Fixpoint calculus for Querying Semistructured Data

N. Bidoit and M. Ykhlef

LaBRI (U.M.R. 5800 du CNRS) Université Bordeaux I
351, Cours de la Libération, F-33405 Talence
email : {Nicole.Bidoit, Mourad.Ykhlef}@labri.u-bordeaux.fr

Abstract. The paper proposes two query languages for semistructured data G-Fixpoint and G-While whose expressive power is comparable to Fixpoint and While respectively. These languages are multi-sorted like logic languages integrating fixpoint path expressions.

1 Introduction

Defining query languages for semistructured data has received a lot of attention during the last few years [1, 7] [11, 10] [20, 3, 8] [14, 15] [16, 4, 17, 12] [13, 21, 5, 9]. The target applications are biological data management, structured documents, heterogeneous data integration, etc. Semistructured data, that is data whose structure is not constrained by a schema, are usually represented in a graph-like manner. The data graph models vary slightly from one approach to an other: graphs may be restricted to trees, data may be associated to edges only, or to edges and terminal vertices, etc. In our framework, we make the choice to represent semistructured data by general multi-graphs called db-graphs whose vertices and edges both carry data. This choice seems to fit in a more natural way applications such as the Web.

Most of the languages proposed so far have been designed as extensions of SQL with, among others, the advantage to provide a userfriendly syntax and commercial flavor. The major focus of the paper is on defining a graph query language in a multi-sorted calculus like style.

This graph query language is closed, meaning that the answer to a query over a db-graph is a db-graph (in the same spirit as the answer of a query over a relational database is a relation). Let us consider the Web db-graph of Figure 3 and the query asking for DB group's projects. The answer will be the sub-graph(s) rooted at the node labeled by projects. Obviously, from a practical point of view, for some application, it may be more convenient to give as equivalent answers the nodes labeled by projects.

In the relational model, a query extracts a relation from the database based on properties of tuples in the database. In our framework, a graph query extracts sub-graphs from the db-graph based on properties of paths. The two major properties that we use to retrieve sub-graphs are: reachability of the root of a sub-graph via a specified path; existence of some paths in the sub-graph.

Thus the core of our language is a path calculus providing a form of navigational querying. The path calculus investigated here is based on path expressions as most other languages do. Its main feature is that path expressions are extended with a fixpoint operator in order to increase its expressive power. The fixpoint operator introduced here has two effects: it allows one to use some form of iteration; it gives the ability to define n-ary properties over paths. The first point is standard. Unsurprisingly, although the Kleene closure is not part of our path expression constructors, we are able to simulate it using the fixpoint operator. The second point concerns the fact that standard path expressions define monadic properties over paths. In some cases, for instance in order to collect pairs of paths having same length, it is useful to have the ability to express n-ary properties over paths. The fixpoint operator allows us to translate the transitive closure of a binary relation represented by a specific db-graph. This is done without the need to introduce new edges in the db-graph. The main results of the paper investigate the expressive power of the fixpoint path calculus. In the one hand, we show that, while hiding the computation of simple paths, any query in Path-Fixpoint can be expressed by a query in Fixpoint [2] over a relational database. The same kind of result holds for the non inflationary version of the fixpoint path calculus Path-While versus While [2]. On the other hand, any query in Fixpoint (resp. in While) over a relational database can be expressed by a query in Path-Fixpoint (resp. in Path-While) by converting the relational database into a db-graph. Thus we claim that our graph query languages G-Fixpoint and G-While based respectively on the fixpoint path query sublanguages Path-Fixpoint and Path-While subsume all known existing semistructured data query languages. However, it is important to know that these languages lack one of the expected features for query languages: the complexity of answering a Path-fixpoint (resp. a Path-While) query may not be polynomial because of simple path computation. Conditions in the spirit of range-restriction [16] have to be studied in order to specify reasonably computable graph queries.

Related Work: The language GraphLog [11, 10] specifies queries using path regular expression. It is shown equivalent to FO+TC or to stratified linear Datalog.

The language UnQL [8] includes the kleene closure which is not sufficient to express the transitive closure of a binary relation represented by a graph. The expressive power of UnQL is strictly included in NLOGSPACE.

Lorel [3] and POQL [9] as UnQL can not compute the transitive closure of a binary relation.

WebLog [15] is a language defined in the spirit of Datalog and recursive Datalog like rules play the role of path regular expressions used in other languages. WebLog is not powerful enough to express transitive closure. The same holds for W3QS [14] and WebSQL [16]. The language STRUQL [12] allows one to express the transitive closure of a binary relation although this requires the creation of edges. STRUQL is equivalent to FO+TC. The hypertext language Gram [6] is strictly less expressive than the class of FO queries when the Kleene closure is not included and express some form of transitive closure when it is added.

Outline of the paper: The next section presents the data model. Section 3 develops the presentation of the path calculus and its fixpoint (Path-Fixpoint) and while (Path-While) extensions. In section 4, the graph query language G-Fixpoint (resp. G-While) induced by the path query language Path-Fixpoint (resp. Path-While) are presented. Section 5 investigates the expressive power of the path query languages Path-Fixpoint and Path-While.

2 Data model

Definition 1. [**db-graph**] *Let $\mathcal{V}$ be an infinite set of values. A db-graph $G = (N, E, \lambda_N, \lambda_E, org, dest)$ over $\mathcal{V}$ is a directed labeled multi-graph where:*

1. *N is a finite set of vertex identifiers and E is a finite set of edge identifiers,*
2. *org (resp. $dest$) is a mapping from E into N which gives the origin (resp. destination) of edges,*
3. *λ_N is a partial labeling mapping from N into $\mathcal{V}$,*
4. *λ_E is a partial labeling mapping from E into $\mathcal{V}$ such that:*
   $(\forall e_1 \in E)(\forall e_2 \in E)\ (e_1 \neq e_2 \wedge org(e_1) = org(e_2) \wedge dest(e_1) = dest(e_2))$
   $\Rightarrow \lambda_E(e_1) \neq \lambda_E(e_2)$.

Condition 4 says that pairs of vertices having same origin and same destination should have different labels. Although the definition allows multiple labeled edges linking two vertices (as long as the labels are distinct), we make the restriction that only one unlabeled edge may link two vertices.

An edge e is totally defined by its origin, its destination and its label ($org(e)$, $dest(e)$, $\lambda_E(e)$). A path is given by a list of edges $\langle e_1, e_2, \ldots, e_k \rangle$ such that $dest(e_i)$=$org(e_{i+1})$ for i=1..k-1. The empty path ε is an empty list of edges. A path is simple w.r.t edges if all its edges are distinct. The mappings org and $dest$ defined on edges are extended to path in the usual manner.

Definition 2. [**Rooted and maximal subgraph**] *A vertex r of a graph G is a source for G if there exists a path from r to any other vertex of G. A rooted graph is a graph with at least one source* [1]*. If G' is a graph rooted at r and is a subgraph of G, the graph G' is said maximal if each vertex of G reachable from r is also reachable from r in G', each edge e of G such that $org(e)$ is reachable from r is an edge of G'.*

A maximal subgraph rooted at r is totally characterized by (one of) its source. Given a db-graph G, it will be convenient in the sequel to consider the set of all maximal rooted subgraphs of G. This set is denoted $\mathrm{Ext}(G)$. For instance, considering again the Web example, each of the maximal rooted subgraphs corresponds to the part of the Web which can be explored in a "forward manner" via the links starting from a specific Web page.

[1] The path $\langle (n_1, n_2, l_1), (n_2, n_1, l_2) \rangle$ is a graph with two sources n_1 and n_2.

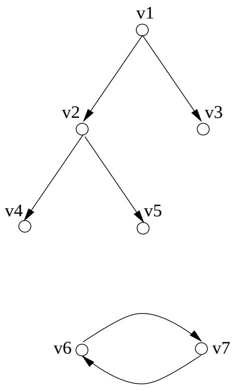

Fig. 1. An example db-graph.

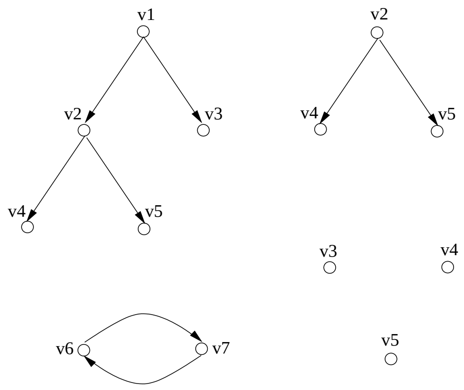

Fig. 2. Db-graph extension.

Example 1. The Figure 1 gives a simple example of a db-graph G. For the sake of simplicity vertex and edge labels are not mentioned. The set $\mathrm{Ext}(G)$ contains six subgraphs which are depicted in Figure 2.

The notion of equality over graphs adopted here is that of bisimulation [19].

Definition 3. [**Bisimulation**] *Two rooted graphs* $G_1 = (N_1, E_1, \lambda_{N_1}, \lambda_{E_1}, org_1, dest_1)$ *and* $G_2 = (N_2, E_2, \lambda_{N_2}, \lambda_{E_2}, org_2, dest_2)$ *are bisimilar, denoted* $G_1 \sim G_2$, *if there is a binary relation* $\simeq \subseteq N_1 \times N_2$ *such that:*

1. $r_1 \simeq r_2$ *where* r_1 *(resp.* r_2*) is a source of* G_1 *(resp.* G_2*).*
2. $(\forall n \in N_1)(\forall n' \in N_2)\ (n \simeq n' \Rightarrow \lambda_{N_1}(n) = \lambda_{N_2}(n'))$.
3. $(\forall n \in N_1)(\forall n' \in N_2)$
   $((\exists (n,m,l) \in E_1)\ n \simeq n') \Rightarrow ((\exists (n',m',l) \in E_2)\ m \simeq m')$ *and*
   $((\exists (n',m',l) \in E_2)\ n \simeq n') \Rightarrow ((\exists (n,m,l) \in E_1)\ m \simeq m')$.

Example 2. Figure 3 represents a small portion of the Web as an example of a db-graph. The vertices (resp. the edges) represent HTML documents (resp. links between documents). Vertex identifiers are http addresses of documents. The vertex identified by `http://www.labri.u-bordeaux.fr` characterizes a maximal rooted subgraph and represents the LaBRI server. The vertex identified by `http://www-db.stanford.edu` has two outgoing edges resp. labeled by `members` and `projects` which point to subgraphs containing information about the members of the DB group at Stanford and resp. its projects.

3 Path calculus

In the remainder of the paper, in order to simplify the presentation, the term **path** always refers a path which is **simple w.r.t edges**.
As outlined in the introduction, the graph query language defined here intends

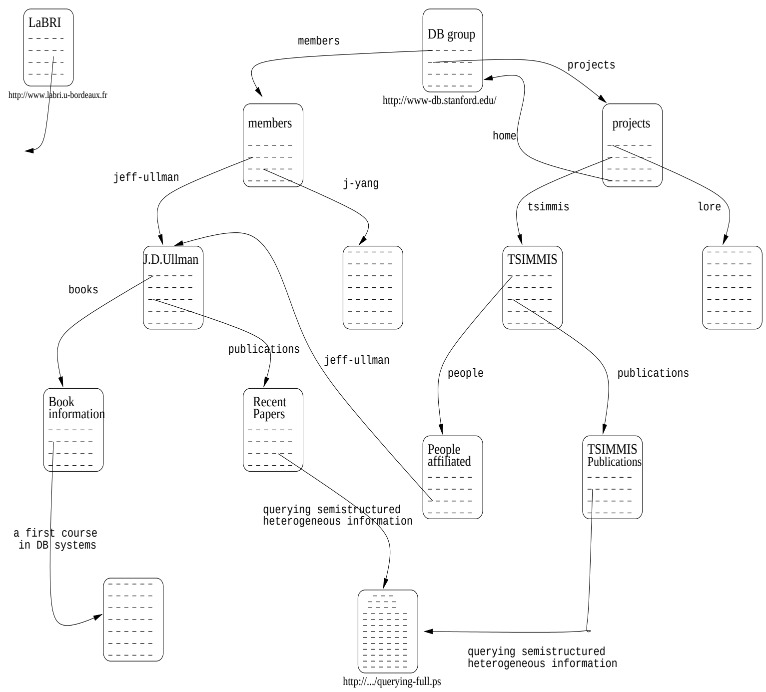

Fig. 3. An example Web db-graph.

to retrieve some subgraphs of the database, as a matter of fact maximal rooted subgraphs (or equivalently their sources). The retrieval process is essentially supported by specifying paths and roughly speaking, the subgraphs returned are either those rooted at the destination or at the source of these paths. Thus it obviously turns out that the first task is to define a path query language. The core of our path query language is rather classical and based on path expressions. However our language is made more powerful by introducing fixpoint on path formulas which, intuitively, allows one to use some form of iteration and also to define intentional n-ary predicates over paths. Thus fixpoint path expressions introduce much richer "contexts" than simple path expression for querying db-graphs.

The language used to define path expressions and formulas is very similar to a multi-sorted first order language. Because in our data model, data are associated to vertices, our path expressions are slightly different from those of Lorel [20, 3] or UnQL [8]. They may contain data variables as abstractions of the content of vertices.

In the following, we assume that the set of values $\mathcal{V}$ is fixed and that four sorts of variable sets are given: a set of graph (resp. path, label, and data) variables.

Graph and path variables are denoted $X, Y, Z, \ldots$. Label (resp. data) variables are denoted $x, y, z, \ldots$ (resp. $\alpha, \beta, \gamma, \ldots$). A **graph term** is either a rooted graph or a graph variable. A **path term** is either a path or a path variable. A **label** (resp. a **data**) **term** is either a label in $\mathcal{V}$ (resp. a value in $\mathcal{V}$) or a label (resp. data) variable.

Although introduced above, graph variables and terms are not needed in this section.

Definition 4. [**Path expression**] *A path expression over $\mathcal{V}$ is defined recursively by:*

1. *An* (*elementary*) *path expression is either a path variable or a label term. An elementary path expression is a pre-free and post-free path expression.*
2. *If s is a pre-free* (*resp. post-free*) *path expression and t is a data term then $t \triangleleft s$* (*resp. $s \triangleright t$*) *is a path expression such that:*
   if s is post-free then $t \triangleleft s$ is pre-bound and post-free otherwise it is pre-bound and post-bound and if s is pre-free then $s \triangleright t$ is pre-free and post-bound otherwise it is pre-bound and post-bound.
3. *If s_1 (resp. s_2) is a post-free (resp. pre-free) path expression then $s_1.s_2$ is a path expression such that:*
   if s_1 is pre-free then $s_1.s_2$ is pre-free otherwise it is pre-bound and if s_2 is post-free then $s_1.s_2$ is post-free otherwise it is post-bound.

Example 3. The path expression `members.y.publications` intends to represent all paths of length three whose first (resp. last) edge is labeled by `members` (resp. `publications`). It is pre-free and post-free.
The path expression "`DB group`"◁`X`▷"`Ullman`" intends to capture all paths whose origin (resp. destination) is equal to the data "`DB group`" (resp. "`Ullman`"). It is pre-bound because the origin is constrained (here by a value) and post-bound because the destination is also constrained.

Note that we do not include path expressions of the form s^* where * denotes the Kleene closure as it is usually done in most path query languages. We will see later on that these expressions are special cases of fixpoint path expressions. Note also that we do not make use of a special symbol # usually called "joker" and intended to abstract any label. The paths captured in a db-graph G by a path expression are now defined formally. The definition makes use of the notion of active domain.

Definition 5. [**Active domain**] *The active domain of a db-graph G denoted $adom(G)$ is the triple $(\mathcal{P}_G, \mathcal{L}_G, \mathcal{D}_G)$ such that $\mathcal{P}_G$ (resp. $\mathcal{L}_G$, $\mathcal{D}_G$) is the set of all paths* (*resp. labels and data*) *in G, including the empty path ϵ.*

A valuation of a variable x over the active domain $adom(G)$ assigns to x a value in either $\mathcal{P}_G$, or $\mathcal{L}_G$, or $\mathcal{D}_G$ depending on the sort of x (path, label or data). The notion of a valuation is extended to a set of variables in the usual manner.

Definition 6. [**Spelling**] *Let* $G = (N, E, \lambda_N, \lambda_E, org, dest)$ *be a db-graph. Let* s *be a path expression whose set of variables is* $Var(s)$*. Let* ν *be a valuation of* $Var(s)$ *over* $adom(G)$*. The set of paths spelled in* G *by* s *under the valuation* ν*, denoted* $Spell_G(\nu(s))$ *is defined by:*

1. s *is* X *and* $Spell_G(\nu(s))=\{\nu(X)\}$,
2. s *is* t *and* $Spell_G(\nu(s))=\{\langle e\rangle \mid e \in E$ *and* $\lambda_E(e) = \nu(t)\}$,
3. s *is* $t \triangleleft s_1$ *and* $Spell_G(\nu(s))= \{p \mid p \in Spell_G(\nu(s_1))$ *and* $\lambda_N(org(p)) = \nu(t)\}$,
4. s *is* $s_1 \triangleright t$ *and* $Spell_G(\nu(s))= \{p \mid p \in Spell_G(\nu(s_1))$ *and* $\lambda_N(dest(p)) = \nu(t)\}$,
5. s *is* $s_1.s_2$ *and*
   $Spell_G(\nu(s))= \{p_1.p_2 \mid p_1 \in Spell_G(\nu(s_1))$ *and* $p_2 \in Spell_G(\nu(s_2))$
   and $dest(p_1) = org(p_2)$ *and* $p_1.p_2$ *is simple* $\}$ [2].

Example 4. Let G be the path $\langle(n_1, n_2, l_2), (n_2, n_1, l_1), (n_1, n_3, l_2)\rangle$ and let s be the path expression $l_1.l_2$. The set of paths $Spell_G(\nu(s))$ is

$$\{\langle(n_2, n_1, l_1), (n_1, n_2, l_2)\rangle, \langle(n_2, n_1, l_1), (n_1, n_3, l_2)\rangle\}.$$

Path queries are specified by path formulas.

Definition 7. [**Atomic path formula**] *An atomic path formula is an expression having one of the following forms:*

1. *a path expression,*
2. $t_1 = t_2$ *where* t_1 *and* t_2 *are terms of the same sort among path, label, data;* $=$ *is the (polymorphic) equality predicate symbol,*
3. $t \in s$ *where* t *is a path term,* s *is a path expression and* $\in$ *is a set membership predicate symbol.*

Note here that paths are compared using equality. Intuitively, the formula $t \in s$ intends to check whether t is one of the paths spelled by the path expression s.

Definition 8. [**Path formula**] *A path formula is an expression having one of the following forms:*

1. *an atomic path formula,*
2. $(\phi \wedge \psi)$ *(resp.* $(\phi \vee \psi)$, $\neg\phi$, $(\exists x)\ \phi$, *and* $(\forall x)\ \phi$*)*
   where ϕ *and* ψ *are path formulas and* x *is a variable (of any of the sort path, label, data).*

The notion of "free" and "bound" occurrences of variables is defined in the usual manner. In the remainder, we will assume that path formulas are normalized in the sense that no distinct pair of quantifiers bind the same variable and no variable occurs both free and bound. Variable substitution may be used to normalize a path formula [2]. The set of variables having at least one free occurrence in the path formula φ is denoted by Free(φ).

[2] Note that we use the same symbol to denote the concatenation over path expressions and the concatenation of paths. Recall also that $\epsilon.p = p.\epsilon = p$ for any path p where ϵ is the empty path.

Example 5. Intuitively, the path formula

$$\text{“}\texttt{DB group}\text{”}\triangleleft\texttt{X}\triangleright\text{“}\texttt{Ullman}\text{”} \;\wedge\; \neg(\exists\texttt{U})(\exists\texttt{V})\ \texttt{X}\in\texttt{U.projects.V}$$

related to the Web database, will be true for a path valuation p of X such that p connects the “`DB group`”'s page to the “`Ullman`”'s page and p does not contain any edge labeled by `projects`.

Path queries (simple ones) are now defined in a straightforward manner.

Definition 9. [**Path Query**] *A path query is an expression of the form*

$$\{(X_1, \ldots, X_n) \mid \varphi\}$$

where φ is a path formula, Free(φ)=$\{X_1, \ldots, X_n\}$ and for i=1..n, X_i is a path variable.

Note that in general, a path query will return **tuples** of paths.

Example 6. [**Navigation and Content search**] The following path query, evaluated on the Web database, will return all paths linking the “`DB group`”'s page and the “`Ullman`”s'page.

$$\{\texttt{(X)} \mid \text{“}\texttt{DB group}\text{”}\triangleleft\texttt{X}\triangleright\text{“}\texttt{Ullman}\text{”}\}$$

Example 7. [**Common edge**] The path query below is a yes/no query and will return “yes” if the db-graph queried contains two distinct paths having at least a common edge.

$$\{\ \mid (\exists\texttt{X})(\exists\texttt{Y})(\exists\texttt{Z})(\exists\texttt{U})(\exists\texttt{V})(\exists\texttt{y})\ (\texttt{X.Y.Z} \wedge \texttt{U.Y.V} \wedge \texttt{Y}\in\texttt{y} \wedge (\neg(\texttt{X = U}) \vee \neg(\texttt{Z = V})))\}$$

The next definitions formally define the answer of a path query. The notion of active domain needs to be extended before.

Definition 10. [**Active domain of a path formula**] *The active domain of a path formula φ denoted adom(φ) is the triple ($\mathcal{P}_\varphi$, $\mathcal{L}_\varphi$, $\mathcal{D}_\varphi$) where $\mathcal{P}_\varphi$ is the set of all ground path terms in φ, $\mathcal{L}_\varphi$ is the set of all ground label terms in φ and all labels of the paths in $\mathcal{P}_\varphi$, and $\mathcal{D}_\varphi$ is the set of all ground data terms in φ and all vertex data of the paths in $\mathcal{P}_\varphi$.*

When G and φ are clearly identified from the context, *adom* denotes ($\mathcal{P}_G\cup\mathcal{P}_\varphi$, $\mathcal{L}_G\cup\mathcal{L}_\varphi$, $\mathcal{D}_G\cup\mathcal{D}_\varphi$). A valuation of a variable x over the active domain *adom* assigns to x a value in either $\mathcal{P}_G\cup\mathcal{P}_\varphi$, $\mathcal{L}_G\cup\mathcal{L}_\varphi$, or $\mathcal{D}_G\cup\mathcal{D}_\varphi$ depending on the sort of x (path, label or data).

Definition 11. [**Path formula satisfaction**] *Let φ be a path formula and ν be a valuation of Free(φ) over adom. The db-graph G satisfies the formula φ under the valuation ν, denoted $G \models \varphi\,[\nu]$, if*

1. φ *is a path expression and* $Spell_G(\varphi[\nu]) \neq \emptyset$.
2. φ *is* $t_1 = t_2$ *and* $\nu(t_1) = \nu(t_2)$ *if* t_1 *and* t_2 *are both path/label/data terms.*
3. φ *is* $t \in s$ *and* $\nu(t) \in Spell_G(\nu(s))$.
4. φ *is* $(\phi \wedge \psi)$ *and* $G \models \phi[\nu]$ *and* $G \models \psi[\nu]$.
5. φ *is* $(\phi \vee \psi)$ *and* $G \models \phi[\nu]$ *or* $G \models \psi[\nu]$.
6. φ *is* $\neg\phi$ *and* $G \not\models \varphi[\nu]$ *(i.e.* $G \models \varphi[\nu]$ *does not hold).*
7. φ *is* $(\exists x)\ \phi$ *and*
   if x *is a path (resp. label or data) variable then there exists a value* a *in* $\mathcal{P}_G \cup \mathcal{P}_\varphi$ *(resp. in* $\mathcal{L}_G \cup \mathcal{L}_\varphi$ *or in* $\mathcal{D}_G \cup \mathcal{D}_\varphi$*) such that* $G \models \phi[\nu \cup \{x/a\}]$.
8. φ *is* $(\forall x)\ \phi$ *and*
   if x *is a path (resp. label or data) variable then for each value* a *in* $\mathcal{P}_G \cup \mathcal{P}_\varphi$ *(resp. in* $\mathcal{L}_G \cup \mathcal{L}_\varphi$ *or in* $\mathcal{D}_G \cup \mathcal{D}_\varphi$*),* $G \models \phi[\nu \cup \{x/a\}]$.

Note that if φ is the path expression X then there exists at least a valuation of X, namely $\nu(X) = \epsilon$, such that $Spell_G(\varphi[\nu]) \neq \emptyset$. To extract non empty paths through a path expression X, one needs to explicitly express it by $X \wedge \neg(X = \epsilon)$. As this situation may occur frequently, it could be convenient to introduce an abbreviation for such a formula like X^+. This point is not developed further here.

Definition 12. [**Path query answering**] *The image of a db-graph* G *under the path query* q *specified by* $\{(X_1, \ldots, X_n) \mid \varphi\}$ *is*

$$q(G)=\{(\nu(X_1), \ldots, \nu(X_n)) \mid \nu \text{ valuation of Free}(\varphi) \text{ over } adom \text{ and } G \models \varphi[\nu]\}.$$

The size of the active domain (which include all **simple paths**) of a db-graph is exponential in the input. This entails that evaluation of path queries with universal quantifiers, for instance, will lead to an exponential cost. This is not reasonable for database querying and restrictions on bindings and quantifications should be added in order to solve this problem. We do not proceed to this study here although ideas from [16] can be borrowed.

Our path query language is quite similar to the relational calculus although relation predicate symbols do not occur. However, the reader would have noticed that path expressions define unary relations and path formulas define intentional n-ary relations over paths. Just as in the case of the relational calculus, we now provide an inflationary extension of path queries with recursion by introducing a fixpoint operator construct allowing the iteration of path formulas evaluation up to a fixpoint. The difficulty to define such fixpoint operator resides in the absence of relation predicate symbols. It turns out that a rather simple notation convention solves this difficulty.

Definition 13. [**Path fixpoint operator**] *Let* $\varphi(s_{X_1}, \ldots, s_{X_n}, X_1, \ldots, X_n)$ *be a path formula with* $2 \times n$ *free path variables namely* $s_{X_1}, \ldots, s_{X_n}$ *called distinguished free path variables and* $X_1, \ldots, X_n$.
$\mu^+_{s_{X_1}, \ldots, s_{X_n}}(\varphi)$ *is called a fixpoint path expression of arity* n *and given a db-graph* G*, it denotes the* n*-ary relation that is the limit of the inflationary sequence* $\{\mathcal{S}_k\}_{k \geq 0}$ *defined by:*

1. $\mathcal{S}_0 = \{(\varepsilon, \ldots, \varepsilon)\}$ [3]
2. $\mathcal{S}_{k+1} = \mathcal{S}_k \cup \varphi(\mathcal{S}_k)$ *where*
   $\varphi(\mathcal{S}_k)$ *denotes the union of the evaluation of the path queries* $\{(X_1, \ldots, X_n) \mid \varphi[\nu_k]\}$ [4] *where* ν_k *is any valuation of the distinguished variables* $s_{X_1}, \ldots, s_{X_n}$ *induced by* $\mathcal{S}_k$.
   A valuation ν_k *is induced by* $\mathcal{S}_k$ *if there exists a tuple* $(v_1, \ldots, v_n)$ *in* $\mathcal{S}_k$ *such that* $\nu_k(s_{X_i}) = v_i$ *for* $i = 1..n$.

As usual, the n-ary fixpoint path expression $\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)$ can be used to build a more complex path formula. For instance, now $\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)(t_1, ..., t_n)$ where each t_i is a path term, can be considered as an atomic path formula. Nesting fixpoint operators does not raise any problem. The extension of the path calculus with the fixpoint construct μ^+ is called **Path-Fixpoint**.

In a straightforward way, a partial non inflationary extension of our path calculus can be provided. It suffices to define a fixpoint operator μ as in the above definition except for item 2. replaced by $\mathcal{S}_{k+1} = \varphi(\mathcal{S}_k)$. This extension is called **Path-While**.

Note that because of the initialization of the relation $\mathcal{S}_0$ the inflationist fixpoint of a path expression is never empty as it contains at least a tuple of empty paths. It may be convenient to get rid of this tuple in the limit of the sequence $\{\mathcal{S}_k\}_{k\geq 0}$. This could be achieved for instance by defining $\mathcal{S}_1$ as $\varphi(\mathcal{S}_0)$.

Example 8. The following three queries show how the Kleene closure can be expressed through a fixpoint path expression. All three queries are of the form $\{(\mathrm{X}) \mid \mu^+_{s_X}(\varphi)(\mathrm{X})\}$ and differ by the specification of φ.

1. φ is $(\exists \mathrm{x})(\exists \mathrm{y})$ $\mathrm{X} \in \mathrm{s}_X.\mathrm{x.y}$
2. φ is $\mathrm{X} \in \mathrm{s}_X.\mathrm{a.b}$
3. φ is $(\exists \mathrm{x})(\exists \mathrm{y})$ $(\mathrm{X} \in \mathrm{x.y} \vee (\mathrm{X} \in s_X.\mathrm{x.y} \wedge (\exists \mathrm{Y})\ s_X \in \mathrm{Y.x.y}))$

The first query returns paths of even length. This query can be expressed in most known languages using the Kleene closure of the path expression $(\#.\#)$ where $\#$ is the so-called *joker* symbol. The second query is equivalent to the Kleene closure $(a.b)^*$. The third query is a generalization of the Kleene closure $(a.b)^*$ where a and b are not constants.

Example 9. Below, the first (resp. second) query returns pairs of paths with identical sequence of edges (resp. vertices). Both queries are of the form $\{(\mathrm{X,Y}) \mid \mu^+_{s_X,s_Y}(\varphi)(\mathrm{X,Y})\}$ and differ by the specification of φ.

1. φ is $(\exists \mathrm{x})$ $(\mathrm{X} \in \mathrm{s}_X.\mathrm{x} \wedge \mathrm{Y} \in \mathrm{s}_Y.\mathrm{x})$
2. φ is $(\exists \mathrm{x})(\exists \mathrm{y})(\exists \alpha)(\exists \beta)$ $((\mathrm{X} \in \alpha \triangleleft \mathrm{x} \triangleright \beta \wedge \mathrm{Y} \in \alpha \triangleleft \mathrm{y} \triangleright \beta) \vee (\mathrm{X} \in \mathrm{s}_X.\mathrm{x} \triangleright \beta \wedge \mathrm{Y} \in \mathrm{s}_Y.\mathrm{y} \triangleright \beta))$

[3] $(\varepsilon, \ldots, \varepsilon)$ is the tuple of empty paths of arity n.
[4] We have taken some liberty with the writing of path queries here in order to simplify the presentation.

R

A	B
a	b
b	c

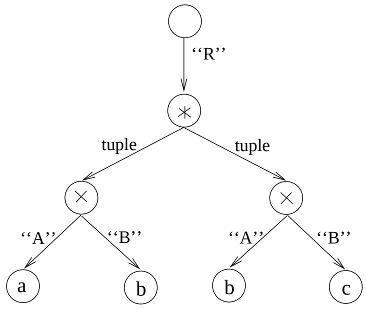

Fig. 4. Representation of a relational database by a db-graph.

Example 10. [**Equal length path**] The query $\{(X,Y) \mid \mu^{+}_{s_X,s_Y}(\mathbf{SL})(X,Y)\ \}$ where the path formula **SL** is defined below is rather interesting: it extracts pairs of paths having same length. The formula **SL** is:

$$(\exists \mathtt{x})(\exists \mathtt{y})\ (\mathtt{X}{\in}\mathtt{s}_X.\mathtt{x} \wedge \mathtt{Y}{\in}\mathtt{s}_Y.\mathtt{y})$$

Note that this query does not make use of any predefined counting predicate. This query illustrates how combining binary intentionally defined "predicate" with iteration increases the expressive power of path languages. This query cannot be expressed neither in Lorel, nor in UnQL. POQL can express it only by using the interpreted function length over paths. This query is rather important for sequence data. One can express a query that extracts all paths generated for example by the language $\{a^n b^n \mid n \in \mathbb{N}\}$ by using the fixpoint of the formula **SL**.

Example 11. [**Transitive closure**] The query $\{(X,Y) \mid \mu^{+}_{s_X,s_Y}(\mathbf{TC})(X,Y)\}$ where the path formula **TC** is defined below, computes (besides the pair of empty paths) pairs of paths whose respective pairs of destination is the transitive closure of a binary relation R represented in a naive way by a db-graph (see Figure 4). It proves that our path query language is powerful enough to express the transitive closure without requiring the creation of new edges as it is done in [12]. The formula **TC** is:

$$\begin{array}{l} \quad (\exists \mathtt{U}_1)\ (\mathtt{U}_1{\in}\text{"R"}.\mathtt{tuple} \wedge \mathtt{X}{\in}\ \mathtt{U}_1.\text{"A"} \wedge \mathtt{Y}{\in}\ \mathtt{U}_1.\text{"B"}) \\ \vee\ (\exists \mathtt{Z})(\exists \mathtt{U}_2)(\exists \alpha)\ \ (\mathtt{U}_2{\in}\text{"R"}.\mathtt{tuple} \wedge \mathtt{X}{\in}\ \mathtt{U}_2.\text{"A"} \wedge \mathtt{Z}{\in}\mathtt{U}_2.\text{"B"} \\ \qquad\qquad\qquad\quad \wedge\ \mathtt{Z}{\rhd}\alpha \wedge \mathtt{S_X}{\rhd}\alpha \wedge \mathtt{Y}{=}\mathtt{S_Y}) \end{array}$$

Example 12. [**Same generation**] Let us consider the theoretical computer science genealogy of `http://sigact.acm.org/genealogy/` Web page (see figure 5) that gives for each researcher some information such as his/her name, his/her adviser and his/her students. The path query $\{(X,Y) \mid \mu^{+}_{s_X,s_Y}(\mathbf{SG})(X,Y)\}$ where the path formula **SG** is defined below, computes pairs of researchers that have some common ancestor which is at the same number of generations away from

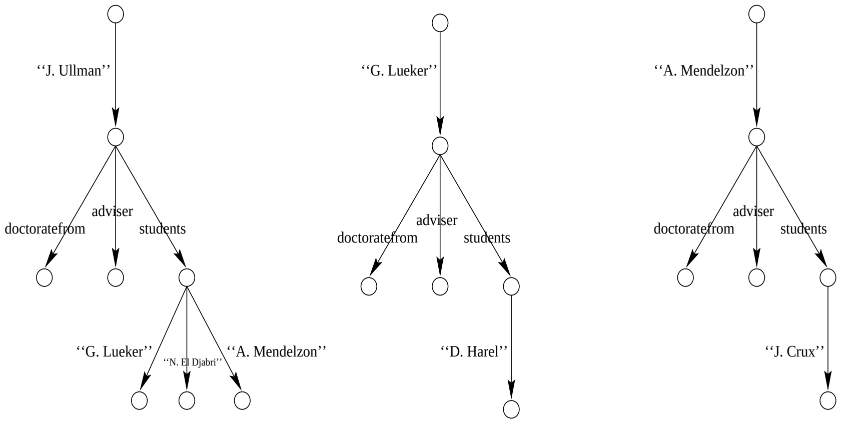

Fig. 5. A TCS genealogy.

each (some sort of cousins). The formula **SG** is:

$$
\begin{array}{ll}
 & (\exists \mathtt{u})\ (\mathtt{X.doctoratefrom} \wedge \mathtt{X}\in\mathtt{u} \wedge \mathtt{Y=X}) \\
\vee & (\exists \mathtt{Xa})(\exists \mathtt{Ya})(\exists \mathtt{x})(\exists \mathtt{y})(\exists \mathtt{u}_1)(\exists \mathtt{v}_1)(\exists \mathtt{u}_2)(\exists \mathtt{v}_2) \\
 & ((\mathtt{Xa.students.X} \wedge \mathtt{Xa}\in\mathtt{u}_1 \wedge \mathtt{X}\in\mathtt{v}_1) \\
 & \wedge\ (\mathtt{Ya.students.Y} \wedge \mathtt{Ya}\in\mathtt{u}_2 \wedge \mathtt{Y}\in\mathtt{v}_2) \\
 & \wedge\ (\mathtt{Xa}\in\mathtt{x} \wedge \mathtt{S}_\mathtt{X} \in\mathtt{x} \wedge \mathtt{Ya}\in\mathtt{y} \wedge \mathtt{S}_\mathtt{Y} \in\mathtt{y}))
\end{array}
$$

4 Graph calculus

Now that a path query language (Path-Fixpoint or Path-While) is available allowing us to specify some kind of navigation in a db-graph, it remains to explain how path sub-queries will be combined with graph constructs (and which graph constructs) in order to retrieve subgraphs (or their sources). Intuitively, a subgraph may be retrieved in two different ways: its source is the destination of some path specified by a path expression or its source is the origin of some path specified by a path expression. In fact, we also introduce a third way to select a (maximal) subgraph which checks if it is a path. This implies introducing an equality predicate symbol whose first operand is a graph variable and second operand is a path term[5]. This predicate is denoted = for sake of simplicity.

Definition 14. [**Atomic graph formula**] *An atomic graph formula is an expression of one of the following forms:*

[5] In a first version of the language, this predicate was not needed because path variables and graph variables were merged. However, this unique sort of variables (for paths and graphs) entailed a rather cumbersome definition of the graph calculus which is now avoided at the price of this special predicate.

1. *a Path-Fixpoint (or a Path-While) formula,*
2. $t : s$ *where* t *is a graph term (i.e. a graph variable or a rooted graph) and* s *is a path expression,*
3. $s[t]$ *where* s *is a path expression and* t *is a graph term,*
4. $t_1 \sim t_2$ *where* t_1 *and* t_2 *are both graph terms where* $\sim$ *is the bisimulation predicate symbol.*
5. $X = t$ *where* X *is a graph variable and* t *is a path term.*

Roughly speaking, $t : s$ tells that s spells at least one path p of the graph t and that the origin of p is the source of t; the intention of $s[t]$ is to check that the graph t has a source which is the destination of a path spelt by s.

A **graph formula** is defined in the usual way by introducing logical connectors and quantifications. We assume that graph formulas are normalized just as path formulas in order to avoid having both free and bound occurrences of the same variable. The notion of active domain of a db-graph G is extended in a straightforward manner.

Definition 15. [**Active domain**] *The active domain of a db-graph* G *denoted* $adom(G)$ *is the quadruple* $(\mathcal{G}_G, \mathcal{P}_G, \mathcal{L}_G, \mathcal{D}_G)$ *where* $(\mathcal{P}_G, \mathcal{L}_G, \mathcal{D}_G)$ *is exactly defined as in Definition 5 and* $\mathcal{G}_G$ *is the set Ext(G) of maximal rooted subgraphs of* G *defined in section 2.*

The notion of active domain of a graph formula φ extends in a simple way the notion of active domain of a path formula. When G and φ are clearly identified from the context, *adom* denotes the union of the active domains of G and φ. graph(*adom*) (resp. path(adom), label(adom) and data(adom)) denotes the first (resp. second, third and fourth) component of *adom*. The notion of a valuation is extended in a straightforward manner.

The definition of satisfaction of a graph formula is restricted to atomic graph formula. The general case does not present any difficulty and is omitted here.

Definition 16. [**Graph formula satisfaction**] *Let* φ *be a graph formula and let* ν *be a valuation of* $Free(\varphi)$. *The db-graph* G *satisfies* φ *under the valuation* ν *over adom, denoted* $G \models \varphi\,[\nu]$, *if*

1. φ *is a Path-Fixpoint (or Path-While) formula and* $G \models \varphi\,[\nu]$.
2. φ *is an entry* $t : s$ *and there exists a path* p *in* $Spell_{\nu(t)}(\nu(s))$ *such that* $org(p)$ *is (one of) the source of* $\nu(t)$.
3. φ *is a selector* $s[t]$ *and there exists a path* p *in* $Spell_G(\nu(s))$ *such that* $dest(p)$ *is (one of) the source of the graph* $\nu(t)$.
4. φ *is* $t_1 \sim t_2$ *and* $\nu(t_1) \sim \nu(t_2)$.
5. φ *is* $X = t$ *and the maximal subgraph* $\nu(X)=\nu(t)$.

This section concludes by defining graph queries and their evaluation.

Definition 17. [**Graph query**] *A graph query is an expression of the form*

$$\{X \mid \varphi\}$$

where $Free(\varphi)=\{X\}$.

In the definition above, we have restricted the free variable X of φ to be a graph variable because our aim is to return maximal subgraphs as answers to a graph query. The intuition behind is that a graph query is essentially expressed by specifying paths in the graph leading to some vertices of interest and the results of the query are the maximal graph rooted at these vertices i.e. all the information that can be reached from these vertices. Considering the Web example, this definition of an answer is not as unpractical as it may seem: the maximal subgraphs (the answers) can be synthesized by (one of) their source and represent all the information reachable from these sources. For simplicity, we have restricted φ to have a single free variable. The generalization to more than one free variable can be done easily.

Definition 18. [**Graph query answering**] *The image of a db-graph G under a graph query q is defined by*

$$q(G) = \{\nu(X) \mid \nu \text{ is a valuation of } Free(\varphi) \text{ and } G \models \varphi[\nu]\}$$

Thus considering a given graph G as its extension $\mathrm{Ext}(G)$ i.e. the set of all its maximal rooted subgraphs, a query defines a mapping from $\mathrm{Ext}(G)$ into $\mathrm{Ext}(G)$. The answer of a query may well be mapped to a db-graph, the "smallest" db-graph whose extension contains q(G). We have made the choice to develop a formal presentation of the language but it is clear that it may be useful for the purpose of specific applications to customize the outputs of answers. An obvious alternative would be to produce as an answer the set of the sources of the graphs in q(G) instead of q(G) itself.

The graph query language G-Fixpoint (based on Path-Fixpoint) is now illustrated through several examples.

Example 13. The following queries are examples of graph queries followed by a brief description of what each query does.

1. [**Identity query**] $\{$X $\mid$ ($\exists$U) (U=ϵ $\wedge$ U[X])$\}$
2. $\{$X $\mid$ tsimmis.publications[X]$\}$
3. $\{$X $\mid$ ($\exists$U)($\exists$V) "DB group"$\triangleleft$U.members.V.books[X]$\}$
4. $\{$X $\mid$ ($\forall$y) (($\exists$U) (members.y.U.publications[X]
   $\Rightarrow$ ($\exists$V) tsimmis.V.people.y))$\}$
5. $\{$X $\mid$ "R".tuple[X]$\}$
6. $\{$X $\mid$ ($\exists$U)($\exists$V) (U[X] $\wedge$ V[X] $\wedge$ $\neg$(U=V) $\wedge$ $\neg$(U=ϵ) $\wedge$ $\neg$(V=ϵ))$\}$
7. [**Cycles**] $\{$X $\mid$ ($\exists$U) (X:U $\wedge$ U[X] $\wedge$ $\neg$(U=ϵ)) $\}$
8. $\{$(X,Y) $\mid$ ($\exists$U)($\exists$V) ($\mu^{+}_{s_X,s_Y}$(**TC**)(U,V) $\wedge$ U[X] $\wedge$ V[Y]
   $\wedge$ $\neg$(U=ϵ) $\wedge$ $\neg$(V=ϵ))$\}$

The second query retrieves all publications of `tsimmis`. The third graph query retrieves the books of the Stanford database group members. The fourth query returns publications whose authors are all members of the Tsimmis project. The fifth query returns all tuples of a binary relation R (see Figure 4). The sixth query returns all maximal subgraph reached by two non identical paths. The eighth query illustrates how our language can be generalized to generate tuples of graphs as answers. This query returns the transitive closure of the binary relation R by selecting the extremities of pairs of paths returned by the fixpoint path query (Example 11) of course discarding the pair of empty paths.

5 Expressive power

In this section, we investigate the expressive power of Path-Fixpoint and Path-While. The two results presented here are complementary and provide an equivalence between Path-Fixpoint over db-graphs (resp. Path-While) and Fixpoint over relational instances (resp. While). However this equivalence hides the computation of simple paths (discussion about simple path computation can be found in [18]). In other words, db-graphs are translated into relational instances whose domain is primarily edge identifiers. In order to simulate (simple) paths, concatenation is provided through a pre-defined predicate *concat*.

The first result maps the semistructured data framework on a specific relational structure and shows that Path-Fixpoint (resp. Path-While) queries can be translated into Fixpoint (resp. While) queries over instances over this schema. The second result goes the other way around: relational instances are mapped on db-graphs and it is showed that Fixpoint (resp. While) queries over relational instances can be expressed by Path-Fixpoint (resp. Path-While) queries over the corresponding db-graphs.

Theorem 1. *There exists a translation $\mathcal{T}$ mapping db-graphs onto relational instances over a fixed schema $\mathcal{R}$ and mapping Path-Fixpoint (resp. Path-While) queries onto Fixpoint (resp. While) queries such that:*
if q_{path} is a Path-Fixpoint (resp. Path-While) query and G is a db-graph then

$$\mathcal{T}(q_{path}(G)) = (\mathcal{T}(q_{path}))(\mathcal{T}(G)).$$

Proof. The transformation $\mathcal{T}$ is outlined below. Db-graphs are translated into relational instances using a fixed relational schema composed of four relation schemas: the unary relation *edge* ($edge(X)$ meaning that X is an edge[6]); the binary relations *dataorig*, *datadest* ($dataorig(X, x)$, resp. $datadest(X, x)$, meaning that the data contents of the origin, resp. the destination, of the edge X is x); the binary relation *label* ($label(X, x)$ meaning that the label of the edge X is x). The transformation of a specific db-graph into an instance over this relational

[6] In fact, to be correct, we should have coded the edges of the graph using a ternary relation associating the origin node identity, the destination node identity and the edge identity.

schema is straightforward and omitted here.

For the purpose of translating path expressions and path formulas, we introduce a build-in predicate *concat*, the literal $concat(Z, X, Y)$ meaning that the path Z is simple and is the result of the concatenation of the simple paths X and Y.

A first order formula denoted $exp\text{-}s(Z)$ is associated to each path expression s. Intuitively, the formula $exp\text{-}s(Z)$ means that Z is a path which is spelt by the path expression s. The variable Z should be a "new" variable (not occurring in s).

Note that, in the proof, for sake of simplicity, we do not detail all restrictions that should naturally be taken while "choosing" variable names. Moreover, we abusively use the same notation for a path/label/data variable X in a path formula and its "translation" in a relational calculus formula.
The formula $exp\text{-}s(Z)$ is defined inductively as follows :

1. if s is X then $exp\text{-}s(Z)$ is $concat(X, Z, \epsilon)$ where ϵ is the empty path,
2. if s is t then $exp\text{-}s(Z)$ is $edge(Z) \wedge label(Z, t)$,
3. if s is $t \triangleleft s_1$ then $exp\text{-}s(Z)$ is $exp\text{-}s_1(Z) \wedge dataorig(Z, t)$,
4. if s is $s_1 \triangleright t$ then $exp\text{-}s(Z)$ is $exp\text{-}s_1(Z) \wedge datadest(Z, t)$,
5. if s is $s_1.s_2$ then
$exp\text{-}s(Z)$ is $(\exists X_1)(\exists X_2)\ (exp\text{-}s_1(X_1) \wedge exp\text{-}s_2(X_2) \wedge concat(Z, X_1, X_2))$.

The transformation $\mathcal{T}(\varphi)$ of path formula (without fixpoint) φ into a first order formula, is defined inductively.

1. if φ is a path expression s then $\mathcal{T}(\varphi)$ is $(\exists Z)\ exp\text{-}s(Z)$,
2. if φ is $t_1{=}t_2$ where t_1 and t_2 are path variables [7] then $\mathcal{T}(\varphi)$ is $concat(t_1, t_2, \epsilon)$,
3. if φ is $t_1{=}t_2$ where t_1 and t_2 are both label terms or data terms then $\mathcal{T}(\varphi)$ is $t_1{=}t_2$,
4. φ is $X \in s$ then $\mathcal{T}(\varphi)$ is $(\exists Z)\ (exp\text{-}s(Z) \wedge concat(X, Z, \epsilon))$,
5. if φ is $(\phi \wedge \psi)$ (resp. $(\phi \vee \psi)$, $\neg\phi$, $(\exists x)\ \phi$, and $(\forall x)\ \phi$) then $\mathcal{T}(\varphi)$ is $(\mathcal{T}(\phi) \wedge \mathcal{T}(\psi))$ (resp. $(\mathcal{T}(\phi) \vee \mathcal{T}(\psi))$, $\neg\mathcal{T}(\phi)$, $(\exists x)\ \mathcal{T}(\phi)$ and $(\forall x)\ \mathcal{T}(\phi)$).

The transformation $\mathcal{T}$ maps a fixpoint path expression $\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)$ into a relational fixpoint expression $\mu^+_T(\psi(T))$ where $\psi(T)$ is defined by:

$$(concat(X_1, \epsilon, \epsilon) \wedge \ldots \wedge concat(X_n, \epsilon, \epsilon))$$
$$\vee\ (\exists s_{X_1})\ldots(\exists s_{X_n})\ (\mathcal{T}(\varphi) \wedge T(s_{X_1}, \ldots, s_{X_n}))$$

Example. The transformation of the fixpoint path expression $\mu^+_{s_X}(\varphi)$ where φ is $X \in s_X.a$ is the relational fixpoint expression $\mu^+_T(\psi(T))$ where $\psi(T)$ is

[7] If t_1 or t_2 or both t_1 and t_2 are ground paths, the translation is a little more complex although it does not entails difficulties.

$$concat(X, \epsilon, \epsilon) \vee (\exists s_X)\ (\mathcal{T}(\varphi) \wedge T(s_X))$$

and $\mathcal{T}(\varphi)$ is

$$\begin{aligned}(\exists Z)[\ (\ (\exists X_1)(\exists X_2)\ & concat(s_X, X_1, \epsilon) \wedge edge(X_2) \wedge label(X_2, a) \\ & \wedge concat(Z, X_1, X_2)\ ) \wedge concat(X, Z, \epsilon)\ ]\end{aligned}$$

□

The use of the build-in predicate *concat* in the above proof is hiding the computation of simple paths. This entails that our translation is not purely relational. This cannot be otherwise because some of our path queries are exponential as (pure) relational queries are polynomialy computable.

Note also that given a relational query over the relational schema given in the above proof, it is not always possible to find an equivalent path query. This is essentially due to the "pointwise" flavor of our path fixpoint operator to be opposed to the "globality" of the relational fixpoint operator.

We now investigate the inverse translation, from relational queries to path queries.

Theorem 2. *There exists a translation $\mathcal{T}$ mapping instances over any relational schema onto db-graphs and mapping Fixpoint (resp. While) queries onto Path-Fixpoint (resp. Path-While) queries such that:*

if q_{rel} is a Fixpoint (resp. While) query and r is an instance over a relational schema then

$$\mathcal{T}(q_{rel}(r)) = (\mathcal{T}(q_{rel}))(\mathcal{T}(r)).$$

Proof. The transformation of an instance r over a relation schema $\mathcal{R}$ into a db-graph is summarized by the example of figure 4. In order to translate a relational query $\{(X_1, \ldots, X_n) \mid \varphi\}$ into a path query, we need to capture the active domain of φ. We chose to do that by augmenting the db-graph $\mathcal{T}(r)$ by $\mathcal{T}(const)$ where *const* is an instance over a unary relation schema CONST(ATT) collecting the constant occurring in φ.

Example. Let us consider the relational instance of figure 4 and let the relational query be $\{(X, Y) \mid R(X, Y) \wedge \neg(X = \text{“}e\text{”}) \wedge \neg(Y = \text{“}f\text{”})\}$. The db-graph of figure 4 representing this instance is augmented by the graph representing the instance {const(e),const(f)}.

Like in the previous proof, for sake of simplicity, we do not detail, all restrictions that should naturally be taken while "choosing" variable names and we abusively use the same notation for a variable X in a first order formula and its "translation" in a path expression or formula.
The transformation of a relational query $q_{rel} = \{(X_1, \ldots, X_n) \mid \varphi\}$ is the path query defined by:

$$\{(X_1, \ldots, X_n) \mid radom(X_1) \wedge \ldots \wedge radom(X_n) \wedge \mathcal{T}(\varphi)\}$$

The path formula $radom(X)$ is meant to capture the domain of the relational instance and is defined by : $(\exists v)\ X \in tuple.v$.

The path formula $\mathcal{T}(\varphi)$ which is the transformation of a first order formula φ is defined inductively as follows:

1. if φ is $R(t_1, \ldots, t_n)$ and R is a relation schema whose attributes are A_1 ... A_n then $\mathcal{T}(\varphi)$ is $(\exists U)\ (U \in \text{“}R\text{”}.tuple \wedge \phi_1 \wedge \ldots \wedge \phi_n)$ where
$$\phi_i = \begin{cases} (\exists \alpha)\ (t_i \triangleright \alpha \wedge (\exists X)\ (X \in U.A_i \wedge X \triangleright \alpha)) & if\ t_i\ is\ a\ variable \\ (\exists X)\ (X \in U.A_i \wedge X \triangleright a) & if\ t_i\ is\ the\ constant\ a \end{cases}$$
2. if φ is $t_1{=}t_2$ and t_1 is variable and t_2 is constant then $\mathcal{T}(\varphi)$ is $t_1 \triangleright t_2$,
3. if φ is $t_1{=}t_2$ and t_1 and t_2 are variables then $\mathcal{T}(\varphi)$ is $(\exists\alpha)(t_1 \triangleright \alpha \wedge t_2 \triangleright \alpha)$,
4. if φ is $(\phi \wedge \psi)$ (resp. $(\phi \vee \psi)$) then $\mathcal{T}(\varphi)$ is $(\mathcal{T}(\phi) \wedge \mathcal{T}(\psi))$ (resp. $(\mathcal{T}(\phi) \vee \mathcal{T}(\psi))$),
5. if φ is $\neg\phi$ then $\mathcal{T}(\varphi)$ is $\neg\mathcal{T}(\phi)$,
6. if φ is $(\exists x)\ \phi$ then $\mathcal{T}(\varphi)$ is $(\exists x)\ (radom(x) \wedge \mathcal{T}(\phi))$,
7. if φ is $(\forall x)\ \phi$ then $\mathcal{T}(\varphi)$ is $(\forall x)\ (\neg radom(x) \vee \mathcal{T}(\phi))$.

The transformation of a relational fixpoint expression $\mu^{+}_{T}(\varphi(T))$ where Free(φ)= $\{X_1, \ldots, X_n\}$ is the fixpoint path expression
$$\mu^{+}_{s_{X_1},\ldots,s_{X_n}}(radom(X_1) \wedge \ldots \wedge radom(X_n) \wedge \mathcal{T}(\varphi))$$
where $\mathcal{T}$ is the transformation defined above augmented by:

8. if φ is $T(t_1, \ldots, t_n)$ then $\mathcal{T}(\varphi)$ is $\phi_1 \wedge \ldots \wedge \phi_n$ where
$$\phi_i = \begin{cases} (\exists \alpha)\ (s_{X_i} \triangleright \alpha \wedge t_i \triangleright \alpha \wedge \neg(s_{X_i} = \epsilon)) & if\ t_i\ is\ a\ variable \\ s_{X_i} \triangleright a \wedge \neg(s_{X_i} = \epsilon) & if\ t_i\ is\ the\ constant\ a \end{cases}$$

Example. Let us consider the relational fixpoint expression $\mu^{+}_{good}(\varphi(T))$ where φ is $(\forall X)\ (\neg R(X,Y) \vee good(X))$, that computes the set of nodes in the relation R that are not reachable from a cycle. The associated path fixpoint expression is $\mu^{+}_{s_Y}(\psi)$ where ψ is
$$\begin{aligned} &(\exists u)\ Y \in tuple.u \\ \wedge\ &(\forall X)\ (\neg\ ((\exists u)\ X \in tuple.u) \\ &\quad \vee\ (\neg(\ (\exists U)\ U \in \text{“}R\text{”}.tuple \\ &\qquad \wedge\ (\exists \alpha_1)\ (X \triangleright \alpha_1\ \wedge\ (\exists X_1)\ (X_1 \in U.\text{“}A\text{”}\ \wedge\ X_1 \triangleright \alpha_1)) \\ &\qquad \wedge\ (\exists \alpha_2)\ (Y \triangleright \alpha_2\ \wedge\ (\exists Y_1)\ (Y_1 \in U.\text{“}B\text{”}\ \wedge\ Y_1 \triangleright \alpha_2))) \\ &\quad\ \vee\ (\exists \alpha)\ (s_Y \triangleright \alpha\ \wedge\ X \triangleright \alpha\ \wedge\ \neg(s_Y{=}\epsilon)))) \end{aligned}$$

□

Acknowledgments

This work was partially supported by the GDR I3. We thank the anonymous reviewers for their valuable remarks and suggestions.

References

1. Abiteboul, S. Querying semistructured Data. In *ICDT* (1997), pp. 1–18.
2. Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases*. Addision-Wesley, 1995.
3. Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.
4. Abiteboul, S., and Vianu, V. Queries and Computation on the Web. In *ICDT* (1997).
5. Abiteboul, S., and Vianu, V. Regular Path Queries with Constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Tucson, Arizona, 12–15 May 1997).
6. Amann, B., and Scholl, M. Gram : A graph data model and query language. In *4th ACM Conf. on Hypertext and Hypermedia* (1992), pp. 138–148.
7. Buneman, P. Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Tucson, Arizona, 12–15 May 1997), pp. 117–121.
8. Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. A query language and optimization techniques for unstructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data) 25*, 2 (1996).
9. Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. From Structured Documents to Novel Query Facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, 24–27 May 1994), R. T. Snodgrass and M. Winslett, Eds., pp. 313–324.
10. Consens, M., and Mendelzon, A. GraphLog: a Visual Formalism of Real Life Recursion. In *Proc. 9th ACM Symp. on Principles of Database Systems* (1990), pp. 404–416.
11. Consens, M. P., and Mendelzon, A. O. Expressing Structural Hypertext Queries in GraphLog. In *ACM Hypertext'89 Proceedings* (1989), Information Retrieval II, pp. 269–292.
12. Fernandez, M., Florescu, D., Levy, A., and Suciu, D. A Query Language for a Web-Site Management System. *SIGMOD Record (ACM Special Interest Group on Management of Data) 26*, 3 (1997), 4–11.
13. Frohn, J., Lausen, G., and Uphoff, H. Access to Objects by Path Expressions and Rules. In *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings* (Los Altos, CA 94022, USA, 1994), J. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann Publishers, pp. 273–284.
14. Konopnicki, D., and Shmueli, O. W3QS: A Query System for the World-Wide Web. In *VLDB* (1995), pp. 54–65.
15. Lakshmanan, L. V. S., Sadri, F., and Subramanian, I. N. A declarative language for querying and restructuring the Web. In *Sixth International Workshop on Research Issues in Data Engineering: interoperability of non traditional database systems: proceedings, February 26–27, 1996, New Orleans, Louisiana* (1996), IEEE, Ed., IEEE Computer Society Press, pp. 12–21.
16. Mendelzon, A. O., Mihaila, G. A., and Milo, T. Querying the World Wide Web. *International Journal on Digital Libraries 1*, 1 (1997), 54–67.
17. Mendelzon, A. O., and Milo, T. Formal Models of Web Queries. In *Proceedings of the sixteenth ACM Symposium on Principles of Database systems* (1997), pp. 134–143.

18. Mendelzon, A. O., and Wood, P. T. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing 24*, 6 (December 1995), 1235–1258.
19. Milner, R. *Communication and concurrency.* Prentice Hall, 1989.
20. Quass, D., Rajaraman, A., Sagiv, Y., and Ullman, J. Querying Semistructured Heterogeneous Information. *Lecture Notes in Computer Science 1013* (1995), 319–344.
21. Van den Bussche, J., and Vossen, G. An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries. *Lecture Notes in Computer Science 760* (1993), 267–282.