Theoretical Computer Science 78 (1991) 217-244
North-Holland

Recursive queries and context-free graph grammars*

Bruno Courcelle

Université Bordeaux-1, Laboratoire d'Informatique, Unité de Recherche Associée au CNRS, 351 cours de la Libération, 33405 Talence, France

Abstract

Courcelle, B., Recursive queries and context-free graph grammars, Theoretical Computer Science 78 (1991) 217-244.

The semantics of a recursive query in the context of a relational database can be formalized in terms of a context-free graph grammar associated with it. This grammar generates oriented, labeled hypergraphs. The labels of the hyperedges of these hypergraphs, called the computation graphs of the query, are the names of the relations forming the database under consideration. Some properties of recursive queries expressible as monadic second-order properties of their computation graphs are decidable. A few examples are given.

Introduction

We consider a *relational database* as a finite set A of relation symbols. Each of these symbols has a fixed arity. An *interpretation*, i.e., a possible state of the database, consists of a domain and of relations on this domain of appropriate arity that give a meaning to the symbols of A.

A *recursive query* is a relation defined by a set of function-free Horn clauses, written with the relation symbols of A. Recursion may be used in such definitions. A typical example is the transitive closure of a binary relation. The value of the query in a given interpretation is a relation, defined in terms of the least solution of a system of equations over relations, associated with the Horn clauses. This way of interpreting recursive queries stems from the work of Kowalski and Van Emden [11], who define in this way, the semantics of Horn clauses in a slightly different context.

An oriented graph can be considered as the representation of a binary relation on a set, namely the set of vertices of the graph. By attaching labels to edges, one can represent several binary relations on the same set in this way. Unary relations

* This work has been supported by the Metheol project of the "Programme de Recherches Coordonnées: Programmation Avancée et Outils Pour l'Intelligence Artificielle".

can be represented by labels attached to vertices, and relations of arity larger than 2, by oriented hyperedges, that is, by edges having a sequence of vertices instead of just a pair. Hence, we shall use hypergraphs in this paper. They will be used as syntactic representations of relations defined by conjunctions of literals, written with the base relation symbols of the set A. Conversely, every hypergraph gets a semantics in this way. Graphs have been used already as representations of conjunctions of literals by Gardarin et al. [9]. They are called *connection graphs*, and they are used as a tool for transforming certain recursive queries into systems of functional equations. In order to simplify the terminology, we shall use "graph" and "edge" instead of "hypergraph" and "hyperedge", respectively.

A *context-free graph grammar* is associated with a recursive query. The basic rewriting step of this grammar is the replacement of an edge by the connection graph corresponding to the right-hand side of a defining clause. The generated graphs have their (hyper)edges labeled by the symbols of A. They are the *computation graphs* of the query.

Our main result states that the semantics of the query in a given interpretation can be defined from the set of computation graphs. Hence, as in the theory of program schemes, we have a syntactic object representing all possible computations in all possible interpretations. This object is here the set of computation graphs, whereas in the case of program schemes it is an infinite tree.

It has been proved in [3, 6, 7] that the monadic second-order theory of a context-free set of graphs is decidable. It follows that properties of recursive queries expressible as monadic second-order properties of their computation graphs are decidable. A few applications are given.

A first version of this work, including also applications to recursive applicative program schemes, has appeared in [8].

1. Preliminaries

Notation 1.1. We denote by $\mathbb{N}$ the set of nonnegative integers, and by $\mathbb{N}_+$, the set of positive integers. We denote by $[n]$ the interval $\{1, 2, 3, \ldots, n\}$ for $n \geqslant 0$, with $[0] = \emptyset$.

For sets A and B, we denote by A-B the set $\{a \in A \mid a \notin B\}$.

The domain of a partial mapping $f: A \to B$ is denoted by $\mathbf{Dom}(f)$. The restriction of f to a subset A' of A is denoted by $f \restriction A'$. The partial mapping with an empty domain is denoted by $\emptyset$, as the empty set. If two partial mappings $f: A \to B$ and $f': A' \to B$ coincide on $\mathbf{Dom}(f) \cap \mathbf{Dom}(f')$, we denote by $f \cup f'$ their common extension into a partial mapping: $A \cup A' \to B$, with domain $\mathbf{Dom}(f) \cup \mathbf{Dom}(f')$.

The powerset of a set A is denoted by $\mathscr{P}(A)$.

The set of nonempty sequences of elements of a set A is denoted by A^+, and sequences are denoted by $(a_1, \ldots, a_n)$ with commas and parentheses. The empty sequence is denoted by ( ), and A^* is $A^+ \cup \{(\,)\}$. When A is an alphabet, i.e., when

its elements are letters, then a sequence $(a_1, \ldots, a_n)$ in A^+ can be written unambiguously $a_1 a_2 \ldots a_n$. The empty sequence is denoted by ε, a special symbol reserved for this purpose. The elements of A^* are called words. The length of a sequence w is denoted by $|w|$.

The symbol $:=$ means "equal by definition", and is used to introduce notations. The symbol $:\Leftrightarrow$ is used similarily to define logical properties.

We shall use the following version of the *fixed point lemma*. See Lassez et al. [12] for a thorough discussion of its variants and their respective paternities.

Lemma 1.2. *Let $(E, \leqslant)$ be a complete partial order with least element $\perp_E$. Let $f: E \to E$ be a monotone and continuous mapping. Then, the element of E defined by $e_0 := \sup\{f^i(\perp_E) \mid i \geqslant 0\}$ is the least solution in E of the equation $x = f(x)$. It is also the least solution of the inequation $x \geqslant f(x)$.*

This element e_0 (that does exist since the sequence $f^i(\perp_E)$, $i \geqslant 0$ is increasing) is also called the least fixed point of f, and is denoted by $\mu x.f(x)$.

The following lemma is basically due to Mezei and Wright [13, Lemma 5.3]. (See also [2, p. 30, Lemma 5.3].)

Lemma 1.3. *Let (E, f) and (E', f') be as (E, f) in Lemma 1.2. If $h: E \to E'$ is a monotone and continuous mapping such that:*

(i) $h(\perp_E) = \perp_{E'}$,

(ii) $f' \circ h = h \circ f$,

then $h(\mu x.f(x)) = \mu x.f'(x)$.

2. Graphs and context-free graph grammars

We shall use the labeled, oriented hypergraphs, equipped with a sequence of distinguished vertices introduced in Bauderon and Courcelle [5, 6]. In order to shorten the statements, we shall simply call these hypergraphs *graphs*, and their hyperedges *edges*.

An edge has a *sequence* of vertices (and not merely a set) and a label chosen in a ranked alphabet, i.e., in a set A, each element of which has an associated nonnegative integer that we shall call its *type*. The type is defined by a mapping $\tau: A \to \mathbb{N}$. The type of the label of an edge must be equal to the length of its sequence of vertices. Here is a formal definition.

Definition 2.1 (*Graphs*). *A concrete graph* is a quintuple $G = \langle \mathbf{V}_G, \mathbf{E}_G, \mathbf{lab}_G, \mathbf{vert}_G, \mathbf{src}_G \rangle$ where:

- $\mathbf{V}_G$ is a set, the set of vertices of G;
- $\mathbf{E}_G$ is a set, the set of its edges;
- $\mathbf{lab}_G: \mathbf{E}_G \to A$ is a total mapping defining the label of an edge;

- $\mathbf{vert}_G$ is a total mapping defining the sequence of vertices $\mathbf{vert}_G(e)$ of an edge e; its length must be equal to $\tau(\mathbf{lab}_G(e))$; we shall denote by $\mathbf{vert}_G(e, i)$ the ith element of the sequence $\mathbf{vert}_G(e)$; the same vertex can occur several times in $\mathbf{vert}_G(e)$; the integer $\tau(\mathbf{lab}_G(e))$ is called the *type* of the edge e, and will also be denoted by $\tau(e)$; we say that $\mathbf{vert}_G(e, i)$ *belongs* to the edge e, and that e is *incident* to $\mathbf{vert}_G(e, i)$.

Finally, $\mathbf{src}_G$ is a finite sequence in $\mathbf{V}_G$, or equivalently, a mapping: $[n] \to \mathbf{V}_G$ for some $n \geqslant 0$. Hence, $\mathbf{src}_G(i)$ denotes the ith element of the sequence $\mathbf{src}_G$. It is called a *source*. If $n = 0$, then G has no source. "Source" is just an easy sounding word for "distinguished vertex". There is no notion of flow involved.

Note that if a is of type 0, then an edge labeled by a has no vertex. We shall see that such edges are useful.

An *internal* vertex of G is a vertex that does not appear in the sequence $\mathbf{src}_G$. A vertex is *isolated* if it does not belong to any edge (even of type 1).

The integer n is called the *type* of G. We also say that G is a *concrete n-graph*. Whenever we need to specify the alphabet A, we say that G is a *concrete graph over A*. We denote by $\mathbf{CG}(A)_n$ the set of concrete n-graphs over A.

Let H and K be concrete n-graphs. A *homomorphism* $h: H \to K$ is a pair of mappings $(h_{\mathbf{V}}, h_{\mathbf{E}})$ such that:

$$h_{\mathbf{V}}: \mathbf{V}_H \to \mathbf{V}_K,$$

$$h_{\mathbf{E}}: \mathbf{E}_H \to \mathbf{E}_K,$$

$$\mathbf{lab}_H = \mathbf{lab}_K \circ h_{\mathbf{E}},$$

$$\mathbf{vert}_K(h_{\mathbf{E}}(e), i) = h_{\mathbf{V}}(\mathbf{vert}_H(e, i)), \quad \text{for all } e \in \mathbf{E}_H,\ i \in [\tau(e)],$$

$$\mathbf{src}_K = h_{\mathbf{V}} \circ \mathbf{src}_H.$$

If $h_{\mathbf{V}}$ and $h_{\mathbf{E}}$ are bijections, then h is an *isomorphism*.

We define a *graph* (resp. an n-graph over A) as the equivalence class of a concrete graph (resp. of a concrete n-graph over A) with respect to isomorphism. We denote by $\mathbf{G}(A)$ (resp. by $\mathbf{G}(A)_n$) the set of all graphs (resp. of all n-graphs) over A. The corresponding sets of finite graphs are denoted by $\mathbf{FG}(A)$ and $\mathbf{FG}(A)_n$. In order to emphasize that a graph is not a concrete graph, but an isomorphism class of concrete graphs, we shall call it an *abstract graph*.

If G and H are two concrete graphs, we say that G is a subgraph of H, and we denote this by $G \subseteq H$, if the following conditions hold:

$$\mathbf{V}_G \subseteq \mathbf{V}_H,$$

$$\mathbf{E}_G \subseteq \mathbf{E}_H,$$

$$\mathbf{lab}_G = \mathbf{lab}_H \restriction \mathbf{E}_G,$$

$$\mathbf{vert}_G = \mathbf{vert}_H \restriction \mathbf{E}_G,$$

and $\mathbf{src}_G$ is the sequence $\mathbf{src}_H$ from which the vertices of $\mathbf{V}_H$-$\mathbf{V}_G$ have been deleted. It follows that the type of G is at most that of H.

We denote by G^0 the graph H such that $\mathbf{src}_H = (\,)$ and everything else in H is as in G.

Example 2.2. (1) Let $n \geqslant 0$. The graph $\mathbf{n}$ is the graph G such that $\mathbf{V}_G = [n]$, $\mathbf{E}_G = \emptyset$, $\mathbf{lab}_G = \emptyset$, $\mathbf{vert}_G = \emptyset$, $\mathbf{src}_G$ is the sequence $(1, 2, \ldots, n)$. In particular we have the *empty* graph $\mathbf{0}$ which is (necessarily) of type 0.

(2) If b is an element of A of type n, then b also denotes the graph G with a single edge labeled by b, and defined by $\mathbf{V}_G = [n]$, $\mathbf{E}_G = \{*\}$, $\mathbf{lab}_G(*) = b$, $\mathbf{vert}_G(*) = \mathbf{src}_G = (1, 2, \ldots, n)$. Note that b is reduced to an edge with no vertex in the special case where $n = 0$.

(3) A less trivial example is the 3-graph G such that $\mathbf{V}_G = \{u, w, x, y, z\}$, $\mathbf{E}_G$ consists of 9 edges labeled by a, b, c, d, such that $\tau(a) = 1$, $\tau(b) = 2$, $\tau(c) = 3$, $\tau(d) = 0$. The sequence $\mathbf{src}_G$ is (u, y, u).

In the drawing of this graph (Fig. 1), we have used the following conventions:

- binary edges are represented as usual;
- unary edges are labels attached to vertices (a vertex may belong to several unary edges);

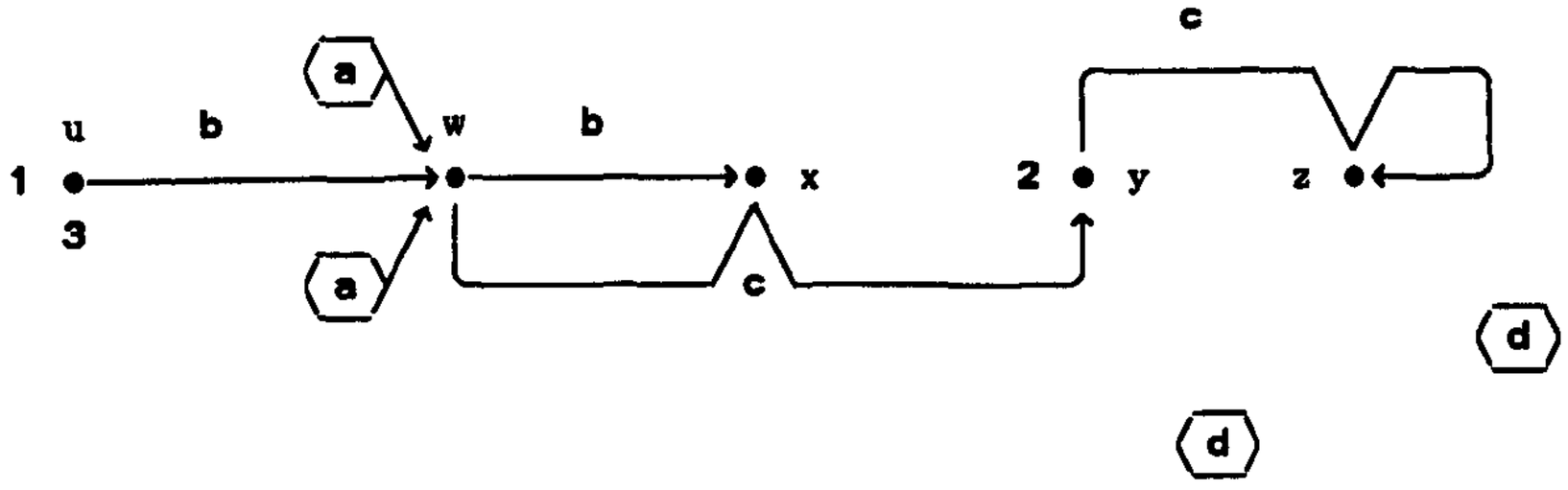


Fig. 1.

- edges of rank greater than 2 are represented as edges with intermediate nodes. In Fig. 1, there is an edge e such that $\mathbf{vert}_G(e) = (w, x, y)$, and another one, e', with $\mathbf{vert}_G(e') = (y, z, z)$.
- nullary edges are represented as boxes floating around.
  The positions of the source vertices in $\mathbf{src}_G$ are indicated by integers.

The notion of context-free rewriting is based on an appropriate notion of substitution: a graph of type n can be substituted for an edge of type n in a graph.

Definition 2.3 (*Graph substitutions*). Let G be a concrete graph; let e be an edge of G of type $p \geqslant 0$. Let H be a concrete p-graph. We denote by $G[H/e]$ the concrete

graph K defined as follows, by means of an isomorphic copy of H (also denoted by H), such that $\mathbf{E}_H \cap \mathbf{E}_G = \emptyset$, $\mathbf{V}_H \cap \mathbf{V}_G = \emptyset$. We define K by letting:

$$\mathbf{E}_K := \mathbf{E}_G \cup \mathbf{E}_H - \{e\},$$

$$\mathbf{V}_K := V/\simeq,$$

where $V = \mathbf{V}_G \cup \mathbf{V}_H$ and $\simeq$ is the equivalence relation on V generated by $\{(\mathbf{vert}_G(e, i), \mathbf{src}_H(i)) \mid i = 1, \ldots, p\}$.

Then, letting f denote the canonical mapping $V \to V/\simeq$, we define

$$\mathbf{vert}_K := ((f \circ \mathbf{vert}_G) \cup (f \circ \mathbf{vert}_H)) \restriction \mathbf{E}_K,$$

$$\mathbf{lab}_K := (\mathbf{lab}_G \cup \mathbf{lab}_H) \restriction \mathbf{E}_K,$$

$$\mathbf{src}_K := f \circ \mathbf{src}_G.$$

It is clear that substitutions commute with isomorphisms. Hence, substitutions can be defined for abstract graphs.

Definition 2.4 (*Context-free graph grammars*). *A context-free graph grammar* is a triple $\Gamma = \langle A, U, P \rangle$ consisting of two finite ranked alphabets A and U, called respectively the *terminal* and the *nonterminal* one (the rank function in $\tau : A \cup U \to \mathbb{N}$), of a finite set P of pairs of the form (u, G) (also denoted by $u \to G$), with $G \in \mathbf{FG}(A)_{\tau(u)}$ and $u \in U$. Each of these pairs is called a *production rule.*

A binary relation on $\mathbf{FG}(A \cup U)$ is associated with Γ as follows: $H \to_\Gamma H'$ if there exists an edge e of H labeled by u, such that $H' = H[G/e]$ for some $(u, G) \in P$.

Note that if $H \to_\Gamma H'$, then H and H' are of the same type.

If $K \in \mathbf{FG}(A \cup U)$, then the set of graphs *generated by* Γ *from* K is

$$\mathbf{L}(\Gamma, K) := \{H \in \mathbf{FG}(A) \mid K \xrightarrow[\Gamma]{*} H\}.$$

Such a set is called a *context-free* set of graphs.

If Γ is given with an axiom Z in $\mathbf{FG}(A \cup U)$, i.e., as a quadruple $\Gamma = \langle A, U, P, Z \rangle$, then we denote by $\mathbf{L}(\Gamma)$ the set $\mathbf{L}(\Gamma, Z)$, and we call it the set of graphs *generated by* Γ. Note that every u in U is also a graph by Example 2.2(2). Hence, the notation $\mathbf{L}(\Gamma, u)$ is meaningful.

In the present paper, grammars will be given without axioms, and we shall be interested in the sets of graphs $\mathbf{L}(\Gamma, u)$ for the various nonterminals u.

Here is a grammar generating the set of oriented series-parallel graphs. Let A consist of one symbol a, of type 2. The context-free graph grammar shown in Fig. 2 generates the set **SP** of series-parallel graphs with oriented edges, all labeled by a. A graph generated by this grammar is also shown in Fig. 2, to the right.

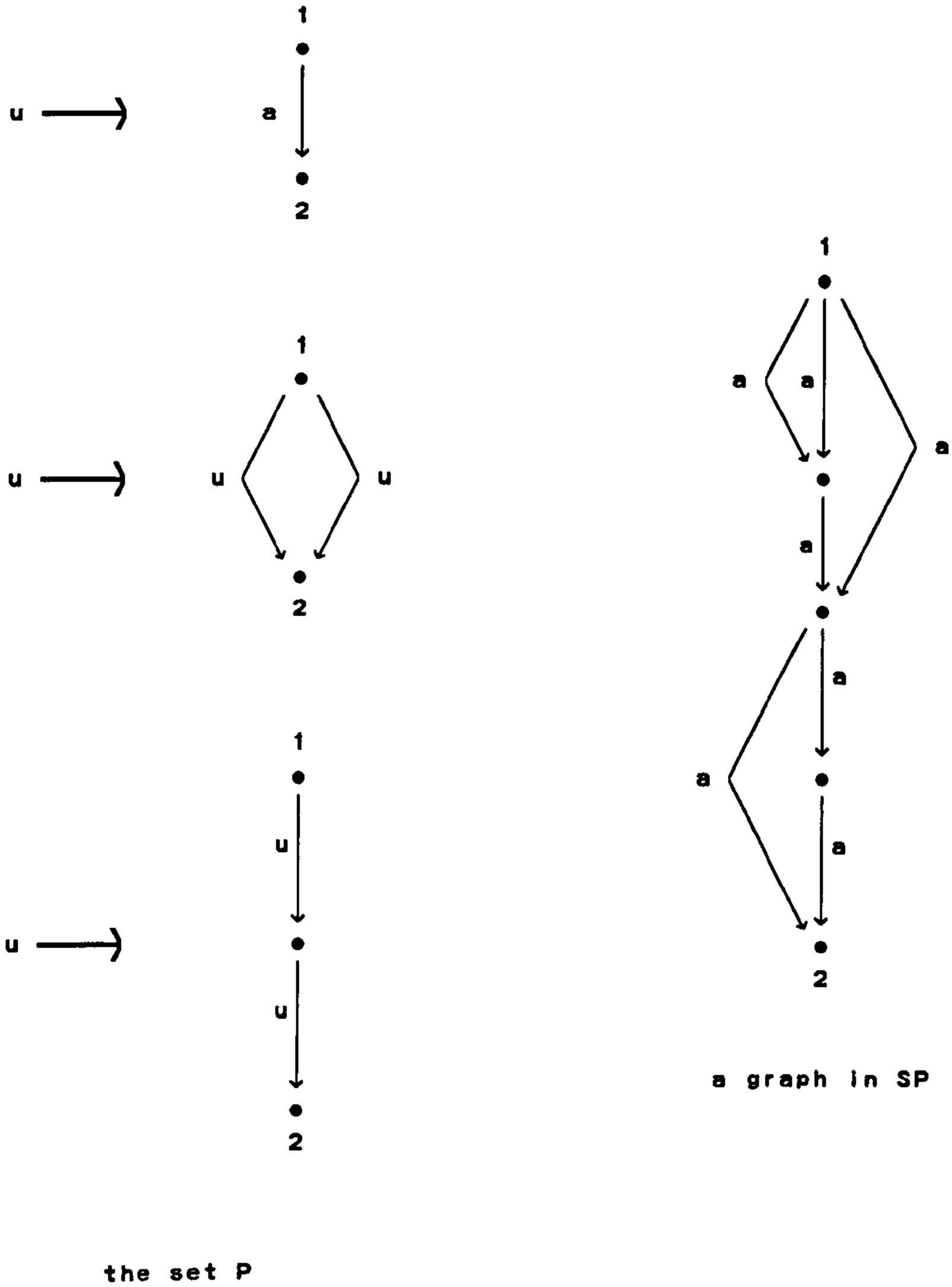


Fig. 2.

We shall recall the fixed-point characterization of the sets $\mathbf{L}(\Gamma, u)$, $u \in U$ given in [1, 6]. We first extend the substitution operation defined in Definition 2.3.

Definition 2.5. Let H be a concrete graph with edges $e_1, \ldots, e_k$ of respective types $n_1, \ldots, n_k$. Let $G_1, \ldots, G_k$ be concrete graphs of respective types $n_1, \ldots, n_k$. Then the result of the successive substitution of G_1 for $e_1, \ldots, G_k$ for e_k in H is the concrete graph $H[G_1/e_1][G_2/e_2]\ldots[G_k/e_k]$, also denoted by $H[G_1/e_1, \ldots, G_k/e_k]$. It can be proved that, for every permutation π of $[k]$, the graphs $H[G_1/e_1, \ldots, G_k/e_k]$ and $H[G_{\pi(1)}/e_{\pi(1)}, \ldots, G_{\pi(k)}/e_{\pi(k)}]$ are isomorphic. Hence, in the definition of $H[G_1/e_1, \ldots, G_k/e_k]$, the ordering of $\{e_1, \ldots, e_k\}$ is

irrelevant. We consider this graph as the result of the *simultaneous substitution* of G_1 for $e_1, \ldots, G_k$ for e_k in H.

If $a_1, \ldots, a_k$ are edge labels, and if $G_1, \ldots, G_k$ are of respective types $\tau(a_1), \ldots, \tau(a_k)$, then, we denote by $H[G_1/a_1, \ldots, G_k/a_k]$ the result of the simultaneous substitution of G_i for all edges labeled by a_i, for all $i = 1, \ldots, k$. If some label a_i has no occurrence in H, then, the corresponding graph G_i is not substituted for any edge of H. This definition applies to abstract graphs in an obvious way, because isomorphisms commute with substitutions.

We now extend substitution so as to substitute sets of graphs. We let $a_1, \ldots, a_k$ be as above, and we let

$$\mathcal{G}_1 \subseteq \mathbf{G}(A)_{\tau(a_1)}, \ldots, \mathcal{G}_k \subseteq \mathbf{G}(A)_{\tau(a_k)}.$$

We let H be a concrete graph. Then, we let $H[\mathcal{G}_1/a_1, \ldots, \mathcal{G}_k/a_k]$ be the set of graphs of the form $H[G_1/e_1, \ldots, G_n/e_n]$ where $e_1, \ldots, e_n$ is an enumeration of the set of edges of H having a label in $\{a_1, \ldots, a_k\}$, and, for each $i = 1, \ldots, n$, the graph G_i belongs to $\mathcal{G}_j$, where a_j is the label of e_i. Again, this definition extends to abstract graphs in a standard way.

Definition 2.6. With a context-free graph grammar $\Gamma = (A, U, P)$, we associate the system $\bar{\Gamma}$ consisting of the following equations (where $U = \{u_1, \ldots, u_m\}$):

$$\mathcal{G}_i = \bigcup \{G[\mathcal{G}_1/u_1, \ldots, \mathcal{G}_m/u_m] \mid (u_i, G) \in P\}, \quad 1 \leqslant i \leqslant m.$$

In this system, $\mathcal{G}_i$ denotes a subset of $\mathbf{G}(A)_{\tau(u_i)}$. This system is fully analogous to the system of equations on languages that is classically associated with a context-free grammar. Solving $\bar{\Gamma}$ consists of finding in $E := \mathcal{P}(\mathbf{G}(A)_{\tau(u_1)}) \times \cdots \times \mathcal{P}(\mathbf{G}(A)_{\tau(u_m)})$, the least fixed-point of a certain mapping $E \to E$, also denoted by $\bar{\Gamma}$.

The following theorem is proved in [1, Theorem 4.14].

Theorem 2.7. *The n-tuple of sets of graphs* $(\mathbf{L}(\Gamma, u_1), \ldots, \mathbf{L}(\Gamma, u_n))$ *is the least solution in E of the system* $\bar{\Gamma}$.

We conclude this section with the definition of paths. This definition is not standard since we deal with hypergraphs.

Definition 2.8 (Paths). With G in $\mathbf{G}(A)$, we associate the set

$$\mathbf{K}(G) := \{(v, e, i, j, v') \mid v, v' \in \mathbf{V}_G, e \in \mathbf{E}_G, i, j \in [\tau(e)], i \neq j,$$
$$v = \mathbf{vert}_G(e, i), v' = \mathbf{vert}_G(e, j)\}.$$

A *path from v to v' in G* is a nonempty sequence w of elements of $\mathbf{K}(G)$ of the form

$$w = (v, e_1, i_1, j_1, v_1)(v_1, e_2, i_2, j_2, v_2) \ldots (v_{k-1}, e_k, i_k, j_k, v').$$

From this notion of path, we get an appropriate extension of the classical notion of connectivity.

3. Datalog programs and context-free graph-grammars

We show in this section that conjunctions of literals can be represented, in a semantically meaningful way, by graphs, and, accordingly, that Datalog programs can be represented by context-free graph grammars.

Our definitions will be given with respect to a finite ranked alphabet A, considered as a set of relation symbols.

Definition 3.1 (*Interpretations*). An *A-interpretation* is an object $I = \langle \mathbf{D}_I, (a_I)_{a \in A} \rangle$, where for each a in A, a_I is $\tau(a)$-ary relation on $\mathbf{D}_I$, i.e., a subset of $\mathbf{D}_I^{\tau(a)}$. From this identification, it follows that the notations

$$(d_1, \ldots, d_k) \in a_I \quad \text{and} \quad a_I(d_1, \ldots, d_k)$$

are equivalent (where $(d_1, \ldots, d_k) \in \mathbf{D}_I^k$ and $k = \tau(a)$). The set $\mathbf{D}_I$ is called the *domain* of I.

In the special case where $k = \tau(a) = 0$, the symbol a can be considered as a Boolean constant denoting either **true** or **false**. Note that in this case, $\mathbf{D}_I^k$ is reduced to the empty sequence ( ) and that a_I is a subset of $\{( )\}$. If $a_I = \{( )\}$, then a_I denotes **true**. If $a_I = \emptyset$, then a_I denotes **false**.

Interpretations are logical structures. Hence, any closed logical formula written with the relational symbols of A is true or false in an A-interpretation.

Definition 3.2 (*The semantics of a graph*). Let I be an A-interpretation. We shall represent it by a 0-graph $\mathbf{G}(I)$ defined as follows:

$$\mathbf{V}_{\mathbf{G}(I)} := \mathbf{D}_I,$$

$$\mathbf{E}_{\mathbf{G}(I)} := \{(a, d_1, \ldots, d_k) \mid a \in A, (d_1, \ldots, d_k) \in a_I\},$$

$$\mathbf{lab}_{\mathbf{G}(I)}((a, d_1, \ldots, d_k)) := a,$$

$$\mathbf{vert}_{\mathbf{G}(I)}((a, d_1, \ldots, d_k)) := (d_1, \ldots, d_k).$$

Now let H be a graph in $\mathbf{G}(A)_n$. A *realization* of H in I is a homomorphism $h : H^0 \to \mathbf{G}(I)$. The n-tuple $(d_1, \ldots, d_n)$ in $\mathbf{D}_I^n$, where $d_i := h_{\mathbf{V}}(\mathbf{src}_{\mathbf{G}(I)}(i))$, is called the *parameter list* of h, and is denoted by $\mathbf{param}(h)$.

The *relation defined by H in I* is defined as the n-ary relation:

$$H_I := \{\mathbf{param}(h) \mid h \text{ is a realization of } H \text{ in } I\}.$$

If $L \subseteq \mathbf{G}(A)_n$, then we let

$$L_I := \bigcup \{H_I \mid H \in L\}.$$

We say that H is *realizable* in I if there exists at least one realization of H in I. There may exist several or none. We say that L is *realizable* in I if some H in L is realizable in I.

Example 3.3. Let $A=\{p, q\}$ with $\tau(p)=3$, $\tau(q)=1$. Let I be the A-interpretation with domain $\mathbb{N}$ such that:

$$(x, y, z) \in p_I \text{ iff } z=x+y,$$

$$x \in q_I \text{ iff } x=1.$$

The corresponding graph $\mathbf{G}(I)$ is (partially) shown in Fig. 3.

Let us now consider the graphs H and H' of Fig. 4.

The first one has many realizations in I with parameter lists of the form (n, n) for all n in $\mathbb{N}$. These realizations h correspond to the solutions of the system of equations $\{x_2=y+x_1; x_1=z+x_2\}$ with $\mathbf{param}(h)=(x_1, x_2)$. The graph H' has no realization in I. A realization h' of H' in I would correspond to a solution of the equation $x_1=1+x_1$, with $\mathbf{param}(h')=(x_1)$.

Definition 3.4 (*Literals and basic formulas*). Let $X_n=\{x_1, \ldots, x_n\}$ and $Y=\{y_1, \ldots, y_m, \ldots\}$ be disjoint sets of variables (intended to range over domains of interpretations). (Note that Y is countably infinite). Let $\mathbf{R}(A, X_n)$ be the set of first-order formulas of the form

$$\exists y_1, \ldots, y_m[\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_p]$$

where each formula φ_i is a *literal*, i.e., a formula of the form $a(z_1, \ldots, z_k)$, or $z_1=z_2$, for some a in A and some $z_1, \ldots, z_k$ in $X_n \cup Y$ with $k=\tau(a)$. We also assume that m is such that each variable $y_1, \ldots, y_m$ occurs in some $\varphi_1, \ldots, \varphi_p$. The formulas of $\mathbf{R}(A, X_n)$ are called *basic formulas.*

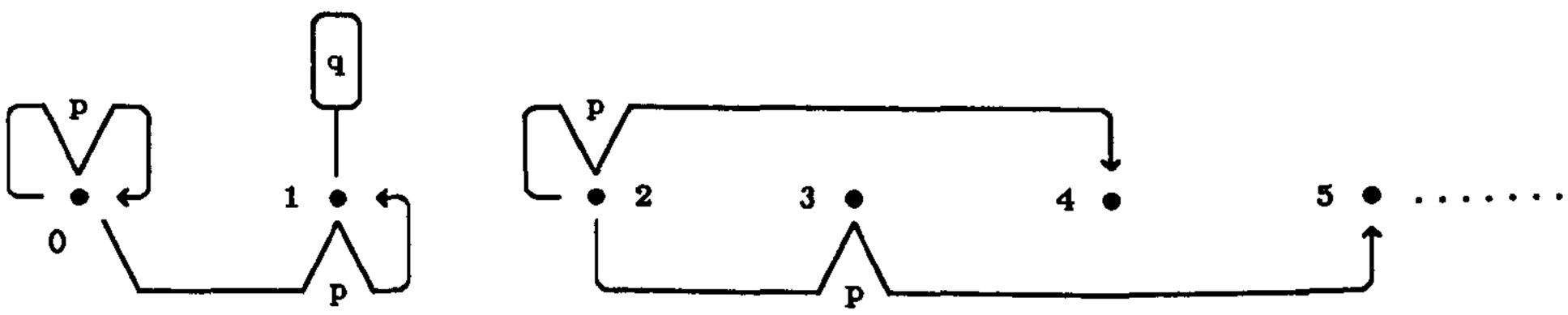


Fig. 3.

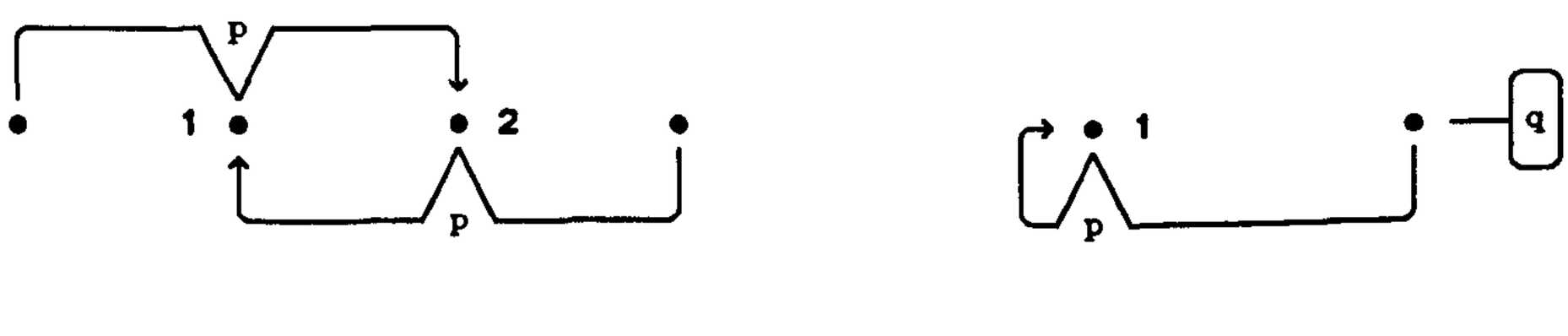


Fig. 4.

In an A-interpretation I, a basic formula ψ in $\mathbf{R}(A, X_n)$ defines a relation $\psi_{I,n} \subseteq \mathbf{D}_I^n$ such that

$$(d_1, \ldots, d_n) \in \psi_{I,n} :\Leftrightarrow (I, d_1, \ldots, d_n) \models \psi.$$

Note that $\psi \in \mathbf{R}(A, X_q)$ for each $q \geqslant n$, and that $\psi_{I,q}$ is not equal to $\psi_{I,n}$ if $q \neq n$. However, if ψ is defined as an element of $\mathbf{R}(A, X_n)$, then we shall call n the *type* of ψ (even if the variables $x_{n-q'}, \ldots, x_n$ do not occur in ψ) and we shall use the simplified notation ψ_I for $\psi_{I,n}$.

The language Datalog and its variants are presented in [10]. We first consider its core, that consists of systems of recursively defined relations. Datalog will be considered at the end of the section.

Definition 3.5 (*Systems of recursively defined relations*). Let U be a finite ranked alphabet, disjoint from A, enumerated as $\{u_1, \ldots, u_n\}$. The type of u_i is also denoted by m_i.

A *system of recursively defined relations over A with set of unknowns U* is a triple $S = \langle A, U, C \rangle$ where A, U are as above and C is a set of *clauses*, i.e., of formulas of the form

$$u_i(x_1, \ldots, x_{m_i}) \Leftarrow \varphi \tag{1}$$

where $\varphi \in \mathbf{R}(A \cup U, X_{m_i})$. One may have several clauses with the same lefthand side. We shall say that the clauses of the form (1) *define the unknown relation u_i*.

Definition 3.6 (*The least solution of a system*). Let S be a system as in Definition 3.5. (We shall say more simply a *system* in the sequel.) Let I be an A-interpretation. An n-tuple of relations $(R_1, \ldots, R_n)$ such that $R_i \subseteq \mathbf{D}_I^{m_i}$ for all i is a *solution* of S iff, for every clause of the form (1), we have

$$R_i \supseteq \varphi_J \tag{2}$$

where J is the $(A \cup U)$-interpretation $\langle \mathbf{D}_I, (a_I)_{a \in A}, (R_i)_{i=1,\ldots,n} \rangle$ and R_i interprets u_i. Condition (2) can also be written

$$J \models \forall x_1, \ldots, x_{m_i} \quad [u_i(x_1, \ldots, x_{m_i}) \Leftarrow \varphi]. \tag{3}$$

The relation φ_J defined by φ on $\mathbf{D}_I$ can also be written as a function of I and the n-tuple $(R_1, \ldots, R_n)$, by

$$\varphi_I[R_1, \ldots, R_n] := \varphi_J,$$

where J is the $(A \cup U)$-interpretation defined above. We can thus define more synthetically $(R_1, \ldots, R_n)$ as a solution of S in I iff, for all $i = 1, \ldots, n$:

$$R_i \supseteq \bigcup \{\varphi_I[R_1, \ldots, R_n] \mid \varphi \in C_i\} \tag{4}$$

where C_i is the set of righthand sides of the clauses of S that define u_i.

Letting $\|S\|_i(R_1, \ldots, R_n)$ denote the righthand side of (4), and $\|S\|(R_1, \ldots, R_n)$ denote the n-tuple $(\|S\|_1(R_1, \ldots, R_n), \ldots, \|S\|_n(R_1, \ldots, R_n))$, we finally obtain that $(R_1, \ldots, R_n)$ is a solution of S iff

$$(R_1, \ldots, R_n) \supseteq \|S\|(R_1, \ldots, R_n) \tag{5}$$

where $\supseteq$ is the component-wise extension of $\supseteq$ to n-tuples of relations on $\mathbf{D}_I$. (Let us recall that $R_i \subseteq \mathbf{D}_I^{m_i}$; in addition, $R_i \subseteq \{(\,)\}$ if $m_i = 0$.)

Let $E := \mathscr{P}(\mathbf{D}_I^{m_1}) \times \cdots \times \mathscr{P}(\mathbf{D}_I^{m_n})$, ordered componentwise by $\subseteq$. This partially ordered set is complete and has a least element $\boldsymbol{\emptyset} := (\emptyset, \emptyset, \ldots, \emptyset)$. It is not hard to verify that the mapping $\|S\|: E \to E$ is monotone and continuous. It follows from Lemma 1.2 that equation (5) has a least solution in E. This solution is also the least solution of the corresponding equation, namely

$$(R_1, \ldots, R_n) = \|S\|(R_1, \ldots, R_n). \tag{6}$$

We call it the n-tuple of relations defined in I by S, and we shall denote it by S_I. It can be defined concretely as:

$$\bigcup\{\|S\|^i(\boldsymbol{\emptyset}) \mid i \geq 0\}.$$

From this definition, it follows immediately that S_I is computable if I has a finite (given) domain, and if the relations (a_I), $a \in A$, are also given. In order to compute S_I, it suffices to compute $\|S\|^i(\boldsymbol{\emptyset})$ for $i = 1, 2, 3, \ldots$, and to stop as soon as $\|S\|^{i+1}(\boldsymbol{\emptyset}) = \|S\|^i(\boldsymbol{\emptyset})$. Such an integer i exists since $\mathbf{D}_I$ is finite, and $\|S\|^i(\boldsymbol{\emptyset})$ is the desired tuple of relations S_I. This algorithm is clearly inefficient. Efficient algorithms are surveyed in Gardarin [10]. (See also the work of Vieille [16], subsequent to [10].)

Example 3.7. Let $A = \{p\}$ with p of type 2. Let I be an interpretation. Its domain is a set of persons and $p_I(x, y)$ is intended to mean: "x is the father or the mother of y". Consider the following system over $U = \{\mathrm{ANC}, \mathrm{SG}\}$:

$$\begin{aligned}
\mathrm{ANC}(x_1, x_2) &\Leftarrow p(x_1, x_2)\\
\mathrm{ANC}(x_1, x_2) &\Leftarrow \exists y \quad [\mathrm{ANC}(x_1, y) \wedge p(y, x_2)]\\
\mathrm{SG}(x_1, x_2) &\Leftarrow x_1 = x_2\\
\mathrm{SG}(x_1, x_2) &\Leftarrow \exists y_1, y_2 \quad [\mathrm{SG}(y_1, y_2) \wedge p(y_1, x_1) \wedge p(y_2, x_2)].
\end{aligned}$$

The pair of relations $(\mathrm{ANC}_I, \mathrm{SG}_I)$ can be understood intuitively as

$$\begin{aligned}
\mathrm{ANC}_I(x, y) &\Leftrightarrow x \text{ is an ancestor of } y,\\
\mathrm{SG}_I(x, y) &\Leftrightarrow x \text{ and } y \text{ are of the same generation.}
\end{aligned}$$

Note also that ANC_I is nothing but the transitive closure of the relation p_I.

Definition 3.8 (*Connection graphs*). Let ψ be a basic formula of type n. It is of the form

$$\exists y_1, \ldots, y_m \quad [\varphi_1 \wedge \cdots \wedge \varphi_p \wedge \varphi_{p+1} \wedge \cdots \wedge \varphi_q] \tag{7}$$

where each literal φ_i, $i = 1, \ldots, p$, is of the form

$$a(z_1, \ldots, z_k) \tag{8}$$

for some $a \in A \cup U$, some $z_1, \ldots, z_k \in Y_m \cup X_n$, with $k = \tau(a)$, and each literal φ_i, $i = p+1, \ldots, q$ is of the form

$$z_1 = z_2,$$

for some z_1 and z_2 as above. (We denote by Y_m the set of variables $\{y_1, \ldots, y_m\}$.)

We now construct a graph H in $\mathbf{FG}(A)_n$ as follows:

$$\mathbf{V}_H := \{x_1, \ldots, x_n, y_1, \ldots, y_m\}/\approx$$

where $\approx$ is the equivalence relation on $X_n \cup Y_m$ generated by the pairs (z_1, z_2) such that $z_1 = z_2$ is a literal in ψ. The equivalence class of a variable z is denoted by $[z]$. We also let

$$\mathbf{E}_H := \{\varphi_1, \ldots, \varphi_p\}$$

$$\left.\begin{aligned} &\mathbf{lab}_H(\varphi_i) := a \\ &\mathbf{vert}_H(\varphi_i) := ([z_1], \ldots, [z_k]) \end{aligned}\right\} \text{where } \varphi_i \text{ is of the form (8)}$$

$$\mathbf{src}_H := ([x_1], \ldots, [x_n]).$$

This graph H is denoted by $\mathbf{H}(\psi)$. As in [9], we call it the *connection graph* of ψ. (Note that two literals φ_i and φ_j may be identical. We obtain then two edges in H with the same label and the same sequence of vertices.)

Conversely, every graph in $\mathbf{FG}(A)_n$ is the connection graph of some basic formula of type n.

Lemma 3.9. *For every interpretation I, for every basic formula ψ we have $\mathbf{H}(\psi)_I = \psi_I$.*

Proof. Let $(d_1, \ldots, d_n) \in \psi_I$. There exists an m-tuple $(d'_1, \ldots, d'_m) \in \mathbf{D}_I^m$ such that $(I, d_1, \ldots, d_n, d'_1, \ldots, d'_m) \models \varphi_1 \wedge \cdots \wedge \varphi_q$ (where d_i is the value of x_i and d'_j is that of y_j). The mappings $h_{\mathbf{V}}$, $h_{\mathbf{E}}$ such that

$$h_{\mathbf{V}}(x_i) := d_i, \quad i = 1, \ldots, n$$

$$h_{\mathbf{V}}(y_i) := d'_i, \quad i = 1, \ldots, m,$$

$$h_{\mathbf{E}}(\varphi_i) := (a, h_{\mathbf{V}}(z_1), \ldots, h_{\mathbf{V}}(z_k)), \quad \text{where } \varphi_i = a(z_1, \ldots, z_k)$$

form a homomorphism $h : \mathbf{H}(\psi)^0 \to \mathbf{G}(I)$, such that $\mathbf{param}(h) = (d_1, \ldots, d_n)$. Hence $(d_1, \ldots, d_n) \in \mathbf{H}(\psi)_I$.

One proves similarly that $\mathbf{H}(\psi)_I \subseteq \psi_I$. □

Remark 3.10. It follows immediately from this lemma that if ψ and ψ' are two basic formulas such that $\mathbf{H}(\psi) = \mathbf{H}(\psi')$, then they are *equivalent*, i.e., they define the same relation in every interpretation. But the converse is not true. The two formulas

$$a(x_1, x_2) \quad \text{and} \quad a(x_1, x_2) \wedge a(x_1, x_2),$$

or the two formulas

$$\exists y_1 \quad [a(x_1, y_1)] \qquad \text{and} \qquad \exists y_1, y_2 \quad [a(x_1, y_1) \wedge a(x_1, y_2)],$$

are equivalent, but their connection graphs are distinct.

Definition 3.11 (*The context-free graph grammar associated with a system of recursively defined relations*). Let S be a system (as in Definition 3.5), with set of unknowns U, and set of clauses C. Let Γ be the context-free graph grammar $\langle A, U, P\rangle$ where P is the set of production rules constructed from C as follows.

For every clause of the form $u_i(x_1, \ldots, x_{m_i}) \Leftarrow \psi$ belonging to C, we put in P, the production rule

$$u_i \to \mathbf{H}(\psi).$$

Note that if ψ is the basic formula **true** (corresponding to an empty conjunction of literals), then the corresponding production rule is $u_i \to \mathbf{k}$, where $k = m_i$. (See Example 2.2(1) for the definition of **k**.) This case should be distinguished from the one where no clause defines u_i.

We shall denote by $\mathbf{L}(S, u_i)$ the set of graphs $\mathbf{L}(\Gamma, u_i)$. The sets of graphs $\mathbf{L}(S, u_1), \ldots, \mathbf{L}(S, u_n)$ are called the sets of *computation graphs* of S. We shall then establish the following theorem.

Theorem 3.12. *For every system S of recursively defined relations over A with set of unknowns $U = \{u_1, \ldots, u_n\}$, for every A-interpretation I, the n-tuple of relations S_I is equal to $(\mathbf{L}(S, u_1)_I, \ldots, \mathbf{L}(S, u_n)_I)$, i.e., to the n-tuple of relations associated with the n-tuple of sets of computation graphs of the system S.*

This theorem can be illustrated by the following commutative diagram:

$$\begin{array}{ccc} S & \xrightarrow{(1)} & S_I = (\mathbf{L}(\Gamma, u_1)_I, \ldots, \mathbf{L}(\Gamma, u_n)_I) \\ \Big\downarrow{\scriptstyle (2)} & & \Big\uparrow{\scriptstyle (4)} \\ \Gamma & \xrightarrow[(3)]{} & (\mathbf{L}(\Gamma, u_1), \ldots, \mathbf{L}(\Gamma, u_n)) \end{array}$$

The arrows (1) to (4) of this diagram refer to the following definitions:

(1) the tuple of relations S_I defined by a system S (Definition 3.6) in an interpretation I;

(2) the graph grammar Γ associated with a system S (Definition 3.11);

(3) the sets of graphs generated by a grammar (Definition 2.4);

(4) the relation L_I associated with a set of graphs L (Definition 3.2).

We aim to establish the theorem by using Lemma 1.3, and Theorem 2.7. We shall need the following lemma.

Lemma 3.13. *Let $H \in \mathbf{FG}(A \cup U)_m$. Let $L_i \subseteq \mathbf{FG}(A)_{m_i}$ for each $i = 1, \ldots, n$. Let I be an A-interpretation. Then, we have*

$$H[L_1/u_1, \ldots, L_n/u_n]_I = H_J$$

where J is the interpretation $\langle \mathbf{D}_I, (a_I)_{a \in A}, (L_{iI})_{i \in [n]} \rangle$.

We omit the proof as it is a straightforward but lengthy verification.

Proof of Theorem 3.12. We know by Theorem 2.7 that the n-tuple $(\mathbf{L}(\Gamma, u_1), \ldots, \mathbf{L}(\Gamma, u_n))$ is the least solution in $(E, \subseteq)$, where $E := \mathscr{P}(\mathbf{FG}(A)_{m_1}) \times \cdots \times \mathscr{P}(\mathbf{FG}(A)_{m_n})$ of the equation $x = f(x)$ where

$$f(L_1, \ldots, L_n) := (f_1(L_1, \ldots, L_n), \ldots, f_n(L_1, \ldots, L_n)),$$

$$f_i(L_1, \ldots, L_n) := \bigcup \{D[L_1/u_1, \ldots, L_n/u_n] / D \in P_i\},$$

and P_i is the set of graphs D such that $u_i \to D$ is a production rule in P.

The tuple of relations defined by S is the least solution in $(E', \subseteq)$ of the equation $x = f'(x)$ where

$$E' := \mathscr{P}(\mathbf{D}_I^{m_1}) \times \cdots \times \mathscr{P}(\mathbf{D}_I^{m_n}),$$

$$f'(R_1, \ldots, R_n) := (f_1'(R_1, \ldots, R_n), \ldots, f_n'(R_1, \ldots, R_n))$$

$$f_i'(R_1, \ldots, R_n) := \bigcup \{\varphi_I[R_1, \ldots, R_n] \mid \varphi \text{ is the righthand side of a clause that defines } u_i\}.$$

Let $h : E \to E'$ be such that

$$h((L_1, \ldots, L_n)) = (L_{1I}, \ldots, L_{nI}).$$

It is monotone and continuous, and clearly $h((\emptyset, \ldots, \emptyset)) = \emptyset$.

In order to be able to apply Lemma 1.3 we need only verify that $h \circ f = f' \circ h$, i.e., that

$$f_i(L_1, \ldots, L_n)_I = f_i'(L_{1I}, \ldots, L_{nI}).$$

But this follows immediately from the construction of Γ and Lemma 3.13, since, for every φ in $\mathbf{R}(A \cup U, X_p)$,

$$\varphi_I[L_{1I}, \ldots, L_{nI}] = \varphi_J = \mathbf{H}(\varphi)_J$$

where $J = \langle \mathbf{D}_I, (a_I)_{a \in A}, (L_{iI})_{i \in [n]} \rangle$.

Hence, we obtain by Lemma 1.3 that $h(\mu x.f(x)) = \mu x.f'(x)$. In other words, and by Theorem 2.7, $(\mathbf{L}(\Gamma, u_1)_I, \ldots, \mathbf{L}(\Gamma, u_n)_I)$ is the tuple of relations defined by S in I. □

In the following sections, we shall see that interesting properties of S_I can be derived from the consideration of the context-free sets of graphs $\mathbf{L}(\Gamma, u_1), \ldots, \mathbf{L}(\Gamma, u_n)$.

We first extend to a larger class of systems the construction of Definition 3.11.

Definition 3.14 (*DATALOG^f, i.e., DATALOG with functions*). In addition to A, let us assume that a finite ranked set F of *function symbols* is given. Each symbol f in F has a rank $\rho(f)$ in $\mathbb{N}$. A symbol of rank 0 is called a *constant*. The set of well formed terms built with F and variables from a set X is denoted by $\mathbf{M}(F, X)$.

An (A, F)-interpretation is a tuple $I = \langle \mathbf{D}_I, (a_I)_{a\in A}, (f_I)_{f\in F}\rangle$ where the a_I are as above, and f_I is a total mapping: $\mathbf{D}_I^{\rho(f)} \to \mathbf{D}_I$ for each f in F.

A *DATALOG^f program* is a pair (S, φ) where S is *DATALOG^f-system* over (A, F) built with a finite ranked set $U = \{u_1, \ldots, u_n\}$ of unknowns (intended to denote relations), and consisting of a set of *clauses*. A clause here is a first-order formula of the form

$$u_i(t_1, \ldots, t_{m_i}) \Leftarrow \exists y_1, \ldots, y_m[\varphi_1 \wedge \cdots \wedge \varphi_p] \tag{1}$$

where $t_1, \ldots, t_{m_i} \in \mathbf{M}(F, Z_q)$, Z_q is an auxiliary set of variables ($Z_q = \{z_1, \ldots, z_q\}$), and $\varphi_1, \ldots, \varphi_p$ are literals of the form $t_1 = t_2$, or $a(t_1, \ldots, t_k)$, or $u_j(t_1, \ldots, t_k)$, with $k = \tau(a) = \tau(u_j)$, for some terms $t_1, \ldots, t_k$ in $\mathbf{M}(F, Y_m \cup Z_q)$.

The component φ of a program (S, φ) is a literal of the form $u_j(t_1, \ldots, t_k)$ as above, with a set of variables X_p for some p. It is called a *query*.

Given an (A, F)-interpretation I, a *solution* of S is an n-tuple of relations $(R_1, \ldots, R_n)$, such that $R_i \subseteq \mathbf{D}_I^{m_i}$ for all i, and every clause is *satisfied* in $J \coloneqq \langle \mathbf{D}_I, (a_I)_{a\in A}, (f_I)_{f\in F}, (R_i)_{i\in[n]}\rangle$. We mean by this that, for a clause of the form (1):

$$J \models \forall z_1, \ldots, z_q \quad [u_i(t_1, \ldots, t_{m_i}) \Leftarrow \exists y_1, \ldots, y_m \quad (\varphi_1 \wedge \cdots \wedge \varphi_p)]. \tag{2}$$

It follows from Lemma 3.16 below that S has a least solution, $(\bar{R}_1, \ldots, \bar{R}_n)$. The *output of the program* (S, φ) is then defined as the set of p-tuples ν in $\mathbf{D}_I^p$ such that $(t_1(\nu), \ldots, t_k(\nu)) \in \bar{R}_j$ (where φ is assumed to be of the form $u_j(t_1, \ldots, t_k)$ and Z_p is the set of variables occurring there).

In the following construction, we translate a DATALOG^f system into a system (in the restricted sense of Definition 3.5) having the same solutions.

Construction 3.15. Let S be a DATALOG^f-system over (A, F). We first create a new set of relation symbols, $\mathbf{A}_F \coloneqq \{\mathbf{r}_f \mid f \in F\}$, with $\tau(\mathbf{r}_f) = \rho(f) + 1$ for all f in F. For every (A, F)-interpretation I, we construct an $(A \cup \mathbf{A}_F)$-interpretation I' as follows:

$$I' \coloneqq \langle \mathbf{D}_I, (a_I)_{a\in A}, (r_{fI'})_{f\in F}\rangle,$$

where

$$\mathbf{r}_{fI'} \coloneqq \{(d_1, \ldots, d_{k+1}) \in \mathbf{D}_I^{k+1} \mid f_I(d_1, \ldots, d_k) = d_{k+1}\}.$$

Every atomic formula of the form $y_1 = t$ where $t \in \mathbf{M}(F, Y_m)$ can be translated into a basic formula that we shall denote by "$y_1 = t$" such that $(I, d_1, \ldots, d_m) \models y_1 = t$ iff $(I', d_1, \ldots, d_m) \models$ "$y_1 = t$". For example, the basic formula "$y_1 = f(g(x_1), x_2, g(x_2))$" is

$$\exists w_1, w_2 \quad [\mathbf{r}_f(w_1, x_2, w_2, y_1) \wedge \mathbf{r}_g(x_1, w_1) \wedge \mathbf{r}_g(x_2, w_2)].$$

Let $U = \{u_1, \ldots, u_n\}$ be a ranked alphabet, with u_i of rank m_i, let $R_1, \ldots, R_n$ be relations, $R_i \subseteq \mathbf{D}_I^{m_i}$ for $i = 1, \ldots, n$. Let J be the extension of I into an $(A \cup U, F)$-interpretation (where R_i interprets u_i), and J' be the similar extension of I' into an $(A \cup \mathbf{A}_F \cup U)$-interpretation.

Let C be a clause of the DATALOG^f system, of the form (1). It is clear that (2) holds iff

$$J' \models \forall x_1, \ldots, x_m \quad [u_i(x_1, \ldots, x_{m_i}) \Leftarrow \exists y_1, \ldots, y_m (\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_p \wedge \text{“}x_1 = t_1\text{”} \wedge \cdots \wedge \text{“}x_{m_i} = t_{m_i}\text{”})]. \qquad (3)$$

The formula inside $[\cdots]$ in the righthand side of (3) can be transformed into a clause over $A \cup \mathbf{A}_F$ (in the sense of Definition 3.5): it suffices to merge the existentially quantified variables of the "$x_i = t_i$" basic formula with the variables $y_1, \ldots, y_m$. This gives us the desired clause C' that translates the given clause C.

We let S' be the set of clauses C' associated with all clauses C of S. Note that S and S' have the same sets of unknowns.

We have already observed that (2)$\Leftrightarrow$(3). This means that S and S' have the same solutions, hence, the same least solution in every interpretation I. This remark ensures that the semantics of S is well-defined. In addition, one can add to S' a new unknown u_0 of type p with defining clause:

$$u_0(x_1, \ldots, x_p) \Leftarrow \exists y_1, \ldots, y_n \quad [u_j(y_1, \ldots, y_{n_j}) \wedge \text{“}y_1 = t_1\text{”} \wedge \cdots \wedge \text{“}y_{m_j} = t_{m_j}\text{”}]$$

where $\varphi = u_j(t_1, \ldots, t_{m_j})$ is a query. In this way, we make a given DATALOG^f program (S, φ) into a system of recursively defined relations of the form of Definition 3.5. It is clear that the output $(S, \varphi)_I$ of (S, φ) in I is the u_0-component of the least solution of S' in I'. Hence, we have proved the following result.

Proposition 3.16. *For every DATALOG^f-program (S, φ) over (A, F), one can construct a system S' over $(A \cup \mathbf{A}_F)$ such that for every (A, F)-interpretation I, the output of (S, φ) in I is the first component of the least solution of S' in the corresponding $(A \cup \mathbf{A}_F)$-interpretation I'.*

A context-free graph grammar can then be associated with S' by Construction 3.11.

Example 3.17. Let f, g, h be function symbols of respective arities 1, 1 and 2. Let a, b be relation symbols of type 2. Consider the DATALOG^f-system S over $(\{a, b\}, \{f, g, h\})$:

$$\begin{aligned}
u(f(z_1), z_2) &\Leftarrow a(z_1, f(z_2)) \wedge v(z_1) \\
u(z_1, z_1) &\Leftarrow \exists y_1 \quad [b(z_1, f(y_1)) \wedge f(z_1) = g(y_1)] \\
v(h(z_1, z_2)) &\Leftarrow \textbf{true} \\
v(z_1) &\Leftarrow \exists y_1 \quad [a(z_1, y_1) \wedge v(g(y_1))].
\end{aligned}$$

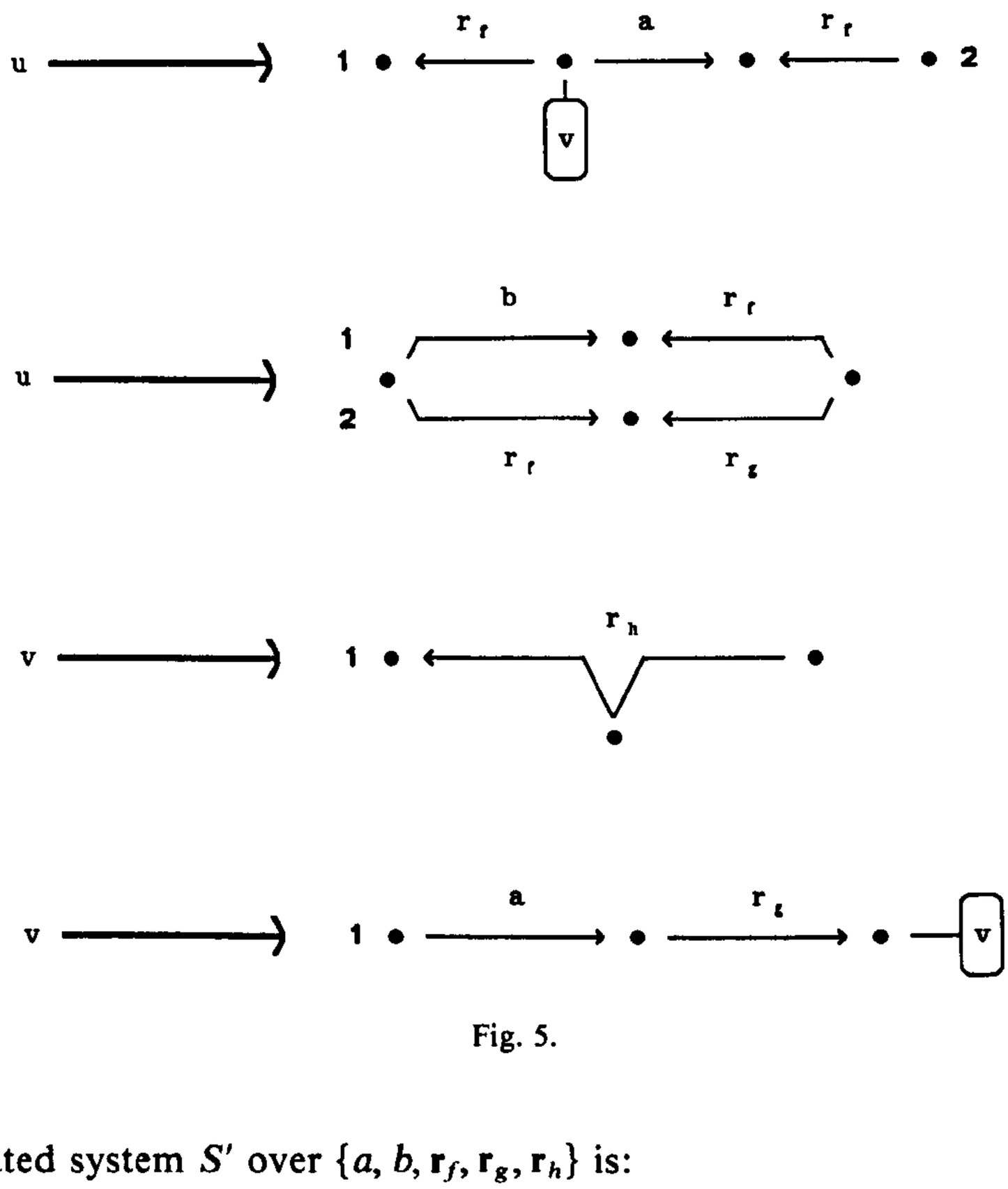


Fig. 5.

The associated system S' over $\{a, b, \mathbf{r}_f, \mathbf{r}_g, \mathbf{r}_h\}$ is:

$$u(x_1, x_2) \Leftarrow \exists z_1, w_1 \quad [\mathbf{r}_f(z_1, x_1) \wedge a(z_1, w_1) \wedge \mathbf{r}_f(x_2, w_1) \wedge v(z_1)]$$

$$u(x_1, x_2) \Leftarrow \exists y_1, w_1, w_2 \quad [b(x_1, w_1) \wedge \mathbf{r}_f(y_1, w_1) \wedge \mathbf{r}_f(x_1, w_2) \wedge \mathbf{r}_g(y_1, w_2) \wedge x_2 = x_1]$$

$$v(x_1) \Leftarrow \exists z_1, z_2 \quad [\mathbf{r}_h(z_1, z_2, x_1)]$$

$$v(x_1) \Leftarrow \exists y_1, w_1 \quad [a(x_1, y_1) \wedge \mathbf{r}_g(y_1, w_1) \wedge v(w_1)]$$

The graph grammar corresponding to S' by Construction 3.11 is shown in Fig. 5 (u_1 is of type 2 and u_2 is of type 1).

4. Monadic second-order properties of graphs

Many useful properties of graphs can be expressed in monadic second-order logic. We shall introduce a two-sorted calculus, and we shall consider a graph G as a two-sorted logical structure, the two domains of which are the set $\mathbf{V}_G$ of vertices, and the set $\mathbf{E}_G$ of edges.

Definition 4.1 (*Graphs as logical structures*). In order to express properties of graphs in $\mathbf{G}(A)_k$, we define the following symbols:

$\mathbf{v}$	the *vertex* sort,
$\mathbf{e}$	the *edge* sort,
$\mathbf{s}_i$	a constant of sort $\mathbf{v}$, for $1 \leqslant i \leqslant k$,
$\mathbf{edg}_a$	a predicate symbol of arity $(\mathbf{e}, \mathbf{v}, \mathbf{v}, \ldots, \mathbf{v})$ for $a \in A$, with $\tau(a)$ occurrences of $\mathbf{v}$.

With $G \in \mathbf{G}(A)_k$ we associate the logical structure

$$|G| := \langle \mathbf{V}_G, \mathbf{E}_G, (\mathbf{s}_{iG})_{i\in[k]}, (\mathbf{edg}_{aG})_{a\in A}\rangle$$

where $\mathbf{V}_G$ is the domain of sort $\mathbf{v}$, $\mathbf{E}_G$ is the domain of sort $\mathbf{e}$, $\mathbf{s}_{iG}$ is the ith source of G, and $\mathbf{edg}_{aG}(e, v_1, \ldots, v_n) = \mathbf{true}$ iff $\mathbf{lab}_G(e) = a$ and $\mathbf{vert}_G(e) = (v_1, \ldots, v_n)$.

We build formulas by using *object variables* $u, x, y, z, u' \ldots$ of sort $\mathbf{v}$ or $\mathbf{e}$, denoting, respectively, vertices or edges, and *set variables* U, X, Y, Z, U' of sort $\mathbf{v}$ or $\mathbf{e}$, denoting, respectively, sets of vertices or sets of edges.

Let $\mathcal{U}$ be a many-sorted set of variables $\{u, u', \ldots, U, U', \ldots\}$, where each variable has a sort in $\{\mathbf{v}, \mathbf{e}\}$ defined by the mapping σ. We denote by $\mathcal{U}_s$ the set $\mathcal{U} \cup \{\mathbf{s}_1, \ldots, \mathbf{s}_k\}$. By convention, we denote set variables by uppercase letters, and the remaining elements of $\mathcal{U}_s$, namely, object variables and constants, by lowercase letters.

The set $\mathcal{A}_{A,k}(\mathcal{U})$ of *atomic formulas* consists of:

$$u = u' \quad \text{for } u, u' \in \mathcal{U}_s, \sigma(u) = \sigma(u'),$$

$$u \in U \quad \text{for } u, U \in \mathcal{U}_s, \sigma(u) = \sigma(U),$$

$$\mathbf{edg}_a(u, w_1, \ldots, w_n) \quad \text{for } u, w_1, \ldots, w_n \in \mathcal{U}_s,$$

$$\sigma(w_1) = \cdots = \sigma(w_n) = \mathbf{v},\ \sigma(u) = \mathbf{e}.$$

The language of *monadic second-order logic* is the set of logical formulas formed with the above atomic formulas together with the Boolean connectives $\wedge$, $\vee$, $\neg$, the object quantifications $\forall u$, $\exists u$ (over vertices and edges), and the set quantifications $\forall U, \exists U$ (over sets of vertices and sets of edges). We denote by $\mathcal{L}(\mathcal{U})$ (or by $\mathcal{L}_{A,k}(\mathcal{U})$ if we must specify A and k), the set of these formulas having their free variables in $\mathcal{U}$.

A *$\mathcal{U}$-assignment* in G is a mapping ν with domain $\mathcal{U}$ such that $\nu(u) \in \mathbf{E}_G$ if $\sigma(u) = \mathbf{e}$, $\nu(u) \in \mathbf{V}_G$ if $\sigma(u) = \mathbf{v}$, $\nu(U) \subseteq \mathbf{E}_G$ if $\sigma(U) = \mathbf{e}$, and $\nu(U) \subseteq \mathbf{V}_G$ if $\sigma(U) = \mathbf{v}$. If $\varphi \in \mathcal{L}(\mathcal{U})$ then either $(|G|, \nu) \models \varphi$ or $(|G|, \nu) \not\models \varphi$. We say that the graph property *φ holds for ν in G* in the former case and we write this: "$\varphi_G(\nu)$ holds". We also write more simply $(G, \nu) \models \varphi$ or $(G, \nu) \not\models G$, by identifying G and $|G|$. If $\varphi \in \mathcal{L}(\emptyset)$, i.e., if φ is *closed*, then it is either true or false in G. This is written $G \models \varphi$ or $G \not\models \varphi$, and this defines a property that G enjoys or does not enjoy. Such a property is said to be *definable*. A set of graphs L is definable if the membership in L is a definable property.

Example 4.2 (*Colorability*). The existence of a vertex coloring of graph G that uses at most m colors, can be expressed as follows:

> There exist sets of vertices $X_1, \ldots, X_m$ such that $X_1 \cup \cdots \cup X_m = \mathbf{V}_G$, $X_i \cap X_j = \emptyset$ for $i \neq j$, and any two vertices of an edge do not both belong to X_i for any i.

It is clear from this expression that a formula φ in $\mathcal{L}$ can be constructed so that φ holds in G iff G is m-colorable. Hence, the m-colorability of a graph is a definable property.

Example 4.3 (*Paths*). We first assume that all symbols of A are of type 2.

It can be proved that, if a binary relation is definable in monadic second-order logic, then so is its transitive closure. (See Courcelle [3, Lemma 3.7] or [7, Lemma 1.2]). It follows that there exists a formula $\varphi \in \mathcal{L}_A(\{x, y, U\})$ such that, for every graph G in $\mathbf{G}(A)$, for every $\{x, y, U\}$-assignment ν in G, $\varphi_G(\nu)$ holds iff there exists a path from $\nu(x)$ to $\nu(y)$, the edges of which are in $\nu(U)$. By identifying x with $\nu(x)$, y with $\nu(y)$, and U with $\nu(U)$, we can say more briefly that $\varphi_G(x, y, U)$ holds iff there exists a path from x to y, all edges of which are in U.

It follows that many graph properties, like the existence of circuits or connectivity, are definable.

If A contains symbols of various types, then the notion of a path has been defined in Definition 2.8. Such a path can be represented as a finite indexed family of sets of edges $(W_\alpha)_{\alpha \in \mathbf{B}(A)}$, where W_α is the set of edges $e \in \mathbf{E}_G$ such that (v, e, i, j, v') belongs to the path, and $\alpha = (\mathbf{lab}_G(e), i, j)$. We denote by $\mathbf{B}(A)$ the set of triples (a, i, j) where $a \in A$, $i, j \in [\tau(a)]$ and $i \neq j$). The formulas expressing the existence of paths can be modified accordingly.

The following theorem has been established in [3]. See [7] for an informal account, and [4] for a similar result applied to a different notion of context-free graph grammar.

Theorem 4.4. *Let $L \subseteq \mathbf{FG}(A)_m$ be a context-free set of graphs defined by a given grammar. Let φ be a formula in $\mathcal{L}_{A,m}$. One can construct a grammar generating $\{G \in L \mid G \models \varphi\}$. One can decide whether $G \models \varphi$ for some G in L, or whether $G \models \varphi$ for all G in L.*

Remark 4.5. If $L = \mathbf{L}(\Gamma, u_1)$ where $U = \{u_1, \ldots, u_n\}$ is the nonterminal alphabet, then, the grammar Γ' such that $\mathbf{L}(\Gamma', u_1') = \{G \in L \mid G \models \varphi\}$ that one can construct from Γ and φ has the following properties. Its nonterminal alphabet $U' = \{u_1', \ldots, u_m'\}$ is, in general, larger than U. The righthand sides of its production rules are obtained from those of Γ by modifications of nonterminal edge labels only.

5. Applications

We give examples of properties of a system, or of the relations defined by a system, that can be formulated in monadic second-order logic on the associated sets of computation graphs, and that can be decided by Theorem 4.4.

5.1. Intra-element redundancies

Certain redundancies investigated in [14, 15] can be detected by the decidability result of Theorem 4.4 applied to the context-free graph grammar associated with a system.

Definition 5.1. A graph has *multiple edges* if it has at least two edges having the same label and the same sequence of vertices. That a graph has multiple edges is expressible in monadic second-order logic (and even in first-order logic); hence, Theorem 4.4 yields:

Proposition 5.2. *Let S be a system. One can decide whether the set $\mathbf{L}(S, u_1)$ contains graphs having multiple edges.*

The existence of a graph G in $\mathbf{L}(S, u_1)$ having multiple edges is called an *intra-element redundancy* in [15]. The presence of multiple edges indicates a redundancy in the computation, as shown by the following example.

Example 5.3. Let S consist of

$$u(x_1, x_2) \Leftarrow a(x_1, x_2) \wedge b(x_1) \wedge b(x_2),$$

$$u(x_1, x_2) \Leftarrow a(x_1, x_3) \wedge u(x_3, x_4) \wedge u(x_4, x_2).$$

Then $\mathbf{L}(S, u)$ contains the graph G with multiple edges that is shown in Fig. 6.

The clauses that define the relation u contain the following redundancy. Let us assume that we have guessed values x_3 and x_4 intended to show, by the second clause, that some pair (x_1, x_2) satisfies u_I. If we know that a pair (x_3, x_4) satisfies u_I by the first clause, then this means that we have verified that x_4 satisfies b_I. If in addition we verify that (x_4, x_2) satisfies u_I by checking that the first clause holds for this pair, then we must verify once again that $b_I(x_4)$ holds. This redundancy is visible on the computation graph G, where a vertex (corresponding to x_4 as above) belongs to two unary edges with label b.

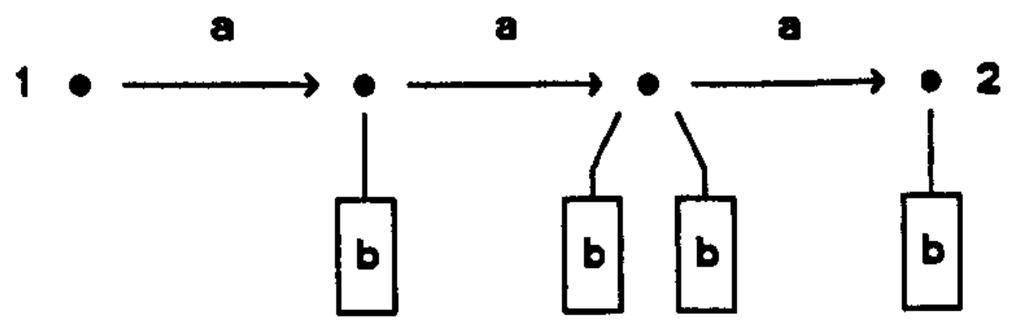


Fig. 6.

5.2. *Properties of interpretations*

When one writes a DATALOG program, one usually intends to execute it in interpretations satisfying certain properties, and not in completely arbitrary interpretations.

Definition 5.4. Let $\mathscr{C}$ be a class of interpretations, let S be a system, and u be an unknown of S. We say that u is *unsatisfiable* in $\mathscr{C}$ if $u_I = \emptyset$ for all I in $\mathscr{C}$. Hence u is unsatisfiable in $\mathscr{C}$ iff no graph G in $\mathbf{L}(S, u)$ is realizable in any interpretation I of $\mathscr{C}$.

A property P of graphs is *stable under homomorphisms* if for all graphs G, G', if h is a homomorphism $G \to G'$, then one has $P(G) \Rightarrow P(G')$. It is *insensitive to multiple edges* if for every graph G, if h is the surjective homomorphism $G \to \tilde{G}$, making identical any two edges having the same label and the same sequence of vertices, then one has $P(\tilde{G}) \Leftrightarrow P(G)$.

For every property P of graphs in $\mathbf{G}(A)_0$, we let $\mathscr{C}(P)$ be the class of all interpretations such that the associated graph (by Definition 3.2) satisfies P.

Theorem 5.5. *Let P be a property of graphs in $\mathbf{G}(A)_0$ that is stable under homomorphisms, and is insensitive to multiple edges. Let $\mathscr{C}$ be the class $\mathscr{C}(\neg P)$,* i.e., *the class of all interpretations that do not satisfy P.*

(1) *If $G \in \mathbf{FG}(A)_0$, then G is realizable in $\mathscr{C}$ iff $P(G)$ does not hold.*

(2) *If S is a system and u is one of its unknowns, then u is unsatisfiable in $\mathscr{C}$ iff $P(G)$ holds for all G in $\mathbf{L}(S, u)$.*

(3) *If, furthermore, P is expressible in monadic second-order logic, then the condition of* (2) *is decidable, and one can construct from S a system S' with an unknown u' such that $\mathbf{L}(S', u')$ is the set of graphs of $\mathbf{L}(S, u)$ that are realizable in $\mathscr{C}$. Hence, $u'_I = u_I$ for all interpretations I in $\mathscr{C}$.*

Proof. (1) Let $G \in \mathbf{FG}(A)_0$ satisfy $\neg P$. Then $\tilde{G}$ also satisfies $\neg P$ since P is insensitive to multiple edges. But $\tilde{G}$ is an interpretation in $\mathscr{C}$ (a finite one), and G is realizable in the interpretation $\tilde{G}$ that belongs to $\mathscr{C}$. (The surjective homomorphism of G onto $\tilde{G}$ is a realization). Hence G is realizable in $\mathscr{C}$.

Conversely, let G be realizable in $\mathscr{C}$. It has a realization in some $I \in \mathscr{C}$. Hence, G does not satisfy P. (Otherwise, the interpretation I would also satisfy P, since P is stable under homomorphisms, and this would contradict the definition of $\mathscr{C}$.)

(2) Immediate consequence of (1), and the definitions.

(3) This follows from Theorem 4.4. One can construct from S and u a context-free graph grammar that generates the set of graphs in $\mathbf{L}(S, u)$ that are realizable in $\mathscr{C}$. From this grammar, one can obtain S' and u' such that $\mathbf{L}(S', u')$ is this set, since every graph is the connection graph of some basic formula. □

Example 5.6 (Continuation of Example 3.7). Let P_0 be the property expressing that a graph in $\mathbf{G}(A)$ has oriented cycles, the edges of which are labeled by p. (A loop is considered as a cycle). This property satisfies all conditions of Theorem 5.5.

The intended interpretations of the system defining ANC and SG will belong to $\mathscr{C}(\neg P_0)$, i.e., to the class of graphs having no cycle. Let us now consider the system consisting of

$$\mathrm{ANC}(x_1, x_2) \Leftarrow p(x_1, x_2)$$

$$\mathrm{ANC}(x_1, x_2) \Leftarrow \exists y \quad [\mathrm{ANC}(x_1, y) \wedge p(y, x_2)]$$

$$\theta(x_1, x_2) \Leftarrow \mathrm{ANC}(x_1, x_2) \wedge p(x_2, x_1).$$

It is clear that the unknown θ is unsatisfiable in $\mathscr{C}(\neg P_0)$ since the relation it defines expresses precisely the existence of a cycle with edges labeled by p.

Definition 5.7 (A class of "good" formulas). Let $\mathscr{S}$ be the subset of $\mathscr{L}$ consisting of all formulas built with set and object variables of sort **v**, with the atomic formulas $x = y$, $y \in X$, $\mathbf{edg}'_a(x_1, \ldots, x_n)$ (for a in A of type n), $\mathbf{edg}''_\alpha(x, y)$ (for α in $\mathbf{B}(A)$; see Example 4.3), with the Boolean connectives $\wedge$ and $\vee$, the existential quantifications $\exists x$ and $\exists X$, and the *bounded* universal quantifications $\forall x \in X$, and $\forall Y \subseteq X$. (These formulas use neither variables of sort **e** nor negation, and the use of universal quantifications is limited.)

The meaning of $\mathbf{edg}'_a(x_1, \ldots, x_n)$ is: there exists an edge having the label a and the sequence of vertices $(x_1, \ldots, x_n)$. The meaning of $\mathbf{edg}''_\alpha(x, y)$ is: there exists an edge with label a, the ith vertex of which is x, and the jth vertex is y, where $\alpha = (a, i, j)$.

It is clear that every formula of $\mathscr{S}$ is not strictly speaking in $\mathscr{L}$ but can be translated into a formula of $\mathscr{L}$.

Proposition 5.8. *The properties defined by formulas in $\mathscr{S}$ satisfy the conditions of Theorem* (5.4), *i.e., are stable under homomorphisms, are insensitive to multiple edges, and are expressible in monadic second-order logic.*

Proof. Since the formulas of $\mathscr{S}$ are written without quantifications over edges and sets of edges, since the basic predicates $\mathbf{edg}'_a$ and $\mathbf{edg}''_\alpha$ do not depend on the multiplicity of edges, the properties defined by these formulas are necessarily insensitive to multiple edges.

Let us now consider a homomorphism $h : G \to G'$ and φ a closed formula in $\mathscr{S}$ such that $G \models \varphi$. We must establish that $G' \models \varphi$.

We shall prove by induction on the structure of φ, either closed or not, the validity of the following assertion:

For every $\mathscr{U}$-assignment γ in G,
where $\mathscr{U}$ is the set of free variables of φ,
if $(G, \gamma) \models \varphi$, then $(G', h \circ \gamma) \models \varphi$.

For the base cases, we have to consider φ of the following forms:

$x = y$

$x \in Y$

$\mathbf{edg}'_a(x_1, \ldots, x_n)$

$\mathbf{edg}''_\alpha(x_1, x_2)$.

The validity of (1) is clear in each of them.

For the inductive cases, we have to consider φ of the following forms:

$\varphi_1 \wedge \varphi_2$

$\varphi_1 \vee \varphi_2$

$\exists x . [\varphi_1]$

$\exists X . [\varphi_1]$

$\forall x \in X . [\varphi_1]$

$\forall y \subseteq X . [\varphi_1]$

We demonstrate only the last one. Let us assume that

$$(G, \gamma) \models \forall Y \subseteq X . [\varphi_1] \tag{1}$$

and verify that

$$(G', h \circ \gamma) \models \forall Y \subseteq X . [\varphi_1]. \tag{2}$$

Let Z be any subset of $h(\gamma(X))$. It is of the form $h(Z')$ for some subset Z' of $\gamma(X)$. Let γ' be the $\mathcal{U} \cup \{Y\}$-assignment extending γ, and such that $\gamma'(Y) = Z'$. By (1), we have $(G, \gamma') \models \varphi_1$, hence $(G', h \circ \gamma') \models \varphi_1$ by the induction hypothesis. Since Z was chosen arbitrarily in $\mathcal{P}(h(\gamma(X)))$, we have proved (2). The other cases are similar or easier to establish. □

The property P_0 considered in Example 5.5 is expressible in finite graphs by the following formula of $\mathcal{S}$:

$$\exists X \quad [\exists x . (x \in X) \wedge \forall x \in X . (\exists y . (y \in X \wedge \mathbf{edg}'_p(x, y)))].$$

Our next example is another application.

Example 5.9 (*Negated basic relations*). Assume that $A = A_0 \cup \{\neg a \mid a \in A_0\}$, where A_0 is a finite ranked alphabet, and $\neg a$ is a new symbol associated with a.

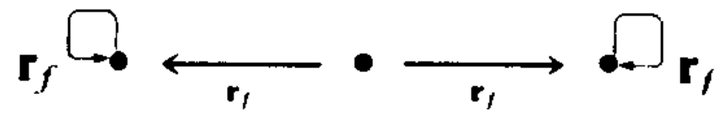
Let $\mathscr{C}_1$ be the class of interpretations I such that for all $d_1, \ldots, d_n$ in $\mathbf{D}_I$, $a_I(d_1, \ldots, d_n)$ and $(\neg a)_I(d_1, \ldots, d_n)$ do not hold simultaneously. It is clear that $\mathscr{C}_1 = \mathscr{C}(\neg P_1)$ where P_1 is the disjunction of the following formulas of $\mathcal{S}$, for all $a \in A_0$, where $n = \tau(a)$:

$$\exists x_1, \ldots, x_n \quad [\mathbf{edg}'_a(x_1, \ldots, x_n) \wedge \mathbf{edg}'_{\neg a}(x_1, \ldots, x_n)].$$

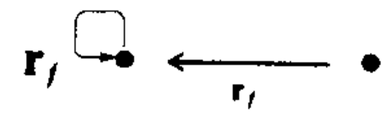
Now let S be a system and u be an unknown relation symbol. One can decide whether u is satisfiable in $\mathscr{C}_1$, whether the set $\mathbf{L}(S, u)$ contains graphs that are unrealizable in $\mathscr{C}_1$. In the latter case, one can construct (S', u') such that $\mathbf{L}(S', u')$ is the set of graphs in $\mathbf{L}(S, u)$ that are realizable in $\mathscr{C}_1$. It follows then that $u_I = u'_I$ for all $I \in \mathscr{C}_1$.

Our next two examples show situations where Theorem 5.5 is not applicable.

Example 5.10 (*Functions without negated basic relations*). Let F be a set of function symbols, let $\mathbf{A}_F$ be the associated set of relation symbols by the construction of Definition 3.15, let A be a set of relation symbols. Let $\mathscr{C}_2$ be the class of $(A \cup \mathbf{A}_F)$-interpretations I' associated with (A, F)-interpretations where the relations $\mathbf{r}_f$, $f \in F$ define total functions. This class is not of the form considered in Theorem 5.5. To see this, consider $f \in F$ of rank 1. The graph



is not the graph of any interpretation in $\mathscr{C}_2$, but its homomorphic image

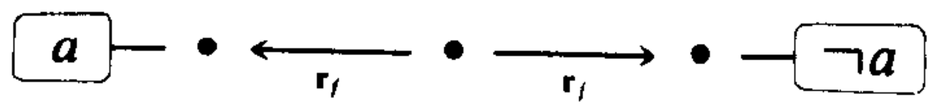


is. This proves that $\mathscr{C}_2$ is not of the form $\mathscr{C}(\neg P)$ for any graph property P that is stable by homomorphisms. Theorem 5.5 is actually irrelevant to the class $\mathscr{C}_2$ because every graph in $\mathbf{FG}(A \cup \mathbf{A}_F)$ is realizable in an interpretation of $\mathscr{C}_2$, in particular in the trivial one with a singleton domain.

Example 5.11 (*Functions and negated basic relations*). Let us now assume that A is as in Example 5.9 and let $\mathscr{C}_3 := \mathscr{C}_2 \cap \mathscr{C}_1$ be the class of $(A \cup \mathbf{A}_F)$-interpretations I where

- the relations $\mathbf{r}_{f1}$ represent total functions,
- $(\neg a)_I$ and a_I are disjoint, for every a in A_0.

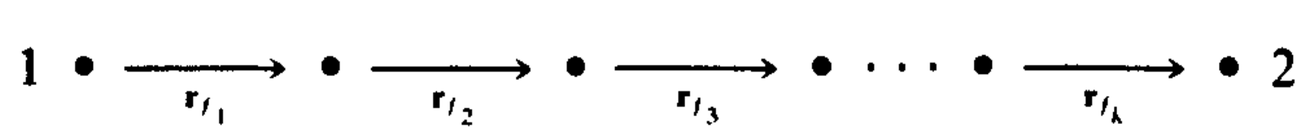
As in Example 5.10, $\mathscr{C}_3$ is not of the form considered in Theorem 5.5. But not every graph in $\mathbf{FG}(A \cup \mathbf{A}_F)$ is realizable in some interpretation of $\mathscr{C}_3$. Consider, for example, the graph:



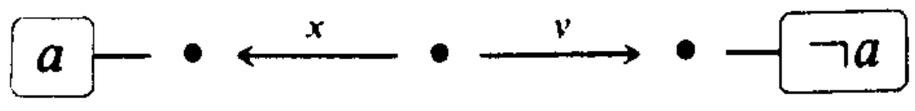
where a is a unary relation symbol in A. (A realization of this graph in an interpretation I of $\mathscr{C}_3$ would make identical the two vertices incident with the edges labeled by a and $\neg a$. But this would contradict the definition of $\mathscr{C}_2$.)

We shall prove that one cannot decide whether every graph in a context-free set is realizable in $\mathscr{C}_3$.

Let $P = (v_i, w_i)_{i=1,\ldots,n}$ be a Post Correspondence Problem with $v_i, w_i \in \{f, g\}^+$. For every word w in $\{f, g\}^*$, let $\mathbf{G}(w)$ be the graph in $\mathbf{FG}(\mathbf{A}_F)_2$ of the form



where $w = f_1 f_2 \ldots f_k$. Let H be the graph



It is easy to construct a context-free graph-grammar generating

$$L = \{H[\mathbf{G}(v_{i_1} v_{i_2} \cdots v_{i_k})/x, \mathbf{G}(w_{i_1} w_{i_2} \cdots w_{i_k})/y] \mid k \geqslant 1, i_1, \ldots, i_k \in [n]\}.$$

It is clear that a graph G in L, corresponding to a sequence $(i_1, i_2, \ldots, i_k)$ is realizable in $\mathscr{C}_3$ iff $v_{i_1} v_{i_2} \cdots v_{i_k} \neq w_{i_1} \cdots w_{i_k}$, i.e., iff $(i_1, \ldots, i_k)$ is not a solution of P.

Hence, every graph in L is realizable in $\mathscr{C}_3$ iff P has no solution, and the problem under consideration is undecidable. This proves also that one cannot express in monadic second-order logic that a graph is realizable in $\mathscr{C}_3$.

5.3. Nonconnected computation graphs

The basic idea of this paper is that a graph expresses a conjunction of literals, and that a vertex common to two edges of the graph corresponds to a variable common to two literals. If a graph has several connected components, this means it corresponds to the conjunction of several "independent" basic formulas. (Independent means without common variables.) In this subsection, we shall develop some consequences of this idea.

Definition 5.12. For every $G \in \mathbf{FG}(A)_m$, we let $\mathbf{C}(G)$ be the subgraph of G consisting of its sources, of the vertices of G connected by some path (see Definition 2.8) to some source, and of all edges incident to these vertices. Let $\mathbf{D}(G)$ be the subgraph of G (necessarily of type 0) consisting of all vertices and edges of G not in $\mathbf{C}(G)$.

The following lemma is easy to check from the definitions.

Lemma 5.13. *For every G in $\mathbf{FG}(A)_m$, for every A-interpretation I, we have:*

$$G_I = \mathbf{C}(G)_I \text{ if } \mathbf{D}(G)_I \neq \emptyset \quad (\text{i.e., if } \mathbf{D}(G)_I = \{(\,)\}, \text{ or equivalently, if } \mathbf{D}(G) \text{ is realizable in } I),$$

$$G_I = \emptyset \text{ if } \mathbf{D}(G)_I = \emptyset.$$

For every context-free subset L of $\mathbf{FG}(A)_m$, one can construct a context-free graph grammar generating $\mathbf{C}(L) := \{\mathbf{C}(G) \mid G \in L\}$. (This proof is not very difficult but quite technical. We omit it since the purpose of this paper is not to give technical results on context-free graph grammars.) If G is a computation graph of some system S, such that $\mathbf{D}(G) \neq \emptyset$, then $\mathbf{D}(G)$ is (semantically) useless. As the multiple edges in Proposition 5.2, it does not contribute to the result, i.e., to the tuples in $G_I = \mathbf{C}(G)_I$. Its only possible effect is to eliminate some tuples when $\mathbf{D}(G)_I = \emptyset$. Hence, we have:

Proposition 5.14. *Let S be a system, and u be an unknown.*

(1) *One can decide whether $\mathbf{D}(\mathbf{L}(S, u))$ (defined as the set of graphs $\mathbf{D}(G)$ for G in $\mathbf{L}(S, u)$) is empty.*

(2) *If $\mathbf{D}(\mathbf{L}(S, u)) \neq \emptyset$, one can construct a system S' with an unknown u' such that $\mathbf{L}(S', u') = \mathbf{C}(\mathbf{L}(S, u))$. This new program (S', u') is equivalent to (S, u) in every interpretation where all graphs of $\mathbf{L}(S, u)$ are realizable.*

6. Conclusion

This paper has introduced new tools for investigating and transforming systems of recursively defined relations and DATALOG^f programs. We do not claim to solve everything by means of these tools, but only to give criteria for establishing that certain properties are decidable.

The theory of transformations of context-free graph grammars is not yet very much developed. We have presently no general theorem stating that if a set of graphs L is context-free, then so is $\{\mathbf{T}(L) \mid G \in L\}$, where $\mathbf{T}$ is some "graph transduction". Such a theory would certainly help in the study of transformations of systems and of DATALOG^f (or even PROLOG) programs.

Let us finally mention that we do not think that these techniques can be extended to the extensions of DATALOG where the *unknown relations* are negated in the righthand sides of clauses. The reason is that, for these extended languages, the operational semantics depends too much on the interpretation. Hence, we do not see how to formulate an extension of Theorem 3.12 that could factor out semantics from computation.

Acknowledgement

I thank K. Callaway for helping me with the English style.

References

[1] M. Bauderon and B. Courcelle, Graph expressions and graph rewritings, *Math. Systems Theory* **20** (1987) 83–127.

[2] B. Courcelle, Equivalences and transformations of regular systems. Applications to recursive program schemes and grammars, *Theoret. Comput. Sci.* **42** (1986) 1–122.

[3] B. Courcelle, The monadic second-order logic of graphs I: Recognizable sets of finite graphs, *Inform. and Comput.* **85** (1990) 12–75.

[4] B. Courcelle, An axiomatic definition of context-free rewriting and its application to NLC graph grammars, *Theoret. Comput. Sci.* **55** (1987) 141–181.

[5] B. Courcelle, A representation of graphs by algebraic expressions and its use for graph rewriting systems, in: H. Ehrig and G. Rozenberg, eds., *Proc. 3rd Internat. Workshop on Graph Grammars*, Lecture Notes in Computer Science **291** (Springer, Berlin, 1987) 112–132.

[6] B. Courcelle, On context-free sets of graphs and their monadic second-order theory, in: H. Ehrig and G. Rozenberg, eds., *Proc. 3rd Internat. Workshop on Graph Grammars*, Lecture Notes in Computer Science **291** (Springer, Berlin, 1987) 133–146.

[7] B. Courcelle, Graph rewriting: an algebraic and logic approach, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 193–242 (in press).

[8] B. Courcelle, On using context-free graph grammars for analyzing recursive definitions, in: K. Fuchi and L. Kott, eds., Programming of Future Generation Computers II (Elsevier/North-Holland, Amsterdam, 1988) 83–122.

[9] G. Gardarin, I. Guessarian and C. De Maindreville, Translation of logic programs into functional fixpoint equations, *Theoret. Comput. Sci.* **63** (1989) 253–276.

[10] G. Gardarin and E. Simon, Les systèmes de gestion de bases de données déductives, *Tech. et Sci. Inform.* **6** (1987) 347-382.
[11] R. Kowalski and H. Van Emden, The semantics of predicate logic as a programming language, *J. Assoc. Comput. Mach.* **23** (1976) 733-742.
[12] J.L. Lassez, V. Nguyen and E. Sonenberg, Fixed points theorems and semantics: a folktale, *Inform. Process. Lett.* **14** (1982) 112-116.
[13] J. Mezei and J. Wright, Algebraic automata and context-free sets, *Inform. and Control* **11** (1967) 3-29.
[14] J. Naughton, Redundancy in function-free recursive inference rules, in: *Proc. IEEE Symp. on Logic Programming*, Salt Lake City, UT (1986) 236-245.
[15] J. Naughton and Y. Sagiv, Minimizing expansions of recursions, in: H. Ait-Kaci and M. Nivat, eds., *Resolution of Equations in Algebraic Structures: Volume 1, Algebraic Techniques* (Academic Press, Boston, MA, 1989) 321-349.
[16] L. Vieille, Recursive query processing: the power of logic, *Theoret. Comput. Sci.*, to appear.