Reihe Informatik
4 / 1994

Access to Objects by Path Expressions and Rules

Jürgen Frohn Georg Lausen Heinz Uphoff

Access to Objects by Path Expressions and Rules

Jürgen Frohn*† Georg Lausen* Heinz Uphoff*†

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 to extend the applicability of path expressions. The rule-language PathLog proposed in the current paper generalizes path expressions in several ways. PathLog adds a *second* dimension to path expressions 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. This general use of path expressions gives rise to many interesting semantic implications.

Keywords: object oriented databases, query languages, path expressions, deductive rules

*Fakultät für Mathematik und Informatik, Universität Mannheim, 68131 Mannheim, Germany, {frohn, lausen, uphoff}@pi3.informatik.uni-mannheim.de

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

1 Introduction

Many applications demand for data models with richer structuring capabilities than the relational model, because using the relational model we are forced to organize the application structures by a set of flat relations. The missing concepts seem to be offered by the object oriented data model. Here, data is structured by means of *objects* which 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. XSQL [KKS92], O_2SQL [BCD92], ESQL [GV92] and OSQL [Fis87]).

To give a first flavor of path expressions let's 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 result 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.

In this paper we propose a language called PathLog, which, on the one hand, gives interesting solutions to the above mentioned problems, 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 main 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

$$\begin{aligned}&\mathsf{X{:}employee[age{\rightarrow}30;\ city{\rightarrow}newYork]}\\&\quad\mathsf{..vehicles{:}automobiles[cylinders{\rightarrow}4]}\\&\quad\mathsf{.color[Z]}\end{aligned} \tag{2.1}$$

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 $\mathsf{newYork}$. 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 allows to define properties of the objects referred to inside a path; only those objects are referenced, which fulfill the specified properties.

To see that our syntax could also be used in a SQL-style, putting (2.1) in XSQL-style gives

```
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 in the above example [city→newYork] by

```
[city→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 several FROM- and WHERE-clauses. The result of the set valued path X.vehicles is treated like a class; hence the second FROM-clause is necessary to *flatten* this set of objects explicitly.

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

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

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 1-dimensional reference. In XSQL a path can be used as a 1-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:

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

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 much simpler than it is in F-logic, because methods can do the same job function symbols had to do in that framework.[1] Second, the usage of methods can be controlled by signatures in the same way as in [KLW93], which makes type checking techniques applicable.

3 Basic Terminology

For the purposes of this paper an object is sufficiently described by its *identity*, its *state* and its *class-membership*. The object identity is a property distinguishing objects from each other. 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 of a person. For this reason, we do not stress the difference between methods and attributes. Both may be scalar or set valued, and may have arguments.

To simplify the framework, objects also denote classes and methods. As a direct consequence, class-membership reduces to a binary relation on objects.

Our simple setting can now be summarized as follows (cf. [KLW93]). 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. Note that only these names are directly accessible to the user, in contrast to the objects' identity, which is a storage level concept. 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 the literals are built out of path expressions, which

[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.

will be defined formally in section 4.

To define a formal semantics we need a *semantic structure*, I, which can be perceived as a set of objects and their properties. From I we can obtain all the needed information about this set of objects. 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 $I_{\mathcal{N}} : \mathcal{N} \mapsto U$ mapping names to object shows which objects are denoted by the names. The class hierarchy $\in_U \subseteq U \times U$ is a partial order 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 elements of U a partial function $U^k \overset{p}{\mapsto} U$, when used as a scalar method with $k-1$, $k \geq 1$, arguments, $I_{\twoheadrightarrow}$ is a function which assigns to elements of U, when used as a set valued method with $k-1$, $k \geq 1$, arguments, a function $U^k \mapsto 2^U$.

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* like .spouse, while a molecule consists of a reference followed by a *filter* like [boss→mary]. Note how paths and molecules may be mutually nested: a path mary.spouse is a reference and may therefore be used in the molecule mary.spouse[boss→mary], which in turn may again be used in the path mary.spouse[boss→mary].age to access the age of the object. It is also possible to nest terms inside the filter, e.g. the name mary may be further specified as in mary.spouse[boss→mary[age →25]].

Definition 1 Given an alphabet, 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 m, c are simple references,
  - then the expressions $t_0.m@(t_1, \ldots, t_k)$ and $t_0..m@(t_1, \ldots, t_k)$ are *references*, also called *paths*.
  - then the expressions $t_0[m@(t_1, \ldots, t_k) \rightarrow t_r]$,

$t_0[m@(t_1,\ldots,t_k)\twoheadrightarrow\{t'_1,\ldots,t'_l\}]$, $t_0[m@(t_1,\ldots,t_k)\twoheadrightarrow t_r]$, and $t_0 : c$ are *references*, also called *molecules*.

The terms t_i, t'_j, m, c are the *sub-references* of the respective references. □

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 brackets 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.

Bracketed references are used to change the usual left-to-right evaluation sequence of a reference. To give an example, let list be a method that yields for any class c the corresponding class "list of c". For example, integer.list denotes the class of all lists of integers. The membership of an object L in this class then is expressed by L : (integer.list). Note the difference to writing L : integer.list; here we express that L is an integer, on which method list has to be applied.

4.2 Scalarity and Well-formedness

A path may be used to either reference exactly one object, or to reference a set of objects. While a path like

p1.age

denotes the invocation of the *scalar* method age, the path

p1..assistants (4.1)

denotes the result of the application of the *set valued* method assistants on p1, i.e., the set of all assistants of p1. With an additional molecule this set can be restricted. For example,

p1..assistants[salary→1000] (4.2)

denotes the set of all assistants of p1 whose salary is 1000. Such set valued references can now be used in the same way as explicitly given sets of objects, e.g. in

p2[friends↠{p3,p4}] (4.3)

where the result of the set valued method friend is specified; we may replace the explicit set by a set valued reference:

p2[friends↠p1..assistants] (4.4)

This formula states that the assistants of p1 are friends of p2. Note that in contrast to (4.2) the formula (4.4) does not denote a *set* of objects, it is merely the specification of a property of *one* object, p2, although it contains the set valued reference p1..assistants. But this sub-reference does not determine the scalarity of the molecule, because for molecules, only the first sub-reference, here p2, determines the scalarity of the entire molecule.

Definition 2 A reference t is *set valued*, iff

- it is a path of the form $t_0..m@(t_1,\ldots,t_k)$; or
- it is a path of the form $t_0.m@(t_1,\ldots,t_k)$ where (at least) one of the references t_i $(0 \leq i \leq k)$ or m is set valued.
- it is a molecule $t_0[\ldots]$ or $t_0 : c$ where the reference t_0 is set valued; or
- it is a simple reference of the form (t_0) where the reference t_0 is set valued.

Otherwise, a reference is *scalar.* □

Note that a path like p1..assistants.salary also is set-valued, because the scalar method salary is invoked on the set of assistants of p1. Thus, this path denotes the set of salaries of p1's assistants.

Certainly, a set valued references cannot be used at every syntactical position in a reference, e.g. in formula (4.5) it is obviously incorrect to assign a set valued reference as result to a scalar method.

$$\text{p2[boss}\rightarrow\text{p1..assistants]} \tag{4.5}$$

Definition 3 A reference t is *well-formed* iff every sub-reference occuring in t is well-formed and the following holds:

- if t is a molecule $t_0[m@(t_1,\ldots,t_k)\rightarrow t_r]$, then m, all t_i $(1 \leq i \leq k)$ and t_r are scalar references; and
- if t is a molecule $t_0[m@(t_1,\ldots,t_k)\twoheadrightarrow s]$, then 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; and
- if t is a molecule $t_0 : c$, then the class c is a scalar reference. □

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

The set of all well-formed references is denoted with $\mathcal{T}$. These references may be used as atomic formulas, which in turn may serve as a basis to build literals, clauses and rules in the usual way.

Note that 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 examples.

It is allowed to compose a path using a set valued reference and a scalar method, like in

p1..assistants.salary

Here we apply method salary on all the assistants of p1. The result is a set of salaries.

We can also apply a set valued method, e.g. projects, to a set valued reference:

p1..assistants..projects

The result is the set of projects of all the assistants of p1.

Finally, let paidFor be a method by which we can compute the price a person paid for a vehicle. Here, this method is applied on a *set* of vehicles which is passed to the method as a parameter.

```
p1.paidFor@(p1..vehicles)
```

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

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 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 an object exists denoted by this path.

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 $t_0.m_1.m_2 \ldots m_k$, $k \geq 1$, 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 $I_{\rightarrow}$, 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. Now 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 usual, valuation is defined for a given variable-valuation, i.e., a function $\alpha : \mathcal{V} \mapsto U$ assigning objects to variables. This variable-valuation is extended to references w.r.t. a given interpretation, yielding a function $\beta_I : \mathcal{T} \mapsto 2^U$.

Assume e.g. α assigns the object with name john to the variable X. Then, using the corresponding valuation function, evaluating X..assistants yields the set of assistants of john. Since john was said to be a bachelor, evaluating X.spouse yields the empty set.

We will evaluate even those sub-references of a reference using this β_I-function, which are scalar due to the well-formedness of references (cf. definition 3). Thus, in the following definition the evaluation of the scalar references yields at most a singleton.

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., $\beta_I : \mathcal{T} \mapsto 2^U$. 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 $I_{\rightarrow}^{(k)}(u_m)(u_0, \ldots, u_k)$ is defined, holds:
$$I_{\rightarrow}^{(k)}(u_m)(u_0, \ldots, u_k) \in \beta_I(t).$$
4. If $t = t_0..t_m@(t_1, \ldots, t_k)$ is a path, then for all objects $u_i \in \beta_I(t_i)$ $(i \in \{m, 0, \ldots, k\})$ holds:
$$I_{\twoheadrightarrow}^{(k)}(u_m)(u_0, \ldots, u_k) \subseteq \beta_I(t).$$
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
$$u_0 \in_U u_c,$$
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
$$I_{\rightarrow}^{(k)}(u_m)(u_0, \ldots, u_k) = u_r,$$
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
$$I_{\twoheadrightarrow}^{(k)}(u_m)(u_0, \ldots, u_k) \supseteq \beta_I(t_r),$$
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, 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, the entailment for a reference may then be defined w.r.t. this 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. t is entailed by I w.r.t. α, i.e., $I \models_\alpha t$, iff $\beta_I(t) \neq \emptyset$. □

Entailment of literals, clauses and rules is defined as usual.

Let's point out some interesting features of this semantics. Set valued references are true, if there is at least one object corresponding to the reference. Thus, this reference

p1..assistants[salary→1000]

denoting all assistants of p1 with salary 1000 is true, if there is at least one such assistant.

It is possible to access successively all assistants in this set by binding them to a variable:

p1[assistants →→{X[salary→1000]}]

This term does not denote a set of objects, but the semantics defines this scalar reference to be true if X is assigned such an assistant. The variable X is *not* bound to the entire set, it ranges only over the universe of objects. Thus, using such a reference in a rule body allows to access all such assistants.

The philosophy that a reference evaluates to the set of all objects denoted by this reference prevents from having multiply nested sets, i.e., the path

john..kids..kids

does not denote a set of sets, but simply the set of john's grandchildren.

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; here we can distinguish intensionally defined methods and virtual objects.

In the first example, we use a rule to define an intensional method for already existing objects:

```
X[power →Y] ⟵
    X:automobile.engine[power→Y]
```

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:

```
X.boss[worksFor→D] ⟵
  X : employee[worksFor→D].                (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. Although the method boss is not defined extensionally for p1, this rule defines a virtual object, the boss of p1. This virtual object can be referenced by applying boss to p1.

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

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

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 addressed 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.

Considering set valued references in a rule head, we observe that our semantics implies only the existence of one object described by that reference. Since in general this object can not be uniquely determined, the usage of set valued *references* in rule heads should be forbidden.

However, set valued *methods* may be defined in rule heads, possibly involving set valued sub-references in a scalar reference like in (4.4). In the sequel, we define a set valued method desc, which represents the transitive closure of a given method kids:

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

If we want to define the transitive closure independently of the concrete method kids as a *generic* operation (similar to [CKW89]), we can take advantage of the fact that e.g. kids in our model itself is the name for an *object*. Consequently, we can also apply a method to this object. For our purpose, we define a method tc, which, applied to kids, yields a new method, representing 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 position, we may replace the method desc in our example 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:

$$\begin{array}{lcl} \mathsf{X[(M.tc){\twoheadrightarrow}\{Y\}]} & \longleftarrow & \mathsf{X[M{\twoheadrightarrow}\{Y\}].} \\ \mathsf{X[(M.tc){\twoheadrightarrow}\{Y\}]} & \longleftarrow & \mathsf{X..(M.tc)[M{\twoheadrightarrow}\{Y\}].} \end{array}$$

Now, given the following facts,

$$\begin{array}{l} \mathsf{peter[kids{\twoheadrightarrow}\{tim,mary\}].} \\ \mathsf{tim[kids{\twoheadrightarrow}\{sally\}].} \\ \mathsf{mary[kids{\twoheadrightarrow}\{tom,paul\}].} \end{array}$$

applying kids.tc to peter yields

$$\mathsf{peter[(kids.tc){\twoheadrightarrow}\{tim,mary,sally,tom,paul\}].}$$

To evaluate rules in PathLog well-known bottom-up techniques may be applied. In one situation, where a path is used as a 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

$\ldots \longleftarrow \mathsf{X[friends{\twoheadrightarrow}p1..assistants]}.$

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, similar to e.g. 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.

References

[AB91] Serge Abiteboul and Anthony Bonner. Objects and views. In *Proc. SIGMOD*, 1991.

[BCD92] François Bancilhon, Sophie Cluet, and Claude 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.

[CKW89] W. Chen, M. Kifer, and D. S. Warren. Hilog : A first-order semantics of higher-order logic programming constructs. In *Proc. of the North American Conference on Logic Programming*, 1989.

[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 *Proceedings of the IEEE Intl. Conference on Data Engineering*, pages 320 – 327, 1992.

[KKS92] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In *Proceedings 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, June 1993. to appear in JACM.

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

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

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

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