hold and the assignment* $x_{z'} := v$ *is not allowed.*

2. *if* $z, z' \in Z$ *with* $\eta(z) = \langle \ldots, A : z', \ldots \rangle$, $v_1 \in Axp_z(L)$, *and* $v_2 \in Axp_{z'}(L)$ *then*

$$v_1.A := v_2 \quad \in Cmd(L)$$

Assignment is also allowed to path expressions.

3. *if* $\alpha, \beta \in Cmd(L)$ *then*

$$\alpha; \beta \ \in Cmd(L)$$

[3]Note, we use the symbol ":=" in two different meanings—for assignment in the command language and for definitions in our meta language. However, the context will always solve the ambiguity.

4. *if* $\epsilon \in Bxp(L)$ *and* $\alpha \in Cmd(L)$ *then*

$$if\ (\epsilon)\ (\alpha) \quad \in Cmd(L)$$

5. *if* $z \in BS \cup TS$, $x \in X_z$, $\upsilon \in Axp_{\{z\}}(L)$, *and* $\alpha \in Cmd(L)$ *then*

$$foreach(x\ in\ \upsilon)\ (\alpha) \quad \in Cmd(L)$$

Note, that the foreach command iterates only over sets of base values or relations. Applying the project-operator κ *on the expression denoting some set-structured objects, we can easily construct a command running over these objects, e.g.,* $foreach(x\ in\ \kappa(x_{MgrType}.Backups))(\alpha)$.

6. *if* $x_z \in X_z$ *for* $z \in T$ *then*

$$x_z.create, x_z.delete \ \in Cmd(L)$$

These two commands are responsible for creating and deleting objects.

7. *if* $x_z \in X_z$ *with* $z \in T$ *with* $\eta(z) = \{z'\} \in SS \cup RS$ *and* $\upsilon \in Axp_{z'}(L)$ *then*

$$x_z.insert(\upsilon), x_z.remove(\upsilon) \ \in Cmd(L)$$

The commands insert (and remove) the object obtained by evaluating υ *into (from) the set-structured object associated with* x_z.

Formulae give us a mechanism which allows to formally state sentences about programs. We can, e.g., assert that a given program never terminates.

Definition 7 (Formulae) *The set* $Fma(L)$ *of formulae is a category of the programming language* L. *The formulae are defined as follows:*

1. $Bxp(L) \subseteq Fma(L)$,
2. *if* $\Phi, \Psi \in Fma(L)$ *then* $(\Phi \Longrightarrow \Psi) \in Fma(L)$,
3. *if* $\Phi \in Fma(L)$ *and* $\alpha \in Cmd(L)$ *then* $[\alpha]\Phi \in Fma(L)$, *and*
4. *if* $\Phi \in Fma(L)$ *and* $x_z \in X_z$ *for some* $z \in Z$ *then* $\forall x_z \Phi \in Fma(L)$.

The formula $\forall x_{MgrType} \forall x_{DepType}((x_{MgrType} == x_{DepType}) == false)$, e.g., states that two variables of type *MgrType* and *DepType*, respectively, have always different values (if they are defined). Note, the construction $(x_{MgrType} == x_{DepType})$ is legal because both variables are of type ANY.

3 Models

This section gives the definition of general and standard models. Whereas the first represents the interpretation of variables and attributes within one state, the second gives semantics to the commands and, thus, treats elementary updates and control—which necessitates state transitions.

3.1 General Model

The main part of the following definition of a model consists of several mappings associated with each state. These mappings are responsible for the interpretation of variables, attributes, and sets. Additionally, each state takes care of class membership and object identifier management.

Definition 8 (General Model) *A (general) model for a language* $L = (Z, X)$ *is a structure* $\mathcal{M} = (\mathcal{A}, S, [\![\cdot]\!])$ *where*

1. $\mathcal{A}$ *is a* Z*-algebra with standard carrier as defined in Definition 2.*
2. S *is a set of states. Associated with each state* $s \in S$ *are*

   (a) *a* valuation $v^s : \bigcup_{z \in Z} X_z \to \mathcal{A}_z^+$ *and an* attribute evaluation $\mathcal{O}^s$ *defining for each tuple-structured sort* $z' \in T$ *with* $\eta(z') = \langle \ldots, A : z, \ldots \rangle$ *an evaluation function* $A_{\mathcal{O}^s} : \mathcal{A}_{z'} \to \mathcal{A}_z^+$.
   The extension of the variable evaluation v^s *to the valuation* V^s *of the algebraic expressions is given in the Appendix.*

In this paper we do not treat typing issues. We presuppose that the image space is always $\mathcal{A}_z^+$. *In a companion paper we have developed the strong typing framework that ensures type safety in all (reachable) states.*

   (b) *a function* $c_z^s : \mathcal{A}_z \to \{\mathbf{1}, \mathbf{0}\}$ *for each* $z \in T$
   where the set $\{a | c_z^s(a) = \mathbf{1}\}$ *is finite for all* $z \in T$ *and all* $s \in S$. *The function* c_z^s *models the extension of type* z, *i.e., the class membership.*

   (c) *a function* $used^s : \mathcal{A}_{ANY} \to \{\mathbf{1}, \mathbf{0}\}$
   where the set $\{a | used^s(a) = \mathbf{1}\}$ *is finite for all* $s \in S$. *This function gathers all object identifiers that have ever been used (or are currently in use).*

(d) for each $z \in T$ *with* $\eta(z) = \{z'\} \in SS \cup RS$ *a function* set_z^s : $\mathcal{A}_z, \mathcal{A}_{z'} \to \{\mathbf{1}, \mathbf{0}\}$
where the set $\{b | set_z^s(a,b) = \mathbf{1}\}$ *is finite for all* $z \in T$, $a \in \mathcal{A}_z$, *and* $s \in S$. *The function* set_z^s *manages the set memberships.*

3. $[\![\cdot]\!]$ *associates with each command* $\alpha \in Cmd(L)$ *a relation* $[\![\alpha]\!] \subseteq S \times S$. *The relation* $[\![\alpha]\!]$ *specifies the state transformation that is caused by the command* α.

The valuation of variables and the evaluation of the attributes is demanded to be well-behaved with respect to typing. For a more elaborate discussion on typing issues see [18]. The function c_z^s is used to indicate that a certain object identifier is contained in a particular class, i.e., in an extension of a type z. Additionally, we use the function $used^s$ where identifiers of objects including those which have been deleted are collected. This is necessary in order to avoid unwanted effects which arise when an object is deleted, some reference to it still exists, and another newly created object receives this object identifier. An alternative would have been to delete all references to this object on its deletion—which is, however, prohibitively expensive in a "real" implementation.

3.2 Standard Model

In this subsection we specify the state transitions involved by commands and we present the satisfiability of the formulae. Some useful relations between states for defining the semantics of the standard model are described in Figure 4. A more elaborate specification of these relations is given in the Appendix. We will give some remarks about them. The relation $s(x/a)t$ specifies the semantics of assignments to variables while the relation $s(A(a)/a')t$ will be utilized to describe the state transition induced by attribute assignment. The notation $s(x/add(a))t$ plays a major role in specifying the state transitions due to object creation. A (new) *OID* will be assigned to the variable x. The relation ensures that no attribute references from x, and no elements of x, respectively, exists. Analogously, the next terminology $s(del(a))t$ is relied upon when specifying the state transition due to deleting an object. The last relation $s(ins(a,a'))t$ is used to specify the state transitions to be

relation	v	$\mathcal{O}$	c_z	set_z
$s(x/a)t$	$v^t(x) = a$	$=$	$=$	$=$
$s(A(a)/a')t$	$=$	$A_{\mathcal{O}^t}(a) = a'$	$=$	$=$
$s(x/add(a))t$	$v^t(x) = a$	$(A_i)_{\mathcal{O}^t}(a) = \omega$ $(\eta(z) \in TS, a \in \mathcal{A}_z)$	$c_z^t(a) = \mathbf{1}$ $(a \in \mathcal{A}_z)$	$set_z^t(a, a') = \mathbf{0}$ $(\eta(z) = \{z'\},$ $a \in \mathcal{A}_z, a' \in \mathcal{A}_{z'})$
$s(del(a))t$	$=$	$(A_i)_{\mathcal{O}^t}(a) = \omega$ $(\eta(z) \in TS, a \in \mathcal{A}_z)$	$c_z^t(a) = \mathbf{0}$ $(a \in \mathcal{A}_z)$	$set_z^t(a, a') = \mathbf{0}$ $(\eta(z) = \{z'\},$ $a \in \mathcal{A}_z, a' \in \mathcal{A}_{z'})$
$s(ins(a, a'))t$	$=$	$=$	$=$	$set_z^t(a, a') = \mathbf{1}$ $(\eta(z) = \{z'\},$ $a \in \mathcal{A}_z, a' \in \mathcal{A}_{z'})$

Due to space limitations, one column *used* is not shown in the table above. It has only one entry in row $s(x/add(a))t$: $used^t(a) = \mathbf{1}$.

Figure 4: Some Relations between States

undertaken when an object—denoted by a'—is inserted into a set-structured object—denoted by a.

Now, we can define our standard model which is responsible for the state transitions.

Definition 9 (Standard Model) *A model $\mathcal{M} = (\mathcal{A}, S, [\![\cdot]\!])$ as defined in Definition 8 is called* standard *iff it satisfies the following conditions:*

1. $[\![x := v]\!]$ *conditions:*
   (a) *if $V^s(v) \neq \omega$ then there exists $t \in S$ such that $s[\![x := v]\!]t$, and*
   (b) *if $s[\![x := v]\!]t$ then $V^s(v) \neq \omega$ and $s(x/V^s(v))t$*

   Note, it is illegal to assign w to a variable.
2. $[\![v.A := v']\!]$ *conditions:*

(a) *if* $V^s(v) \neq \omega$, $V^s(v') \neq \omega$, *and* $c^s_{ANY}(V^s(v)) = \mathbf{1}$ *then there exists* $t \in S$ *such that* $s[\![v.A := v']\!]t$, *and*

(b) *if* $s[\![v.A := v']\!]t$ *then* $V^s(v) \neq \omega$, $V^s(v') \neq \omega$, $c^s_{ANY}(V^s(v)) = \mathbf{1}$, *and* $s(A(V^s(v))/V^s(v'))t$

3. $[\![\alpha;\beta]\!] = [\![\alpha]\!] \circ [\![\beta]\!]$; *two relations* P *and* R *are composed by* $P \circ R := \{(s,t) \mid$ *if there exists* u *such that* $(s,u) \in P$ *and* $(u,t) \in R\}$

4. $[\![if(\epsilon)(\alpha)]\!] = \{(s,t) \mid V^s(\epsilon) = \mathbf{1}$ *and* $s[\![\alpha]\!]t\} \cup \{(s,s) \mid V^s(\epsilon) = \mathbf{0}\}$
Note, if $V^s(\epsilon) = \omega$ *then the relation equals to* $\emptyset$.

5. $s[\![foreach(x \text{ in } v)(\alpha)]\!]t$ *iff* $V^s(v) \neq \omega$, $V^s(v) = \{a_1, \ldots, a_n\}$, *and there exists* $s = s_0, \ldots, s_{2n} = t \in S$ *such that for all* $0 \leq i \leq n-1$:

(a) $s_{2i}(x/a_{i+1})s_{2i+1}$ *and*

(b) $s_{2i+1}[\![\alpha]\!]s_{2i+2}$

The enumeration of the elements of $V^s(v)$ *is not fixed, since we have sets in the mathematical sense. However, after evaluation of* v *we have determined an arbitrary order of the elements.*

6. $[\![x_z.create]\!]$ *conditions:*

(a) *for each* $s \in S$ *exists* $t \in S$ *and* $id \in \mathcal{A}_{z!}$ *with* $used^s(id) = \mathbf{0}$ *such that* $s[\![x_z.create]\!]t$, *and*

(b) *if* $s[\![x_z.create]\!]t$ *then there exists* $id \in \mathcal{A}_{z!}$ *with* $used^s(id) = \mathbf{0}$ *and* $s(x_z/add(id))t$

7. $[\![x_z.delete]\!]$ *conditions:*

(a) *if* $V^s(x_z) \neq \omega$ *then there exists* $t \in S$ *such that* $s[\![x_z.delete]\!]t$, *and*

(b) *if* $s[\![x_z.delete]\!]t$ *then* $V^s(x_z) \neq \omega$, $V^s(x_z) = id$, *and* $s(del(id))t$

8. $[\![x.insert(v)]\!]$ *conditions:*

(a) *if* $V^s(x) \neq \omega$ *and* $V^s(v) \neq \omega$ *then there exists a state* $t \in S$ *such that* $s[\![x.insert(v)]\!]t$, *and*

(b) *if* $s[\![x.insert(v)]\!]t$ *then* $V^s(x) \neq \omega$, $V^s(v) \neq \omega$, *and* $s(ins(V^s(x), V^s(v)))t$

Note, an insertion of an existing element into a set is legal.

We have omitted the semantics of the *remove* command.

For proving the correctness of state transitions we have to extend our formalism. The extension allows us to describe that a state s of a given model $\mathcal{M}$ fulfills a condition denoted by a formula Φ. We can say that a formula Φ *holds in* $\mathcal{M}$ at state $s \in S$, denoted by $\mathcal{M} \models_s \Phi$. In the following definition we give such a formalism enriched with some rules of implication and quantification.

Definition 10 (Satisfiability) *Let $\Phi \in Fma(L)$ and $\mathcal{M}$ be a model as defined in Definition 7 and Definition 8, respectively. We then define $\mathcal{M} \models_s \Phi$ inductively as follows:*

1. *If $\Phi \in Bxp(L)$ then $\mathcal{M} \models_s \Phi$ iff $V^s(\Phi) = \mathbf{1}$.*
2. *$\mathcal{M} \models_s \Phi \Longrightarrow \Psi$ iff $\mathcal{M} \models_s \Phi$ implies $\mathcal{M} \models_s \Psi$.*
3. *$\mathcal{M} \models_s [\alpha]\Phi$ iff for all $t \in S$ such that $s[\![\alpha]\!]t$, $\mathcal{M} \models_t \Phi$.*
4. *$\mathcal{M} \models_s \forall x_z \Phi$ iff for all $a \in \mathcal{A}_z$, if $s(x_z/a)t$ then $\mathcal{M} \models_t \Phi$.*

We have defined an object-oriented model in a logical manner. We have used dynamic logic as been axiomatised by [12]. We distinguish between a value and an object part. Therefore, we need an algebra and a state-based approach for specifying the semantics—the models of our formal model. Now, we have a well-defined (formal) model which allows us to derive some properties about object-oriented database programming.

4 Formalized Features

For illustration we present three special features of our formal model. First, the model allows us to show non-termination of a sample program. Second, the termination of another program assumed we verify some facts about the object base. Last, we integrate queries into the algebraic expressions. The formal model shows how the evaluation of queries works. Thus, query transformations will be proven to be correct.

4.1 Dangling References

Deleting objects may arise dangling references. Since in our formal model references to objects—as attribute or as set membership—are uni-directional it is very expensive to remove all references to a given object. Therefore, the references to an object may survive the object itself. We will formulate a program fragment which enforce a dangling reference.

$$
\begin{array}{ll}
x_{EmpType}.create;\ x_{DepType}.create;\ x_{MgrType}.create; & !! := \alpha_1 \\
x_{EmpType}.WorksIn := x_{DepType};\ x_{DepType}.Mgr := x_{MgrType}; & !! := \alpha_2 \\
x_{DepType}.delete; & !! := \alpha_3 \\
x_{MgrType} := x_{EmpType}.WorksIn.Mgr & !! := \alpha_4
\end{array}
$$

Our proof mechanism shows that the program does not terminate. We are able to detect non-termination.

Sketch of Proof Let M be any standard model and $s \in S$ be an appropriate state. We want to show $M \models_s [\![\alpha_1;\alpha_2;\alpha_3;\alpha_4]\!]false$ which is equivalent to the implication for all $t \in S$ $s[\![\alpha_1;\alpha_2;\alpha_3;\alpha_4]\!]t$ *implies* $M \models_t false$. We have to derive the formula *not exists* $t \in S$ $s[\![\alpha_1;\alpha_2;\alpha_3;\alpha_4]\!]t$.

We assume that $s_1, s_2, s_3 \in S$ exist such that $s[\![\alpha_1]\!]s_1$, $s_1[\![\alpha_2]\!]s_2$, $s_2[\![\alpha_3]\!]s_3$, and $s_3[\![\alpha_4]\!]t$ hold. We can derive:

1. $V^{s_1}(x_{EmpType}) = emp_1$, $V^{s_1}(x_{DepType}) = dep_1$, $V^{s_1}(x_{MgrType}) = mgr_1$
   The *OIDs* emp_1, dep_1, and mgr_1 are chosen arbitrarily.

2. $WorksIn_{\mathcal{O}^{s_2}}(emp_1) = dep_1$, $Mgr_{\mathcal{O}^{s_2}}(dep_1) = mgr_1$

3. $c^{s_3}_{DepType}(dep_1) = \mathbf{0}$, $Mgr_{\mathcal{O}^{s_3}}(dep_1) = \omega$

4. For the last state transition from s_3 to t the precondition fails (see Definition 9): $Mgr_{\mathcal{O}^{s_3}}(WorksIn_{\mathcal{O}^{s_3}}(V^{s_3}(x_{EmpType}))) = Mgr_{\mathcal{O}^{s_3}}(dep_1) = \omega$. The right side of the assignment command is not defined.

The last state transition fails and therefore, the program does not terminate. □

4.2 A Step to Program Verification

We want to show that after termination of the following program each manager earns at least as much money as her/his backups.

$$
\begin{array}{l}
\mathit{foreach}(x\ \mathit{in}\ \mathit{ext}(\mathit{MgrType})) \\
\quad (\mathit{foreach}(y\ \mathit{in}\ \kappa(x.\mathit{Backups})) \\
\qquad \mathit{if}(y.\mathit{Salary} > x.\mathit{Salary})(x.\mathit{Salary} := y.\mathit{Salary})\ldots) \qquad !! := \alpha
\end{array}
$$

Sketch of Proof Let M be any standard model and $s, t \in S$ be two states with $s[\![\mathit{foreach}(\ldots)]\!]t$. Let $a \in \mathcal{A}_{MgrType}$ and $b \in \mathcal{A}_{Emp\,Type}$ be two *OIDs* with $c^s_{MgrType}(a) = \mathbf{1}$ and $set^s_{EmpSetType}(Backups_{\mathcal{O}^s}(a), b) = \mathbf{1}$. We divide the state transition from s to t into $s, \ldots, s_1, s_2, \ldots, t$ with $s_1[\![\alpha]\!]s_2$, $V^{s_1}(x) = a$, and $V^{s_1}(y) = b$. The condition $V^{s_1}(y.\mathit{Salary} > x.\mathit{Salary}) = \omega$ implies that s_2 does not exist. The condition $V^{s_1}(y.\mathit{Salary} > x.\mathit{Salary}) = \mathbf{1}$ implies $s_1[\![x.\mathit{Salary} := y.\mathit{Salary}]\!]s_2$. Hence, we can derive $\mathit{Salary}_{\mathcal{O}^{s_2}}(a) \geq \mathit{Salary}_{\mathcal{O}^{s_2}}(b)$ (if a state s_2 and a state t, respectively, exist.) In a state t the three properties $c^t_{MgrType}(a) = \mathbf{1}$, $set^t_{EmpSetType}(Backups_{\mathcal{O}^t}(a), b) = \mathbf{1}$, and $\mathit{Salary}_{\mathcal{O}^t}(a) \geq \mathit{Salary}_{\mathcal{O}^t}(b)$ hold as well.
Therefore, for all $a \in \mathcal{A}_{MgrType}$ and for all $b \in \mathcal{A}_{Emp\,Type}$ the condition $c^t_{MgrType}(a) = \mathbf{1}$ is a prerequisite for the implication that the set membership $set^t_{EmpSetType}(Backups_{\mathcal{O}^t}(a), b) = \mathbf{1}$ implies $\mathit{Salary}_{\mathcal{O}^t}(a) \geq \mathit{Salary}_{\mathcal{O}^t}(b)$. Hence, we can equivalently assume that for all $a \in \mathcal{A}_{MgrType}$, $b \in \mathcal{A}_{Emp\,Type}$ two states t', t'' exist with $t(x_{MgrType}/a)t'$ and $t'(y_{Emp\,Type}/b)t''$ such that $(\mathcal{M} \models_{t''} (x_{MgrType}\ \mathit{in}\ \mathit{ext}(\mathit{MgrType}))$ implies $(\mathcal{M} \models_{t''} (y_{Emp\,Type}\ \mathit{in}\ \kappa(x_{MgrType}.\mathit{Backups}))$ implies the formulae $\mathcal{M} \models_{t''} (x_{MgrType}.\mathit{Salary} \geq y_{Emp\,Type}.\mathit{Salary})))$. This is equivalent to the following formula:

$$
\begin{array}{l}
\mathcal{M} \models_t \forall x_{MgrType} \forall y_{Emp\,Type} (x_{MgrType}\ \mathit{in}\ \mathit{ext}(\mathit{MgrType})) \Longrightarrow \\
((y_{Emp\,Type}\ \mathit{in}\ \kappa(x_{MgrType}.\mathit{Backups})) \Longrightarrow (x_{MgrType}.\mathit{Salary} \geq y_{Emp\,Type}.\mathit{Salary}))
\end{array}
$$

Concluding, if a state t with $s[\![\mathit{foreach}(\ldots)]\!]t$ exists then t fulfills the demanded property. □

4.3 Query Evaluation

As a query in QUEL, a GOMql query [16] consists of several clauses. The *range* clause defines ranges—extensions of types—to which the local variables

of the query are bound. The *retrieve* clause specifies the structure of the outcome of the query while the *where* clause specifies the selection predicate that has to be satisfied by the actual bindings in order to be included in the result. We introduce an example query q_{exam} before defining queries as algebraic expressions.

Example "*Retrieve the managers of departments which generate losses and, at the same time, pay at least one of their employees an exorbitant salary exceeding 200K!*"

$$\begin{array}{lll} q_{exam} := & \mathit{range} & e : \mathit{EmpType}, m : \mathit{MgrType} \\ & \mathit{retrieve} & \mathit{Mgr} : m \\ & \mathit{where} & (\ (m == e.\mathit{WorksIn}.\mathit{Mgr})\ \mathit{and} \\ & & \quad (e.\mathit{WorksIn}.\mathit{Profit} < 0)\ \mathit{and} \\ & & \quad (e.\mathit{Salary} > 200000)) \end{array}$$

In this example we have introduced two variables e and m. They are typed by *EmpType* and *MgrType*, respectively. We have bound each variable to a set summarizing all objects existing in the object base of the respective type. The selection predicate is given by a boolean expression. All objects of type *MgrType* which fulfill the selection predicate form the set of result. For further transformations the set of result is represented by tuples with one attribute *Mgr*.

Corresponding to the example we define a query language as follows.

Definition 11 (Query Language) *For $1 \leq i \leq n$ let $z_i \in SS \cup RS$ with $\eta(z_i) = \{z_i'\}$ and $x_i \in X_{z_i'}$; for $1 \leq j \leq m$ let $z_j'' \in BS$, $\tau_j \in Axp_{z_j''}(L)$ and $A_j \in Attr$; and further, let $\epsilon \in Bxp(L)$. Then, a query q has the following form:*

$$\begin{array}{lll} q := & \mathit{range} & x_1 : z_1, \ldots, x_n : z_n \\ & \mathit{retrieve} & A_1 : \tau_1, \ldots, A_m : \tau_m \\ & \mathit{where} & (\epsilon) \end{array}$$

We include q into $Axp_{\{\langle A_1 : z_1'', \ldots, A_m : z_m'' \rangle\}}(L)$.

We included q in $Axp(L)$; therefore queries can be utilized in commands, e.g., in *foreach* expressions. Further, we allow for nested queries.

Since the query is (entirely) based on extensions and returns a relation, the knowledge of relational query processing and optimization can be utilized. But still, some object-oriented specialities must be considered. First, access structures especially developed for object-oriented databases (e.g., [15, 17]) must be taken into account. Second, the cost model for estimating the overhead of different evaluation alternatives must be adapted to the particularities of the object-oriented model. Furthermore, since object-oriented data models have more expressive power than the relationals the optimization is much more than (simple) transformation of relations.

We now give the semantics of the query q by expanding the valuation V^s to q.

Definition 12 (Query Valuation) *A model $\mathcal{M}$ is given; for each $s \in S$ the valuation of a query q as in Definition 11 is defined as follows:*

$$V^s(q) := \begin{cases} \omega & \text{if there is } z_i \text{ with } V^s(\mathrm{ext}(z_i)) = \omega \\ \omega & \text{if there are } b_1 \in \mathrm{ext}(z_1), \ldots, b_n \in \mathrm{ext}(z_n) \text{ with} \\ & V^s(\epsilon[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n]) = \omega \\ \omega & \text{if there are } b_1 \in \mathrm{ext}(z_1), \ldots, b_n \in \mathrm{ext}(z_n) \text{ with} \\ & V^s(\epsilon[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n]) = \mathbf{1}, \text{ and} \\ & V^s(\tau_j[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n]) = \omega \\ res(q) & \text{else} \end{cases}$$

$$\begin{aligned} \text{with } res(q) := \; & \{[A_1 : a_1, \ldots, A_m : a_m] \mid \text{ there are} \\ & b_1 \in \mathrm{ext}(z_1), \ldots, b_n \in \mathrm{ext}(z_n) \text{ with} \\ & V^s(\epsilon[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n]) = \mathbf{1}, \text{and} \\ & \text{for } 1 \leq j \leq m \; a_j = V^s(\tau_j[x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n])\} \end{aligned}$$

We have used the notation of substitution, e.g. $\epsilon[x \leftarrow b]$, as defined in the Appendix.

We will now give a nested loop implementation to evaluate the query as defined in Definition 11. Let $m := 1$ (single target) and y_t with $\eta(t) = \mathbf{\{}\langle A_1 : z_1 \rangle\mathbf{\}}$ be a variable not occurring in q. Then, the nested loop implementation

of a (single target) query q is defined as follows.

$$\begin{aligned} nested(q) := \; & y_t.create; \\ & foreach\ (x_1\ in\ \mathrm{ext}(z_1))(\\ & \quad \ddots \\ & \quad foreach\ (x_n\ in\ \mathrm{ext}(z_n))(\\ & \quad\quad if\ (\epsilon)(y_t.insert([A_1 : \tau_1])) \ldots) \end{aligned}$$

The following theorem shows the equivalence of the nested loop evaluation.

Theorem 13 (Nested Loop Evaluation)
For each standard model $\mathcal{M} = (\mathcal{A}, S, [\![\cdot]\!])$ *and state* $s \in S$*:*

$$\mathcal{M} \models_s [nested(q)](\kappa(y_t) == q)$$

□

The interpretation of Theorem 13 is as follows: "*after termination of the program* $nested(q)$ *the result of the valuation of* $\kappa(y_t)$ *is the same as the one of* q*.*"

Sketch of Proof The proof of Theorem 13 is composed of several lemmata. First, we give a lemma for initialization. It provides for a well-known starting state ($V^{s_1}(\kappa(y_t)) = \emptyset$). Second, there are two lemmata securing loop independence. In each loop i there is a set of variables $\{x_1, \ldots, x_{i-1}\}$ and a set of algebraic expressions $\{v_1, \ldots, v_n\}$ ($v_i = ext(z_i)$) which never change. Then, we subdivide the nested loop evaluation into some (single) state transitions $s_{(1:0,\ldots,n:0,n+1:0)}, \ldots, s_{(1:k_1,\ldots,n:k_n,n+1:1)}$. We can thus show how we get the set of result step by step, state by state. There are still three lemmata necessary. The first one shows in which state the variables have values we are interested in. The second one propagates monotony; it proves that we do not loose a tuple of result we have found so far. The last lemma secures the right insertion of tuples in the set of result. Then, it is easy to complete the proof. □

5 Conclusion and Future Research

In this paper we presented a formal framework for specifying the semantics of an object-oriented data model. The formalization is based on *dynamic*

logic which provides the necessary mechanisms for handling both aspects of object-orientation: the *structural* as well as the *behavioral* dimension—the latter being modeled unsatisfactorily in most other approaches. The behavioral component is captured by specifying state transitions in response to object modifications. In our framework we distinguish between the *value* part (comprising the relational model) and the *object* part (dealing with the dynamic behavior). The value part is modeled—in a standard manner—by many-sorted algebras.

This work should be seen as a step towards a thorough formalization of the object-oriented data model(s). We provided the stable basis within which further investigations can be carried out. In this paper we outlined three sample applications of our formal model: non-termination by dangling references, (partial) program verification, and specifying the semantics of an object-oriented query language. Many other object-oriented features remain to be formally specified. Among these are typing concepts that ensure type consistency of the object base states and schema evolution—in our current framework we disallow any schema updates. Finally, a proof theory should be developed for our formal model that could—ultimately—lead to automatic program verification (support).

References

[1] S. Abiteboul. Updates, a new frontier. In M. Gyssens, J. Paredaens, and D. VanGucht, editors, *Proc. of the Intl. Conf. on Database Theory (ICDT)*, volume 326 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, Aug 1988.

[2] S. Abiteboul. Towards a deductive object-oriented database language. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 419–438, Kyoto, Japan, Dec 1989.

[3] S. Abiteboul, C. Beeri, M. Gyssens, and D. Van Gucht. An introduction to the completeness of languages for complex objects and nested relations. In S. Abiteboul, P.C. Fischer, and H.-J. Schek, editors, *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science (LNCS)*, pages 117–138. Springer Verlag, 1987.

[4] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. In *Proc. Intl. Conf. Extending Database Technology (EDBT)*, pages 271–293, 1988.

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

[6] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 40–57, Kyoto, Japan, Dec 1989.

[7] F. Bancilhon. A logic-programming/object-oriented cocktail. *SIGMOD RECORD*, 15(3):11–21, 1986.

[8] F. Bancilhon and K. Koshafian. A calculus for complex objects. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 53–59, 1986.

[9] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. of the Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 370–395, Kyoto, Japan, Dec 1989.

[10] C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic database language (LDL1). In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 21–37, Austin, TX, 1987.

[11] F. Cacace, S. Ceri, et al. Integrating object-oriented data modelling with a rule-based programming paradigm. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 225–236, 1990.

[12] R. Goldblatt. *Axiomatising the Logic of Computer Programming*, volume 130 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag, 1982.

[13] G. Gottlob, G. Kappel, and M. Schrefl. Semantics of object-oriented data models - the evolving algebra approach. In J.W. Schmidt and A.A. Stogny, editors, *First International East/West Database Workshop*, number 504 in Lecture Notes in Computer Science (LNCS), pages 144–160, Kie, UDSSR, Oct 1990. Springer-Verlag.

[14] R. Hull and J. Su. On accessing object-oriented databases: Expressive power, complexity, and restrictions. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 147–158, 1989.

[15] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–268, Denver, USA, May 1991.

[16] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, Aug 1990.

[17] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[18] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. der GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik-Fachberichte Nr. 270.

[19] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. of the ACM Conf. on Object-Oriented Programming Languages and Systems (OOPSLA)*, pages 408–416, Nov 1986.

[20] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 134–146, Portland, OR, 1989.

[21] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 379–393, 1989.

[22] G. Kuper. Logic programming sets. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 11–20, Austin, TX, 1987.

[23] C. Lécluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. Intl. Conf. Extending Database Technology (EDBT)*, pages 556–562, 1988.

[24] D. Maier. Why isn't there an object-oriented data model. Technical report, Oregan Graduate Center, 1989.

[25] H.-J. Schek and M. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, 1986.

[26] G. M. Shaw and S. B. Zdonik. A query algebra for object-oriented databases. In *Proc. IEEE Conf. on Data Engineering*, pages 154–162, L.A., CA, Feb 1990.

[27] J. D. Ullman. Database theory: Past and future. In *Proc. ACM Symp. on Principles of Database Systems*, pages 1–10, 1987.

[28] S. L. Vandenberg and D. J. DeWitt. Algebraic support for complex objects with arrays, identity, and inheritance. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 158–167, Denver, CO, May 1991.

[29] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–32. Morgan-Kaufman Publ. Co., 1989.

Appendix

Substitution A substitution replaces an expression by another expression. For two expressions x and α the substitution $[x \leftarrow \alpha]$ describes the replacement of x by α. If this substitution is applied to an expression v (written as $v[x \leftarrow \alpha]$) every occurrence of x in v is replaced by α.

Valuation Let $\mathcal{M}$ be a model as defined in Definition 8. For each $s \in S$ the canonical extension of the evaluation v^s to the valuation V^s is defined as follows:

1. *if* $v \in \mathcal{A}_z$ *then* $V^s(v) := v$
2. *if* $v \in X_z$ *then* $V^s(v) := v^s(v)$
3. *if* $v \in Pxp(L)$ *then* $V^s(v.A) := \begin{cases} \omega & \text{if } V^s(v) = \omega \\ A_{\mathcal{O}^s}(V^s(v)) & \text{else} \end{cases}$
4. $V^s(false) := \mathbf{0}$
5. *for* $rel \in \{==, <, >, in\}$
   $V^s(v \ rel \ \tau) := \begin{cases} \omega & \text{if } V^s(v) = \omega \text{ or } V^s(\tau) = \omega \\ V^s(v) \ rel \ V^s(\tau) & \text{else} \end{cases}$
6. $V^s(ext(z)) := \{a | c_z^s(a) = \mathbf{1}\}$
7. $V^s([A : v]) := \begin{cases} \omega & \text{if } V^s(v) = \omega \\ [A : V^s(v)] & \text{else} \end{cases}$
8. *if* $v \in Axp_{\langle \ldots, A:z, \ldots \rangle}(L)$ *then* $V^s(v.A) := \begin{cases} \omega & \text{if } V^s(v) = \omega \\ a & \text{if } V^s(v) = [\ldots, A : a, \ldots] \end{cases}$
9. $V^s(v \circ v') := \begin{cases} \omega & \text{if } V^s(v) = \omega \text{ or } V^s(v') = \omega \\ \omega & \text{if } v \in Axp_{\langle A_1:z_1, \ldots, A_n:z_n \rangle}(L), v' \in Axp_{\langle A'_1:z'_1, \ldots, A'_m:z'_m \rangle}, \\ & \quad \text{and } A_i = A'_j \text{ for some } i, j \\ V^s(v) \circ V^s(v') & \text{else} \end{cases}$
10. $V^s(\{v\}) := \begin{cases} \omega & \text{if } V^s(v) = \omega \\ \{V^s(v)\} & \text{else} \end{cases}$
11. $V^s(fetch(v)) := \begin{cases} \omega & \text{if } V^s(v) = \omega \\ \omega & \text{if } V^s(v) = \emptyset \\ a_i & \text{if } V^s(v) = \{a_1, \ldots, a_i, \ldots, a_n\} \text{ with } n \in \mathbf{N} \end{cases}$

    The definition is indeterminate if $V^s(v)$ *denotes a set with more than one element.*
12. $V^s(v_1 union \ v_2) := \begin{cases} \omega & \text{if } V^s(v_1) = \omega \text{ or } V^s(v_2) = \omega \\ V^s(v_1) \cup V^s(v_2) & \text{else} \end{cases}$

13. $V^s(v_1 setminus\ v_2) := \begin{cases} \omega \quad if\ V^s(v_1) = \omega\ or\ V^s(v_2) = \omega \\ V^s(v_1) \setminus V^s(v_2) \quad else \end{cases}$

14. $V^s(\sigma_{\lambda x(\epsilon)}(v)) := \begin{cases} \omega\ if\ V^s(v) = \omega \\ \omega\ if\ V^s(\epsilon[x \leftarrow a]) = \omega\ for\ some\ a \in V^s(v) \\ \{a | a \in V^s(v), V^s(\epsilon[x \leftarrow a]) = \mathbf{1}\}\ else \end{cases}$

15. $V^s(\pi_{\lambda x(\tau)}(v)) := \begin{cases} \omega\ if\ V^s(v) = \omega \\ \omega\ if\ V^s(\tau[x \leftarrow a]) = \omega\ for\ some\ a \in V^s(v) \\ \{V^s(\tau[x \leftarrow a]) | a \in V^s(v)\}\ else \end{cases}$

16. $V^s(v_1 \bowtie_{\lambda x(\epsilon)} v_2) := \begin{cases} \omega\ if\ V^s(v_1) = \omega\ or\ V^s(v_2) = \omega \\ \omega\ if\ V^s(\epsilon[x \leftarrow a \circ b]) = \omega\ for\ some\ a \in V^s(v_1),\ and \\ \quad b \in V^s(v_2) \\ \{a \circ b | a \in V^s(v_1), b \in V^s(v_2), V^s(\epsilon[x \leftarrow a \circ b]) = \mathbf{1}\} \\ \quad else \end{cases}$

17. $V^s(\kappa(v)) := \begin{cases} \omega \quad if\ V^s(v) = \omega \\ \omega \quad if\ v \in Axp_z(L)\ with\ \eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle, and \\ \quad V^s(v.A_i) = \omega\ for\ some\ 1 \le i \le n \\ [A_1 : V^s(v.A_1), \ldots, A_n : V^s(v.A_n)]\ if \\ \quad \eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle\ (v \in Axp_z(L)), and \\ \quad V^s(v.A_i) \neq \omega\ for\ all\ 1 \le i \le n \\ \{a | set^s_z(V^s(v), a) = \mathbf{1}\} \quad else \end{cases}$

Some Relations between States First, we give some boolean operators for comparing almost identical functions.

If g and h are functions defined over the same domain and almost equal we use the following abbreviations:

- if g and h are functions with the same domain G, then we write

$$g =_b h\ \ iff\ \ forall\ x \in G\ s.t.\ x \neq b\ implies\ g(x) = h(x)$$

$$g =_{\{b_1,\ldots,b_n\}} h\ \ iff\ \ forall\ x \in G\ s.t.\ x \notin \{b_1, \ldots, b_n\}\ implies\ g(x) = h(x)$$

- if g and h are functions with the same domain $G_1 \times G_2$, then we write

$$g =_{a,b} h \quad iff \quad forall\ x_1 \in G_1\ and\ forall\ x_2 \in G_2\ s.t.\ (x_1 \neq a\ or\ x_2 \neq b)\ implies\ g(x_1, x_2) = h(x_1, x_2)$$

$$g =_{a,_} h \quad iff \quad forall\ x_1 \in G_1\ and\ forall\ x_2 \in G_2\ s.t.\ x_1 \neq a\ implies\ g(x_1, x_2) = h(x_1, x_2)$$

Second, we define the relations between states which have been sketched in Figure 4.

$s(x/a)t$	$v^s =_x v^t$	$v^t(x) = a$	$\mathcal{O}^s = \mathcal{O}^t$	$c^s_z = c^t_z$
	$used^s = used^t$	$set^s_z = set^t_z$		
$s(A(a)/a')t$	$v^s = v^t$	$\mathcal{O}^s =_A \mathcal{O}^t$	$A_{\mathcal{O}^s} =_a A_{\mathcal{O}^t}$	$A_{\mathcal{O}^t}(a) = a'$
	$c^s_z = c^t_z$	$used^s = used^t$	$set^s_z = set^t_z$	
$s(x/add(a))t$	$v^s =_x v^t$	$v^t(x) = a$	$used^s =_a used^t$	$used^t(a) = \mathbf{1}$
	for each $z \in T$ with $a \in \mathcal{A}_z$			
	$c^s_z =_a c^t_z$	$c^t_z(a) = \mathbf{1}$		
	if $\eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle \in TS$			
	then $\mathcal{O}^s =_{\{A_1,\ldots,A_n\}} \mathcal{O}^t, (A_i)_{\mathcal{O}^s} =_a (A_i)_{\mathcal{O}^t}, (A_i)_{\mathcal{O}^t}(a) = \omega$			
	if $\eta(z) = \{z'\} \in SS \cup RS$			
	then $set^s_z =_{a,-} set^t_z, set^t_z(a, a') = \mathbf{0}$ for all $a' \in \mathcal{A}_{z'}$			
	for each $z \in T$ with $a \notin \mathcal{A}_z$			
	$\mathcal{O}^s = \mathcal{O}^t$	$c^s_z = c^t_z$	$set^s_z = set^t_z$	
$s(del(a))t$	$v^s = v^t$	$used^s = used^t$		
	for each $z \in T$ with $a \in \mathcal{A}_z$			
	$c^s_z =_a c^t_z$	$c^t_z(a) = \mathbf{0}$		
	if $\eta(z) = \langle A_1 : z_1, \ldots, A_n : z_n \rangle \in TS$			
	then $\mathcal{O}^s =_{\{A_1,\ldots,A_n\}} \mathcal{O}^t, (A_i)_{\mathcal{O}^s} =_a (A_i)_{\mathcal{O}^t}, (A_i)_{\mathcal{O}^t}(a) = \omega$			
	if $\eta(z) = \{z'\} \in SS \cup RS$			
	then $set^s_z =_{a,-} set^t_z, set^t_z(a, a') = \mathbf{0}$ for all $a' \in \mathcal{A}_{z'}$			
	for each $z \in T$ with $a \notin \mathcal{A}_z$			
	$\mathcal{O}^s = \mathcal{O}^t$	$c^s_z = c^t_z$	$set^s_z = set^t_z$	
$s(ins(a, a'))t$	$v^s = v^t$	$\mathcal{O}^s = \mathcal{O}^t$	$c^s_z = c^t_z$	$used^s = used^t$
	if $z \in T$ with $\eta(z) = \{z'\}$, $a \in \mathcal{A}_z$, and $a' \in \mathcal{A}_{z'}$			
	then $set^s_z =_{a,a'} set^t_z, set^t_z(a, a') = \mathbf{1}$			
	else $set^s_z = set^t_z$			