Access to Objects by Path Expressions and Rules

Jürgen Frohn*
Fakultät für Mathematik und Informatik
Universität Mannheim
68131 Mannheim, Germany
frohn@pi3.informatik.uni-mannheim.de

Georg Lausen Heinz Uphoff*
Institut für Informatik
Universität Freiburg
79104 Freiburg, Germany
{lausen,uphoff}@informatik.uni-freiburg.de

Abstract

Object oriented databases provide rich structuring capabilities to organize the objects being relevant for a given application. Due to the possible complexity of object structures, path expressions have become accepted as a concise syntactical means to reference objects. Even though known approaches to path expressions provide quite elegant access to objects, there seems to be still a need for more generality. To this end, the rule-language PathLog is introduced. A first contribution of PathLog is to add a second dimension to path expressions in order to increase conciseness. In addition, a path expression can also be used to reference virtual objects. Both enhancements give rise to interesting semantic implications.

*Work supported by Deutsche Forschungsgemeinschaft, La 598/3-1.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

1 Introduction

Using the relational model we are forced to organize the application structures by a set of flat relations. Therefore, many applications demand for data models with richer structuring capabilities than the relational model. The missing concepts seem to be offered by the object oriented data model. Here, data is structured by means of *objects* which may have a *complex structure* and are assigned to *classes* which in turn are arranged hierarchically to offer an *inheritance mechanism.* Each object has a systemwide unique identifier, typically called *oid*, which is the basis of a reference-based access to the objects. Such references usually are obtained as the result of applying a method.

The complexity of the object structures finds its counterpart in the languages proposed to manipulate objects. To ease the task of accessing objects path expressions have been proposed. The idea here is to follow a link between objects without having to write down explicit join conditions.

This idea has appeared several times before. While one of the first approaches, GEM ([Zan83]), was based on QUEL, most approaches discuss possible extensions of SQL (e.g. OSQL [Fis87], ORION [Kim89], XSQL [KKS92], O_2SQL [BCD92] and ESQL [GV92]). To give a first flavor of path expressions let us go through some examples. For the time being we are interested in the color of the automobiles belonging to certain employees. We further assume, that a link between employees and their vehicles is established via a set-valued method (attribute) vehicles and that automobiles are a special kind of vehicles.

In O_2SQL we would write the following query:

```
SELECT Y.color
FROM X IN employee
FROM Y IN X.vehicles                    (1.1)
WHERE Y IN automobile
```

Here, the variables are ranging over objects; X.vehicles is a path expression which can be read as "apply method vehicles on object X". In general, a path may have arbitrary length.

XSQL contributes to this kind of languages the concept of selectors, which may be used to specify intermediate results in a path. Using selectors we can write more concisely:

```
SELECT Z
FROM employee X, automobile Y           (1.2)
WHERE X.vehicles[Y].color[Z]
```

In this example, the selectors [Y] and [Z] are used to restrict an intermediate result (vehicles have to be automobiles) and to provide a result-position for the query (the color is placed in Z).

A more calculus oriented proposal for path expressions is given in [VV93]. Here the usage of class names in a path is allowed making possible the following query:

```
{ Z | employee.vehicles.
                                        (1.3)
    automobile.color[Z]}
```

Even though the above approaches provide quite elegant techniques to access objects, we can observe certain limitations, as far as path expressions are concerned.

Path expressions in all languages we are aware of can only be applied in *one* dimension. Starting from a certain object, a composition of method applications can be specified, where each application, if the respective method is defined, references result objects. It would be nice, if we could also refer to other methods of such an object as part of the *same* path. For example, in XSQL, if we want to specify that the vehicles of interest should have 4 cylinders, to our knowledge, there is no way to express this in the same path. Instead, we have to break one path into two and in general, into many pieces, which leads to the following solution:

```
SELECT Z
FROM employee X, automobile Y
WHERE X.vehicles[Y].color[Z]            (1.4)
AND Y.cylinders[4]
```

What is missing is a *second* dimension which would allow us to refer to the properties of any object that is referenced in a path without having to leave that path. While the first dimension goes into depth, this second dimension would go into breadth.

Another way to increase the flexibility of object oriented models is to introduce *virtual* objects or classes ([AB91, KLW93]), which correspond to views in the relational model. While the technique used in XSQL builds on function symbols in a way proposed in [KW93], [AB91] propose a referencing technique based on methods (attributes). The latter approach seems to be more natural for path expressions; however no formal semantics of this approach has been presented. In the current paper we use methods to define and reference virtual objects and give formal semantics to this technique. Moreover, because methods can be controlled by signatures, virtual objects may be defined w.r.t. given type restrictions.

We propose a language called PathLog, which, on the one hand, gives interesting solutions to the problems mentioned above, and, on the other hand, extends the application area of path expressions to rule languages. The techniques we shall propose are applicable for different kinds of rule languages, e.g. deductive, production or active rules. This generality holds because path expressions are a convenient tool to *reference* objects; the way in which a set of rules is being evaluated is an orthogonal issue.

Despite the independence from certain evaluation paradigms, we discuss our techniques in a deductive framework. This provides us with a generally accepted terminology and a rigorous basis of semantics. Moreover, this decision is quite natural for us, because PathLog builds upon F-logic [KLW93]. PathLog extends the syntax of F-logic by path expressions and proposes a *direct* semantics for the enhanced syntax. As only a small subset of F-logic is relevant for the exposition of PathLog, the current paper still is self-contained.

The structure of the paper is as follows. We first present some characteristic features of PathLog (section 2). Next we introduce the terminology used throughout the paper (section 3). Syntax and semantics of PathLog follow in section 4 and section 5. Section 6 contains a discussion of interesting properties of PathLog. Section 7 finally gives a conclusion.

2 A First Look at PathLog

One striking characteristic of PathLog is its convenient concise syntax. We extend path expressions by a general means to specify properties of objects referenced within a path. For example, for each employee X, the path

```
X:employee[age→30; city→newYork]
  ..vehicles:automobiles[cylinders→4]          (2.1)
  .color[Z]
```

provides us with a reference to the colors of the vehicles of X, which are automobiles with 4 cylinders, if X is 30 years old and lives in newYork. If such a car indeed exists for employee X, variable Z will contain the corresponding color. As usual, variables are capitalized.

Note that in this kind of path expressions we can distinguish two dimensions. The first dimension is given by the composition of method-applications syntactically expressed by . (scalar methods) and .. (set-valued methods). The second dimension is given by a bracketed list of expressions in order to assert properties of the objects referred to inside a path; only those objects are referenced, which fulfill the specified properties.

Using a notation in the style of SQL, (2.1) becomes

```
SELECT Z
FROM employee X, automobile Y
WHERE X[age→30; city→newYork].                 (2.2)
  vehicles[cylinders→4][Y].color[Z]
```

The reader may have already noticed the similarity to *molecules* as they are used in F-logic. Here the question arises, how much PathLog does add to the known languages, if we abstract from syntax.

Two observations are worth to notice. On the one hand, in the setting of PathLog a path may be treated as a *reference* to objects. As a consequence of this first view, in PathLog a path may be used wherever we expect an object. Therefore, we can extend molecules by allowing path expressions also *inside* molecules. For example, in (2.2) we can replace [...→newYork] by

```
[...→X.boss.city],                              (2.3)
```

to indicate that we are only interested in the color of those vehicles, whose owner lives in the same city as the respective boss.

On the other hand, a path may be treated as a formula. In (2.2) a path was used inside the WHERE-clause and thus is assigned a truth-value. In fact, PathLog allows these two views under the same roof: a path may be treated as a reference and as a formula. Modifying (2.2) according to (2.3), the sub-path X.boss.city is treated as a reference while the whole path in the WHERE-clause corresponds to a formula.

To further demonstrate the impact of the second dimension in path expressions in PathLog, we discuss one more example. Consider the following O_2SQL query which asks for those managers X who have a red vehicle produced by a company located in Detroit where X itself is the president of that company.

```
SELECT X
FROM X IN manager
FROM Y IN X.vehicles
WHERE Y.color = red
   AND Y.producedBy.city = detroit
   AND Y.producedBy.president = X
```

This query in O_2SQL requires two FROM-clauses and a conjunctive WHERE-clause. The result of the set-valued path X.vehicles is treated as a class; hence the second FROM-clause is necessary to *flatten* this set of objects explicitly.

In PathLog, taking advantage of the possibility to mutually nest paths and molecules, we may combine scalar and set-valued paths in one reference. The above query may be expressed by a single reference:

```
X : manager..vehicles[color→red]
      .producedBy[city→detroit;president→X]
```

There is no necessity in PathLog to flatten the set of vehicles owned by a manager explicitly, since the methods color and producedBy are applied to each of the vehicles in turn.

We are not aware of any other language, which allows path expressions in a comparable generality. In O_2SQL a path can only be used as a one-dimensional reference. In XSQL a path can be used as a one-dimensional reference or formula, however semantics is only sketched by a transformation into F-logic, while we will give a *direct* semantics in this paper. In fact, this direct semantics of paths in PathLog gives rise to many interesting semantic implications.

Our direct semantics allows to use a path also to reference *virtual* objects. Adopting an example from [AB91], the following rule defines addresses as virtual objects for persons with given attributes city and street:

$$\begin{aligned}&\mathsf{X.address[city \rightarrow X.city;}\\ &\qquad \mathsf{street \rightarrow X.street]} \longleftarrow \mathsf{X : person.}\end{aligned} \qquad (2.4)$$

In this example, address-related attributes of persons are restructured into one new address object for each person. For each person X, X.address is used as a reference to the virtual address-object defined for X. Here we use *methods* (like address) to reference virtual objects; we do *not* need function symbols as in F-logic, or, with a similar aim, virtual class-names as in XSQL. Our approach has two benefits. First, our framework is simpler than it is in F-logic, because methods can do the same job function symbols had to do in that framework.[1] Second, methods can be controlled by signatures to make type checking techniques applicable.

3 Basic Terminology

For the purposes of this paper, objects are assumed to be distinguishable and are further described by their *state* and their *class-membership*. The state may be defined *extensionally*, i.e., by a given set of objects together with their (stored) attributes, or *intensionally*, by defining results of methods using rules. A *virtual* object in this setting is an object not given in the extensional part, but existing in the intensional part only.

On the language level there is no need to distinguish between extensional and intensional information, as may be seen in example (2.4), where one mechanism is sufficient to reference both the (intensional) address and the (extensional) city and street of a person. For this reason, we do not stress the difference between methods and attributes.

To simplify the framework, objects also denote classes and methods. Thus, the methods and classes used in the previous examples formally are objects; e.g. in (2.1), vehicles and automobile are names for objects which are used to denote a method and a class, respectively. As a direct consequence, class-membership reduces to a binary relation on objects. When objects denote methods, they may be overloaded according to their scalarity and arity. Thus, an object may be used to denote a scalar method as well as a set-valued one, and an object may be used to denote methods with different numbers of arguments.

Our simple setting can now be summarized as follows. Let $\mathcal{N}$ be a set of *names*. For simplicity, we don't distinguish between objects and values, thus $\mathcal{N}$ also includes integer numbers and strings. The alphabet of PathLog then consists of $\mathcal{N}$, a set of variables $\mathcal{V}$, auxiliary symbols, logical connectives and quantifiers. Formulas in PathLog, e.g. rules, are then defined as usual, the only difference here is that literals are built out of path expressions, which will be defined formally in section 4.

To define a formal semantics we need a *semantic structure* (cf. [KLW93]), I, which can be perceived as a set of objects with their properties. From a semantic structure I we can obtain all the needed information about a certain set of objects.[2] As usual, the set of all objects U is called the *universe*. Then, a semantic structure I is a tuple

$$I = (U, \in_U, I_{\mathcal{N}}, I_{\rightarrow}, I_{\twoheadrightarrow}).$$

Here, the function $I_{\mathcal{N}} : \mathcal{N} \mapsto U$ maps names to objects. The class hierarchy $\in_U \subseteq U \times U$ is a partial or-

[1] In fact, it is possible to replace the usual type constructors, e.g. cons by methods. A discussion of this aspect, however, is beyond the scope of the current paper.

[2] For the following, it is crucial to understand that a semantic structure covers extensionally given and intensionally defined aspects.

der telling us how objects are related to classes. $I_{\rightarrow}, I_{\twoheadrightarrow}$ interpret methods, i.e., define the state of the respective objects. $I_{\rightarrow}$ is a function which assigns to each element of U a partial function $U^k \overset{p}{\mapsto} U$, when this element is used as a scalar method with $k-1$, $k \geq 1$, arguments. $I_{\twoheadrightarrow}$ refers to set-valued methods and thus assigns a function $U^k \mapsto 2^U$ to each element of U, when this element is used as a set-valued method with $k-1$, $k \geq 1$, arguments; however, set-valued functions are total, slightly contrasting to [KLW93].[3]

4 Syntax of PathLog

In this section we will formally define the syntax of PathLog. We will introduce *paths* and *molecules*. Since paths as well as molecules are means to denote objects, they can be mutually nested in a very liberal way: in a molecule, wherever a (sub-) molecule is allowed, we can also use a path; in a path, wherever a (sub-) path is allowed, a molecule can be used. Therefore, both kinds of expressions are called *references*. References are distinguished according to their scalarity, i.e., they are either *set-valued* or *scalar*.

4.1 References to Objects

The most simple form of a reference are names and variables. Such simple references act as starting points for more complicated references. A path consists of a reference followed by a *method call*, e.g. .spouse, while a molecule consists of a reference followed by a *filter*, e.g. [boss→mary]. Note how paths and molecules may be mutually nested: a path

mary.spouse

is a reference and may therefore be used as part of the molecule

mary.spouse[boss→mary]

which in turn may again be used as part of the path

mary.spouse[boss→mary].age

[3]It turned out that we arrive at a rather opaque semantics if inside a path we have to distinguish between an undefined set-valued method and a method which returns the empty set.

to access the age of the object. It is also possible to nest terms inside a filter, e.g. the name mary may be further specified as in

mary.spouse[boss→mary[age →25]].

We first define rather general references; however, not all of them meet well-formedness as defined later.

Definition 1 Given an alphabet of PathLog, the set of all references can now be defined inductively as follows.

- A name $n \in \mathcal{N}$ and a variable $X \in \mathcal{V}$ is a *reference*, also called a *simple reference*.
- If t is a reference, then the expression (t) is a *reference*, also called a *simple reference*.
- If t_i $(0 \leq i \leq k)$, t'_j $(1 \leq j \leq l)$ and t_r are references, and if t_m, t_c are simple references,
  - then the expressions

    $$t_0.t_m@(t_1,\ldots,t_k),$$

    $$t_0..t_m@(t_1,\ldots,t_k)$$

    are *references*, also called *paths*.
  - then the expressions

    $$t_0[t_m@(t_1,\ldots,t_k)\rightarrow t_r],$$

    $$t_0[t_m@(t_1,\ldots,t_k)\twoheadrightarrow\{t'_1,\ldots,t'_l\}],$$

    $$t_0[t_m@(t_1,\ldots,t_k)\twoheadrightarrow t_r],$$

    $$t_0 : t_c$$

    are *references*, also called *molecules*. t_0 is called the *prefix* of the molecule and the expression $[t_m@\ldots]$, resp. $: t_c$, its *filter*. □

Methods may be called with parameters, e.g. john.salary@(1994), denoting john's salary in 1994. When methods are called without parameters, we will omit the parenthesis and the @-symbol, i.e., write mary.boss instead of mary.boss@(). In a sequence of filters, e.g. mary[age→30][boss→peter], all elements are applied to the first reference, which is mary in this case. To stress this fact we write as a shorthand mary[age→30;boss→peter], i.e., a reference with a *list* of filters is a molecule as well.

The XSQL-style of selectors e.g. in X..vehicles.color[Z] is used as an abbreviation for a filter specifying the built-in method self; the above example therefore is interpreted as X..vehicles.color[self→Z]. For every object the method self yields the object itself.

References surrounded by parenthesis are used to change the usual left-to-right evaluation sequence of a reference. To give an example, let kids be a method that yields the children of a person. Since kids itself is an object, we could therefore apply a method to this object. Here, we use a method transitiveClosure or tc with the following intention: applied to an object in the role of a method, e.g. kids, it yields a new method computing the transitive closure of that method, which is denoted by the path kids.tc.[4]

Applying this new method to a person mary we have to use parenthesis:

mary..(kids.tc) (4.1)

Note the difference to writing mary..kids.tc: here we apply kids to mary, and on the resulting set of persons, the method tc is applied (which probably does not make any sense).

A second point is that, on the one hand, the path kids.tc is a *scalar* reference, since it denotes a single object, i.e., the result of the invocation of the method tc on the object kids. On the other hand, in path (4.1) this single object is used to denote the invocation of a *set-valued* method.

4.2 Scalarity and Well-formedness

A reference may contain scalar methods as well as set-valued methods. However, set-valued methods cannot appear at every syntactical position. To this end, references have to be well-formed. Before giving a precise definition we present some motivating examples.

While the path

p1.age

[4] The actual definition of this method by means of PathLog-rules will follow in Section 6.

denotes the result of the application of the *scalar* method age on p1, the path

p1..assistants (4.2)

denotes the result of the application of the *set-valued* method assistants, i.e., the set of all assistants of p1. To restrict the possible elements of this set, we may use a filter. For example,

p1..assistants[salary→1000] (4.3)

denotes the set of all assistants of p1 whose salary is 1000. Set-valued references can now be used instead of explicitly given sets of objects. Instead of writing

p2[friends↠{p3,p4}] (4.4)

we may replace the explicit set by a set-valued reference:

p2[friends↠p1..assistants] (4.5)

This formula states that all the assistants of p1 are friends of p2. Note that in contrast to (4.3) the formula (4.5) does not denote a *set* of objects, it merely specificies a property of *one* object, p2, although it contains the set-valued reference p1..assistants. But this reference does not determine the scalarity of the molecule, because for molecules, only the prefix, here p2, determines the scalarity of the entire molecule.

Definition 2 Let t_i $(0 \le i \le k)$ be references and t_m, t_c simple references. A reference t is *set-valued*, iff one of the following conditions holds:

- t is a path of the form $t_0..t_m@(t_1,\ldots,t_k)$,
- t is a path of the form $t_0.t_m@(t_1,\ldots,t_k)$ where (at least) one of the t_i $(0 \le i \le k)$ or t_m is a set-valued reference,
- t is a molecule $t_0[\ldots]$ or $t_0 : t_c$ where the reference t_0 is set-valued,
- t is a simple reference of the form (t_0) where the reference t_0 is set-valued.

Otherwise, a reference is *scalar*. □

According to this definition the path

p1..assistants.salary

is set-valued, because the scalar method salary is invoked on every member in the set of assistants of p1. Thus, this path denotes the set of salaries of p1's assistants.

Certainly, a set-valued reference cannot be used at every syntactical position in a reference, e.g. in formula (4.6) it is obviously incorrect to assign a set-valued reference as result to a scalar method:

p2[boss→p1..assistants] (4.6)

Definition 3

- The filter of a molecule t is called *well-formed* iff the following conditions are fulfilled:
  - if $t = t_0[t_m@(t_1, \ldots, t_k) \rightarrow t_r]$, then t_m, all t_i $(1 \leq i \leq k)$ and t_r are scalar references,
  - if $t = t_0[t_m@(t_1, \ldots, t_k) \twoheadrightarrow s]$, then t_m and all t_i $(1 \leq i \leq k)$ are scalar references and s is either a set valued reference or an explicit set $\{t'_1, \ldots, t'_l\}$ where all t'_j $(1 \leq j \leq l)$ are scalar references,
  - if $t = t_0 : t_c$, then the class t_c is a scalar reference.
- A reference is called *well-formed* if all filters mentioned in it are well-formed. The set of all well-formed references is denoted by $\mathcal{T}$. □

In other words, the scalarity of a reference at a result position has to agree with the scalarity of the corresponding method call; furthermore, it is not allowed to use set-valued references as methods, arguments or classes in molecules.

Formulas in PathLog can now be defined as usual. Every well-formed reference may be used as an atomic formula, which in turn may serve as a basis to build literals, clauses and rules.

Well-formedness only restricts the usage of set-valued references in molecules, but not in paths. This interesting feature of PathLog is further demonstrated by the following example showing a path with a set-valued argument.

Let paidFor be a method by which we can compute the price a person paid for a vehicle. This method is applied to a *set* of vehicles which is passed to the method as a parameter:

p1.paidFor@(p1..vehicles) (4.7)

denotes the set of prices which p1 paid for all her vehicles.

The reason for the restrictions on the usage of set-valued references in molecules is that a filter in a molecule has to be unambiguous. Consider, e.g. the following molecule similar to the path in (4.7):

p1[paidFor@(p1..vehicles)→1000] (4.8)

Again, the argument of paidFor is given by the set-valued reference p1..vehicles, which in general denotes several objects, i.e., vehicles. Neither of the following interpretations seems natural to us: we could define that for at least *one arbitrary* argument belonging to the set of p1's vehicles the invocation of the method paidFor on p1 yields 1000, or we could define that for *all* of p1's vehicles as argument the invocation of paidFor on p1 has to yield 1000. Looking back at definition 3 we see that the filter in (4.8) does not fit the first restriction and therefore the reference is not well-formed.

5 Direct Semantics of PathLog

For semantics, on the one hand we are interested, whether certain statements (*formulas*) about some objects are true or false under a given semantic structure I. On the other hand, for *terms* specifying the application of a method (or a composition of applications of methods) on some object, we like to know, which objects are denoted by these terms in I. For these two aspects we need appropriate notions of *entailment* and *valuation*.

In our setting, the semantics covering both molecules and paths in their various forms is surprisingly simple, since they may simultaneously be considered as a *formula*, having a truth value, as well as a *term*, denoting an object. For this reason, we regard both molecules and paths as references. Let's see, how these two views go hand-in-hand.

Let $I = (U, \in_U, I_\mathcal{N}, I_\rightarrow, I_\twoheadrightarrow)$ be a semantic structure.

If we ask for *entailment* of a molecule $t = t_0[\ldots]$ in I, we have to check whether the object denoted by t_0 fulfills all specifications given in the filter $[\ldots]$ of t.

Consider now the entailment of a molecule t with an empty list of filters, i.e., $t = t_0[\,]$. Obviously, no specification has to be fulfilled, but t_0 has to denote an existing object. But in case t_0 is a path, it can not be taken for granted that such an object exists. A method call may be undefined for a certain object: for a bachelor john the path john.spouse does not denote an object, consequently, this path is considered false. Thus, a path is entailed by I if the object being denoted by this path indeed does exist.

The idea that a path denotes certain objects is reflected by a *valuation*. The use of a valuation function with respect to paths is motivated by the similarity between a function symbol in first order predicate calculus and a method, because both are interpreted by functions. Therefore, a path of the general form

$$t_0.m_1.m_2\ldots m_k,$$

where m_i is a method ($1 \le i \le k$), can be considered as a composition of (partial) functions

$$m_k(\ldots m_2(m_1(t_0))\ldots).$$

As a direct consequence, because the interpretation of the methods can be obtained from I, i.e., is given by the respective $I_{\rightharpoonup}$, the compositional expression can be evaluated by simply inspecting the given semantic structure I.

Molecules can now be treated in an analogous fashion. Since we may use molecules inside a path or molecule, we are interested in the objects denoted by this molecule. Consequently, we also define a valuation for molecules.

It turns out, that once we have given a semantic structure, we can conveniently switch from one view to the other.

Next we will make this introductory discussion more concrete. To deal in a uniform framework with references not denoting an object and to deal with set-valued references, we define a valuation function to yield *sets* of objects. In the case of a scalar reference, these sets are either a singleton or empty.

As long as variables are considered, a valuation is as usual a function

$$\alpha : \mathcal{V} \mapsto U$$

mapping variables to objects. This variable-valuation is extended to references w.r.t. a given interpretation, yielding a function

$$\beta_I : \mathcal{T} \mapsto 2^U.$$

Assume that $\alpha(\mathsf{X}) = u$. Then, using the corresponding valuation function β_I, evaluating β_I(X..assistants) yields the set of assistants of u. The evaluation of β_I(X.spouse) yields the empty set, if u is a bachelor, or a set containing u's spouse, otherwise.

Definition 4 A *variable-valuation* is a function $\alpha : \mathcal{V} \mapsto U$ mapping variables to objects. This valuation is extended for a given interpretation I to a function β_I mapping references to sets of objects, i.e., $\beta_I : \mathcal{T} \mapsto 2^U$. Let t_i $(0 \le i \le k)$, t_r, t'_j $(1 \le j \le l)$ be references and t_m, t_c simple references. For a well-formed reference $t \in \mathcal{T}$, the valuation $\beta_I(t)$ is defined to be the smallest set fulfilling the following conditions:

1. If $t = \mathsf{X} \in \mathcal{V}$ is a variable, then
$$\beta_I(t) = \{\alpha(\mathsf{X})\},$$
2. If $t = \mathsf{n} \in \mathcal{N}$ is a name, then
$$\beta_I(t) = \{I_{\mathcal{N}}(\mathsf{n})\},$$
3. If $t = t_0.t_m@(t_1,\ldots,t_k)$ is a path, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m,0,\ldots,k\})$, such that $I^{(k)}_{\rightharpoonup}(u_m)(u_0,\ldots,u_k)$ is defined, there holds:
$$I^{(k)}_{\rightharpoonup}(u_m)(u_0,\ldots,u_k) \in \beta_I(t).$$
4. If $t = t_0..t_m@(t_1,\ldots,t_k)$ is a path, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m,0,\ldots,k\})$ there holds:
$$I^{(k)}_{\twoheadrightarrow}(u_m)(u_0,\ldots,u_k) \subseteq \beta_I(t).$$
5. If $t = t_0 : t_c$ is a molecule, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{c,0\})$, such that
$$u_0 \in_U u_c,$$
there holds $u_0 \in \beta_I(t)$.

6. If $t = t_0[t_m@(t_1,\ldots,t_k)\rightarrow t_r]$ is a molecule, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m, r, 0, \ldots, k\})$, such that $I_{\rightarrow}^{(k)}(u_m)(u_0,\ldots,u_k)$ is defined and
$$I_{\rightarrow}^{(k)}(u_m)(u_0,\ldots,u_k) = u_r,$$
there holds $u_0 \in \beta_I(t)$.

7. If $t = t_0[t_m@(t_1,\ldots,t_k)\twoheadrightarrow t_r]$ is a molecule, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m, 0, \ldots, k\})$, such that
$$I_{\twoheadrightarrow}^{(k)}(u_m)(u_0,\ldots,u_k) \supseteq \beta_I(t_r),$$
there holds $u_0 \in \beta_I(t)$.

8. If $t = t_0[t_m@(t_1,\ldots,t_k)\twoheadrightarrow\{t'_1,\ldots,t'_l\}]$ is a molecule, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m, 0, \ldots, k\})$, such that
$$I_{\twoheadrightarrow}^{(k)}(u_m)(u_0,\ldots,u_k) \supseteq S,$$
where S is defined below, there holds $u_0 \in \beta_I(t)$.
S is the set resulting from evaluating the t'_i, i.e., $S = \{u \in \beta_I(t'_j) \mid j \in \{1,\ldots,l\}\}$.

□

As already mentioned before, entailment may then be defined w.r.t. valuation.

Definition 5 Let I be a semantic structure, t a reference and α a variable-valuation. Let further β_I be the valuation function implied by α and I. A reference *t is entailed by I w.r.t. α*, i.e., $I \models_\alpha t$, iff $\beta_I(t) \neq \emptyset$. □

Entailment of literals and clauses is defined as usual: $I \models_\alpha \phi \wedge \psi$ iff $I \models_\alpha \phi$ and $I \models_\alpha \psi$; $I \models_\alpha \phi \vee \psi$ iff $I \models_\alpha \phi$ or $I \models_\alpha \psi$; $I \models_\alpha \neg\phi$ iff not $I \models_\alpha \phi$. The meaning of quantifiers is standard: $I \models_\alpha (\forall X)\phi$ $((\exists X)\phi)$ iff for every (some, resp.) α' that agrees with α everywhere, except possibly on X, $I \models_{\alpha'} \phi$ holds. For a closed formula, we may omit the valuation α. Rules are implicitly $\forall$-quantified; entailment is defined based on the clausal form.

The aim of the following discussion is to further clarify the relationship between entailment and valuation. The expression (used as a fact)

$$\text{p1[m}\twoheadrightarrow\text{p1..assistants[salary}\rightarrow\text{1000]].} \tag{5.1}$$

defines a method m, such that p1..m denotes the set of all assistants of p1 with a salary of 1000. The same set can be defined by using a molecular style a la F-Logic:

$$\begin{aligned}&\text{p1[m}\twoheadrightarrow\text{\{X\}]} \leftarrow\\ &\quad\text{p1[assistants}\twoheadrightarrow\text{\{X[salary}\rightarrow\text{1000]\}].}\end{aligned} \tag{5.2}$$

Although in both cases the same set is defined, the semantic *explanation* is different. In (5.1), the set of all those assistants is determined by valuation and asserted to be the result of m applied on p1. However, in (5.2) entailment of the body defines that p1 has at least an assistant X, whose salary is 1000. $\forall$-quantification and entailment of the rule implies that the head of the rule is entailed for all such assistants X. Here, the variable X does range over the set of objects, i.e., the universe, and is *not* bound to a set of objects (cf. [KW93]).

Note that every reference evaluates to the set of all objects denoted by this reference, where in the case of a scalar reference we get a singleton set or the empty set. Thus, we can handle scalar and set-valued references in the same way, e.g. applying a method to a reference t means to apply this method to every member of the set $\beta_I(t)$, but not to apply the method to the set itself. Furthermore, the invocation of a set-valued method on a set of objects again yields a flat set of objects, but not a set of sets. This philosophy prevents from having multiply nested sets and the need to flatten sets.

In the following example we apply a set-valued method, e.g. projects, to a set-valued reference:

p1..assistants..projects

The valuation of this reference does not denote a set of sets, but simply the set of projects of p1's assistants.

6 Programming in PathLog

After having presented the semantics, we now discuss rules in more detail and give PathLog solutions to some interesting problems.

Rules are a means to define intensional knowledge; we can distinguish intensionally defined methods and virtual objects.

In the next example, we use a rule to define an intensional method which is defined for already existing objects:

$$\begin{aligned}&\mathsf{X[power \rightarrow Y]} \longleftarrow \\ &\qquad \mathsf{X{:}automobile.engine[power \rightarrow Y]}\end{aligned}$$

The result of this rule is to extend all given automobile-objects by a method power, derived from their engine's power. Here, existing objects are equipped with additional methods — no virtual objects are defined. This is in contrast to the following, where a path in a rule head may lead to the definition of virtual objects:

$$\begin{aligned}&\mathsf{X.boss[worksFor \rightarrow D]} \longleftarrow \\ &\quad \mathsf{X : employee[worksFor \rightarrow D].}\end{aligned} \tag{6.1}$$

This rule states that employees and their bosses work for the same department. Assume that only the information p1:employee[worksFor→cs1] is given. The method boss is not defined extensionally for p1, however, this rule defines a *virtual* object, the boss of p1. This virtual object can be referenced by applying boss to p1.[5]

In contrast to (6.1) the following rule states that only employees and their *already defined* bosses work for the same department:

$$\begin{aligned}&\mathsf{Z[worksFor \rightarrow D]} \longleftarrow \\ &\quad \mathsf{X : employee[worksFor \rightarrow D].boss[Z].}\end{aligned} \tag{6.2}$$

Our approach to virtual objects differs from the view mechanism in XSQL. There, a new class EmployeeBoss has to be defined as a view (6.3), and the view's name simultaneously serves as a function symbol, so the defined object has to be referenced by EmployeeBoss(p1):

```
CREATE VIEW EmployeeBoss
SELECT WorksFor = D
FROM Employee X                                  (6.3)
OID FUNCTION OF X
WHERE X.WorksFor[D]
```

In our setting, using methods instead of function symbols to define virtual objects makes function symbols like EmployeeBoss superfluous, and thus simplifies the query language and makes the typing system usually defined for methods (cf. [KLW93]) applicable for virtual objects.

While scalar references when used as a rule head may define virtual objects, the semantics of set-valued references as rule heads is a bit problematic. Consider a rule head with the set-valued reference:

$$\mathsf{p1..assistants[salary \rightarrow 1000]} \longleftarrow \ldots$$

We can distinguish two different cases: If no assistant at all is already defined, this rule will define exactly one assistant with salary 1000 — analogous to the scalar case. But, assuming that assistants of p1 are already defined, according to definition 5 this rule head is entailed if there is at least one assistant with salary 1000. Thus, the minimal way to satisfy this rule head leads to a non-deterministic semantics in the case of p1 having several assistants: only one arbitrary assistant is required to have this salary. Since in general this object can not be uniquely determined, we suggest to forbid set-valued references in rule heads. However, set-valued *methods* may be defined in rule heads, possibly involving set-valued sub-references in a scalar reference like in (4.5).

Now we define a set-valued method desc, which computes the transitive closure of a given method kids:

$$\begin{aligned}&\mathsf{X[desc \twoheadrightarrow \{Y\}]} \longleftarrow \mathsf{X[kids \twoheadrightarrow \{Y\}].} \\ &\mathsf{X[desc \twoheadrightarrow \{Y\}]} \longleftarrow \mathsf{X..desc[kids \twoheadrightarrow \{Y\}].}\end{aligned} \tag{6.4}$$

We may define this method even more concisely using the facts:

$$\begin{aligned}&\mathsf{X[desc \twoheadrightarrow X..kids].} \\ &\mathsf{X[desc \twoheadrightarrow X..desc..kids].}\end{aligned} \tag{6.5}$$

While in (6.4), the descendants are bound to the variable Y, here the sets are treated in their entirety. Moreover, while in (6.4) we use recursive rules, (6.5) has to be read as fixpoint equations. This further emphasizes the flexibility and conciseness of PathLog.

If we want to define the transitive closure independently of the concrete method kids as a *generic* operation (similar to [CKW93]), we can take advantage of the fact that kids formally is the name of an *object*. Consequently, we can also apply a method to this object. For our purposes, we define a method tc, which, applied to kids, yields a new method, which computes the transitive closure of kids. This new method is denoted by the path kids.tc. Since a path may be used at any syntactic position, even at the method positi-

[5] Of course, if for some object a boss-object is already defined, then this rule only affects the worksFor-method.

on, we may replace the method desc in example (6.4) by the method kids.tc. Generalizing from the concrete method kids by introducing a variable M, we can define transitive closure as a generic operation:

$$X[(M.tc) \twoheadrightarrow \{Y\}] \longleftarrow X[M \twoheadrightarrow \{Y\}].$$
$$X[(M.tc) \twoheadrightarrow \{Y\}] \longleftarrow X..(M.tc)[M \twoheadrightarrow \{Y\}].$$

Now, given the following facts,

peter[kids→→{tim,mary}].
tim[kids→→{sally}].
mary[kids→→{tom,paul}].

applying kids.tc to peter yields

peter[(kids.tc)→→{tim,mary,sally,tom,paul}].

To evaluate rules in PathLog well-known bottom-up techniques may be applied. In one situation, where a path denotes the result of a set valued method in a rule *body*, stratification of the rules becomes necessary in a similar way to [NT89]. A rule of the following structure

... ⟵ X[friends→→p1..assistants].

should only then be applied, if the set of p1's assistants is already defined. However we would like to stress that in all other cases the treatment of sets in PathLog does not imply stratification (cf. O-Logic [KW93]).

7 Conclusion

This paper presents PathLog, a rule language, whose basic building blocks are paths and molecules. PathLog generalizes path expressions in several ways. A second dimension is added to path expression which makes it possible to use only one path in situations where known one-dimensional path expressions require a conjunction of several paths. In addition, a path expression can also be used to reference virtual objects. We have shown by several examples how to adopt path expressions generalized in this way to object oriented SQL dialects.

Because of the generality in syntax, expressions in PathLog allow to query objects in a very compact way; however, PathLog has a concise direct semantics, such that even in those cases its use remains transparent to the user. Moreover, even though we have presented PathLog in terms of a deductive rule language, the main ideas of PathLog can be also applied in the context of other kinds of rule languages, e.g. production rules or active rules.

Acknowledgments

We would like to thank the referees for their insightful comments on an earlier draft of this paper. Our colleagues, Bertram Ludäscher und Paul Th. Kandzia, have contributed much to this work by many vivid discussions. Our thanks go also to the students in our group implementing a prototype of PathLog. Especially, we appreciate the suggestions of Thorsten Pferdekämper and Till Westmann, which improved the design of the language.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 238–247, 1991.

[BCD92] F. Bancilhon, S. Cluet, and C. Delobel. A query language for O_2. In François Bancilhon, Claude Delobel, and Paris Kanellakis, editors, *Building an Object-Oriented Database System – The Story of O_2*, pages 234 – 255. Morgan Kaufmann, 1992.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.

[Fis87] D.H. Fishman et al. Iris: an object-oriented database management system. In *ACM Transaction on Office Information Systems*, pages 48–69, 1987.

[GV92] G. Gardarin and P. Valduriez. ESQL2: An object-oriented SQL with F-Logic semantics. In *Proc. of the IEEE Intl. Conference on Data Engineering*, pages 320 – 327, 1992.

[Kim89] W. Kim. A model of queries for object-oriented databases. In *Proc. of the Intl.*

Conference on Very Large Data Bases, pages 423–432, 1989.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 393 – 403, 1992.

[KLW93] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical report 93/06, Department of Computer Science, SUNY at Stony Brook, April 1993. To appear in JACM.

[KW93] M. Kifer and J. Wu. A logic for programming with complex objects. *Journal of Computer and System Sciences*, 47(1):77 – 120, August 1993.

[NT89] S. Naqvi and S. Tsur. *A logical Language for Data and Knowledge Bases.* Computer Science Press, New York, 1989.

[VV93] J. Van den Bussche and G. Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *Proc. of the Intl. Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 267 – 282, 1993.

[Zan83] C. Zaniolo. The database language GEM. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207 – 218, 1983.