Logic Programming as Hypergraph Rewriting*

Andrea Corradini, Francesca Rossi
Università di Pisa
Dipartimento di Informatica
Corso Italia 40
56125 Pisa - ITALY

Francesco Parisi-Presicce
Università dell'Aquila
Dipartimento di Matematica
I-67100 L'Aquila - ITALY

Abstract Logic Programming and (Hyper-)Graph Rewriting are two well known fields of Computer Science. In this paper we show how to model logic program computations through algebraic techniques familiar to the graph rewriting community. Clauses of a logic program are represented by graph productions, goals by suitable hypergraphs (called jungles), and resolution steps by an algebraic construction involving three pushouts. The correspondence between the two formalisms is further analyzed by providing a precise algebraic characterization of specialization and unfolding of clauses.

1 Introduction

The "theory of graph-grammars" basically studies a variety of formalisms which extend the theory of formal languages in order to deal with structures more general than strings, like graphs and maps. A graph-grammar allows to describe finitely a (possibly infinite) collection of graphs, i.e., those graphs which can be obtained from an initial graph through repeated application of graph productions. The form of graph productions, and the rule stating how a production can be applied to a graph and what the resulting graph is, depend on the specific formalism.

The development of this theory, originated in the late 60s, is well motivated by many fruitful applications in different areas of computer science: among them we recall data bases, software specification, incremental compilers, and pattern recognition (cf. [CER79, ENR83, ENRR87, EKR91], the proceedings of the first four international workshops on graph-grammars).

---

* Research supported by the GRAGRA Basic Research Esprit Working Group n. 3299.

Although there is a fairly natural correspondence between some notions of logic programming and the basic ingredients of graph-grammars, to our knowledge the relationship between these two areas has not been considered in depth yet. Actually, the concepts of goal, clause and refutation are very close to the notions of graph, graph production and graph derivation, respectively. This correspondence has been exploited for example in [RM88], where the graph-grammar behaviour is implemented in logic programming for modeling the generation and solution of "hierarchical networks of constraints", and in [RM90], where it is used to describe a relaxation algorithm scheme (for constraint satisfaction problems) in logic programming. Both papers reformulate a problem, easily expressible in the theory of graph-grammar, as a logic program, taking advantage of the executability of such programs.

In this paper we take the dual approach of representing goals and clauses of a logic program as suitable graphs and graph productions, respectively, and we model a resolution step with a clean algebraic construction, expressed in terms of pushouts in a suitable category. In a recent joint work with Ehrig, Löwe, and Montanari [CMREL91], the first two authors already addressed this problem. However, there they proposed a quite different representation of clauses (transforming them first into a 'canonical form'), resulting in a less intuitive correspondence (from the logic programming point of view) between a direct graph derivation and a resolution step. In this paper, the canonical form of clauses is abandoned, in favour of a more natural representation which matches the logic programming spirit better.

The meaning and the usefulness of such a mapping (between logic programs and graph grammars) can be obviously found in a potential 'cross-fertilization' of the two fields. For example,

1) logic programming can provide an efficiently executable representation of a class of graph-grammars;
2) well known graph grammar techniques can be applied to logic programs, providing new tools for program transformation and metaprogramming;
3) a rich collection of results about parallelism and concurrency in the graph-grammar theory could be exploited in the logic programming framework, in order to formally analyze and prove properties of parallel execution frameworks.

This paper represents a first step towards a deeper analysis of the points just stressed. Besides proving the correctness of our translation, we consider some results of graph-grammar theory, and we apply them to logic programs in this algebraic framework. For example, we show how program clauses (represented as graph productions) can be combined in a clean way in order to produce new clauses. These manipulation operations on clauses have strict relationships with the classic notions of unfolding and partial evaluation.

Among the various formulations of graph rewriting, we consider the so called 'algebraic theory of graph-grammars' [Eh87], which provides a rich collection of formal results, mainly in the field of concurrent and parallel rewriting [Eh83, Kr87]. In Section 2 we briefly introduce the main ingredients of this theory, in the more general variant of *hypergraph* rewritings. In fact, the atomic formulas appearing in a logic program can be easily represented by a particular kind of hypergraphs, called *jungles*. Logic programs are introduced in Section 3, while jungles and their correpondence with terms and formulas are presented in Section 4, toghether with the representation of program clauses by productions. In Section 5 we discuss in depth how a resolution step is modeled by an algebraic construction involving three pushouts in the category of jungles. Section 6 is devoted to the description of some constructions which enrich a program with new clauses, preserving its semantics.

2 A short introduction to hypergraph rewriting

In this section we present the basic concepts of the algebraic theory of graph rewriting, as summarized for example in [Eh87]. However, since the atomic formulas and the terms of a logic programs are easily represented by some special kind of *hypergraphs*, we consider the generalization of that theory to hypergraph rewriting [HK87].

A (directed) hypergraph straightforwardly generalizes a (directed) graph: it includes a set of nodes and a set of hyperarcs. Every hyperarc has a (possibly empty) list of source nodes and a list of target nodes, instead of exactly one source and one target node.

2.1 Definition *(hypergraphs)*

Let $C = (C_V, C_E)$ be a fixed pair of color sets. A (colored, directed) **hypergraph** G is written as $G = (V, E, s, t, m, l)$, where

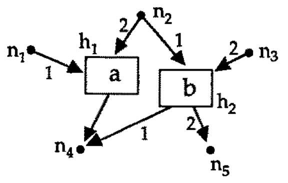
- V is a set of *nodes* (or *vertices*);
- E is a set of *(hyper)arcs* (or *hyperedges*),
- $s,t: E \to V^*$ are the source and target function respectively, assigning a tuple of source nodes and target nodes to each hyperarc, and
- $l: V \to C_V$ and $m: E \to C_E$ are the coloring functions for nodes and arcs respectively. ♦

In the following we will never consider colored nodes, so a hypergraph will always be written as $G = (V, E, s, t, m)$, and $C \equiv C_E$ is the fixed set of edge colors.

2.2 Example *(graphical representation of hypergraphs)*

Consider the hypergraph $G = (\{n_1, n_2, n_3, n_4, n_5\}, \{h_1, h_2\}, s, t, m)$, where

- $s(h_1) = \langle n_1, n_2 \rangle$, $s(h_2) = \langle n_2, n_3 \rangle$;
- $t(h_1) = \langle n_4 \rangle$, $t(h_2) = \langle n_4, n_5 \rangle$;
- $m(h_1) = a$, $m(h_2) = b$.



In the graphical representation of hypergraphs (like G in the above picture), we will take the following conventions. Black dots represent nodes, while hyperarcs are depicted as boxes with as many "tentacles" as the total number of source and target nodes. Each tentacle connects one node to the box: tentacles connecting to source nodes are oriented towards the box, while tentacles connecting to target nodes are oriented towards the node. The hyperarc color is written inside the box, and each tentacle is numbered to express the ordering in the tuples of source and target connections. The numbering can be avoided if there is exactly one source or target node. ♦

2.3 Definition *(hypergraph morphisms)*

A **hypergraph morphism** $f: G_1 \to G_2$ consists of a pair of functions between arcs and nodes respectively, which are compatible with the source and target functions and which are color preserving. More precisely, if $G_1 = (V_1, E_1, s_1, t_1, m_1)$ and $G_2 = (V_2, E_2, s_2, t_2, m_2)$ are hypergraphs, then $f = (f_V, f_E)$ such that

- $f_V: V_1 \to V_2$, and $f_E: E_1 \to E_2$;
- $s_2 \circ f_E = f^*_V \circ s_1$;
- $t_2 \circ f_E = f^*_V \circ t_1$;
- $m_2 \circ f_E = m_1$.

where f^*_V is the obvious extension of f_V to lists of nodes. A hypergraph morphism $f = (f_V, f_E): G_1 \to G_2$ is *injective* iff both f_V and f_E are injective functions. ♦

2.4 Definition *(the category of hypergraphs $\mathbf{HGraph}_C$)*

The category whose objects are hypergraphs colored on C, and whose arrows are hypergraph morphisms will be called $\mathbf{HGraph}_C$. ♦

We need now the categorical definition of pushout, which is extensively used in the algebraic description of graph grammars.

2.5 Definition *(pushout)*

Given any category C and two arrows $b: K \to B$, $d: K \to D$ of C, an object H together with two arrows $h: B \to H$ and $c: D \to H$ is called a **pushout of b and d** if:

- (commutativity property) $h \circ b = c \circ d$;
- (universal property) for all objects H' and arrows $h': B \to H'$ and $c': D \to H'$, with $h' \circ b = c' \circ d$, there exists a unique arrow $f: H \to H'$ such that $f \circ h = h'$ and $f \circ c = c'$. This situation is depicted in the following picture.

$$\begin{tikzcd}
K \arrow[r, "b"] \arrow[d, "d"'] & B \arrow[d, "h"'] \arrow[dr, "h'"] & \\
D \arrow[r, "c"] \arrow[rrd, "c'"'] & H \arrow[dr, dashed, "f"] & \\
 & & H'
\end{tikzcd}$$

Moreover, in a pushout square like this one, the object D (together with arrows d and c) is called a **pushout complement** of b and h. ♦

2.6 Example *(pushouts in Set and in $HGraph_C$)*

The paradigmatical example of category is **Set**, i.e., the category having *sets* as objects and *total functions* as arrows. It is easy to show that the pushout of two arrows in **Set** always exists, and that it is characterized (up to isomorphism) as follows. If $b: K \to B$ and $d: K \to D$ are two functions, then the pushout of $\langle b, d \rangle$ is the set $H \equiv (B + D)/_{\approx}$ (where '+' denotes disjoint union, and '$\approx$' is the least equivalence relation such that for all $k \in K$ $b(k) \approx d(k)$), together with the two functions $h: B \to H$ and $c: D \to H$ that send each element to its equivalence class.

Also in category $\mathbf{HGraph}_C$ the pushout of two arrows always exists: it can be computed componentwise (as a pushout in **Set**) for the nodes and the edges, and the source and target mappings are uniquely determined. More precisely, if $X = (V_X, E_X, s_X, t_X, m_X)$ for $X \in \{K, B, D, H\}$ are objects of $\mathbf{HGraph}_C$, and $b = (b_V, b_E): K \to B$ and $d = (d_V, d_E): K \to D$ are hypergraph morphisms, then the pushout H of $\langle b, d \rangle$ can be obtained as follows.

- V_H is the pushout in **Set** of $b_V: V_K \to V_B$ and $d_V: V_K \to V_D$,
- E_H is the pushout in **Set** of $b_E: E_K \to E_B$ and $d_E: E_K \to E_D$,
- $m_H([e]) = \begin{cases} m_B(e) \text{ if } e \in E_B \\ m_D(e) \text{ otherwise} \end{cases}$
- $s_H([e]) = \langle [v_1], \ldots, [v_n] \rangle$, where $\begin{cases} s_B(e) = \langle v_1, \ldots, v_n \rangle \text{ if } e \in E_B \\ s_D(e) = \langle v_1, \ldots, v_n \rangle \text{ if } e \in E_D \end{cases}$
- $t_H([e]) = \langle [v_1], \ldots, [v_n] \rangle$, where $\begin{cases} t_B(e) = \langle v_1, \ldots, v_n \rangle \text{ if } e \in E_B \\ t_D(e) = \langle v_1, \ldots, v_n \rangle \text{ if } e \in E_D \end{cases}$

It can be easily shown that this definition is correct, i.e., it does not depend from the choice of the representative of an equivalence class. ♦

It is worth noting that considering an arbitrary category the pushout of two arbitrary morphisms does not always exist. This is the case, for example, of the category $\mathbf{Jungle}_C$ introduced in Section 4.

A hypergraph rewriting rule, analogously to term rewriting rules, describes how to replace the occurrence of a subgraph L of a graph G with another graph R. While in the case of terms the embedding of R inside G is uniquely determined, this is not true in the more general case of graphs. Thus a third graph K is needed to give the connection points of R in G.

2.7 Definition *(hypergraph rewriting rule)*

A **hypergraph rewriting rule** p is written as $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, where L, R, and K are hypergraphs and l and r are hypergraph morphisms. Also, l is always assumed to be injective. L, K, and R are called the left-hand side (lhs), the interface, and the right-hand side (rhs) of p, respectively. ♦

To apply a rewriting rule p to a graph G, we first need to find an *occurrence* of its lhs L in G, i.e., a morphism g from L to G. Next, to model the deletion of such occurrence of L in G, we construct the pushout complement of g and l, producing the 'context' graph D where the rhs R has to be embedded. Such an embedding is then expressed by a second pushout. It must be noticed that such a construction can fail, since the pushout complement of two arrows does not always exist.

2.8 Definition *(direct rewriting)*

Given a hypergraph G, a hypergraph production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, and an occurrence $g: L \to G$, a **direct rewriting** from G to H exists if the following two pushouts can be constructed. In this case D is called the *context* graph, and we write $G \Rightarrow_p H$.

$$\begin{tikzcd}
L \arrow[d, "g"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$
♦

2.9 Definition *(rewriting)*

Given hypergraphs G and H, and a set of hypergraph productions P, a **rewriting** from G to H over P, denoted by $G \Rightarrow_P^* H$, is a finite sequence of direct rewriting steps of the form $G \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} G_n = H$, where $p_1, \ldots, p_n$ are in P. ♦

3 Logic programming

For an introductory and complete treatment of logic programs, see [Ll87]. Here we introduce just the concepts which will be used in the rest of the paper.

Let $\Sigma = \cup_n \Sigma_n$ be a ranked set of function symbols, where Σ_n is the set of functions of rank n (in particular, Σ_0 is the set of constants). Also, let $\Pi = \cup_n \Pi_n$ be a ranked set of predicate symbols (where Π_n is the set of predicates with n arguments), and let X be a set of variables. A *term over Σ* is an element of $T_\Sigma(X)$, the free Σ-algebra generated by X, that is

- a variable in X, or
- a constant in Σ_0, or
- $f(t_1, \ldots, t_n)$, if $t_1, \ldots, t_n$ are terms over Σ, and $f \in \Sigma_n$.

If $t_1, \ldots, t_n$ are terms over Σ, and $p \in \Pi_n$, then $p(t_1, \ldots, t_n)$ is an *atomic formula over (Σ, Π)*. A *(conjunctive) formula* is a list of atomic formulas separated by commas, like $B_1, \ldots, B_n$.

A *definite clause* C is an expression of the form

$$H :\!- B_1, \ldots, B_n \qquad (n \geq 0)$$

where ':-' means logic implication (right to left), and ',' means logical conjunction.

A *goal G* is a formula of the form

$$A_1, \ldots, A_n \qquad (n > 0)$$

A *logic* (or *HCL*) *program P* is a finite set of definite clauses.

A logic program can be interpreted in many different but equivalent ways (see [Ll87] for a formal treatment of all of them). As a first order theory, its semantics is defined as its *least Herbrand model*. Under the operational reading, instead, a *resolution rule* states how to transform a goal into another. The operational semantics is then defined as the set of all (ground) atomic formulas which can be transformed into the empty goal through a sequence of resolution steps. The equivalence between the operational and model-theoretic semantics of a program is proved by showing that the resolution inference rule is both sound and complete for definite clauses. In this paper, since we are interested in showing that the operational behaviour of a program can be faithfully simulated in the graph grammar context, we will focus on the operational semantics, that is on the answer substitutions computed by a program for a given goal, as defined in the rest of this section.

Given a set of variables X and a ranked set of function symbols Σ, a *substitution* is a function $\theta: X \rightarrow T_\Sigma(X)$ which is the identity on all but a finite number of variables. Thus, θ can be written as $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$, where $\theta(x_i) = t_i$ for each $1 \leq i \leq n$, and $\theta(x) = x$ otherwise.

The *application* of a substitution $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ to a term t produces a new term (written $t\theta$) obtained from t by replacing each occurrence of x_i with t_i, for $1 \leq i \leq n$. This definition, and the corresponding notation, is also used for the application of substitutions to formulas. Given two substitutions θ and δ, θ is said to be *more general* than δ if there exists a substitution ω such that $\omega \circ \theta = \delta$ (where $\omega \circ \theta$ is such that, for all terms t, $t(\omega \circ \theta) = (t\theta)\omega$).

Two atomic formulas A and B *unify* if there exists a substitution θ such that $A\theta = B\theta$. In this case θ is called a *unifier* of A and B. The set of unifiers of any two atomic formulas is either empty, or it has a most general element (up to variable renaming) called the *most general unifier (mgu)*.

Given a clause $C = (H \text{ :- } B_1, \ldots, B_n)$ and a goal $G = (G_1, \ldots, G_n)$, G' is the *resolvent* of G and C if

- (unification step) there exists G_i and a substitution θ which is the mgu of G_i and H;
- (rewriting step) G' is $(G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta$.

In this case we will say that there is a *resolution step from G to G' via C and θ.*

A *refutation* of a goal G is a finite sequence of resolution steps which starts with G and ends with the empty goal. If the refutation has length n, where step i uses clause C_i and the mgu θ_i, then the substitution $\theta = (\theta_n \circ \ldots \circ \theta_1) \mid_{Var(G)}$ (i.e., the restriction of $\theta_n \circ \ldots \circ \theta_1$ to the variables appearing in G), is called a *computed answer substitution* for G. In this case we say that there is a *refutation of G via $C_1, \ldots, C_n$ and $\theta = (\theta_n \circ \ldots \circ \theta_1) \mid_{Var(G)}$.*

4 Logic programs as jungle rewriting

Jungles were introduced in [HKP88] as special hypergraphs which allow to represent faithfully collections of terms with possibly shared subterms. A hyperarc of a jungle is colored with an operator of a fixed signature Σ, and the number of its outgoing tentacles must be equal to the arity of the operator. Every node of a jungle uniquely determines a term built from operators in Σ. On the other hand, a term can be represented by many non isomorphic jungles, since common subterms can be either collapsed or duplicated.

In [HKP88] it is shown how the rules of a term rewriting systems can be represented as productions (as defined in the previous section), where the left and right hand sides are jungles. Moreover the double pushout construction described above faithfully models the application of a rewrite rule to a term. Actually, since many occurrences of the same subterm can collapse in a jungle, a single derivation step can model the application of the same rule to many distinct subterms of the original term.

It must be mentioned that an alternative representation of terms with graphs could have been considered, using directed acyclic graphs as defined in [PEM87]. However, by a result presented in [CMREL91] the two representations are completely equivalent, in the sense that for a given signature Σ the categories $\mathbf{Jungle}_\Sigma$ of jungles over Σ, and the category $\mathbf{DAG}_\Sigma$ of dags over Σ are equivalent. A direct consequence of this result is that all the results about dags automatically hold for jungles, and viceversa.

The presentation in [HKP88] considers many sorted signatures, coloring the nodes of a jungle with sorts. Since we want to model (collections of) formulas of a logic program, in the following we will consider a fixed two sorted signature including predicate and function symbols as the set of edge colors. Colors for nodes are not needed, because in the syntax of a logic program predicate symbols cannot be nested.

4.1 Definition *(the category of jungles over (Σ, Π))*

In the following, let (Σ, Π) be a fixed two sorted signature, where Σ is a (ranked) set of function symbols, and Π is a (ranked) set of predicate symbols. A hypergraph $G = (V_G, E_G, s_G, t_G, m_G)$ is a **jungle over (Σ, Π)** iff

- G is acyclic[1]
- *outdegree*$_G$(v) ≤ 1 for each v $\in V_G$[2]
- for each e $\in E_G$, $m_G(e) \in \Sigma \cup \Pi$. Moreover,

  if $m_G(e) \in \Sigma_n$ then $\#s_G(e) = 1$ and $\#t_G(e) = n$;

  if $m_G(e) \in \Pi_m$ then $\#s_G(e) = 0$ and $\#t_G(e) = m$, (where #t denotes the length of the tuple t).

A **jungle morphism** h: G $\rightarrow$ G' is simply a hypergraph morphism. We denote by **Jungle**$_{\Sigma,\Pi}$ the category including jungles over (Σ, Π) as objects, and jungle morphisms as arrows. ♦

Intuitively, the outgoing tentacles connect an edge to the arguments of the (function or predicate) symbol coloring it, while the ingoing tentacles are used to compose subterms. The fact that edges colored by predicate symbols have no source nodes corresponds to the requirement that an atomic formula cannot appear as a proper subterm.

4.1 Representing formulas with jungles

In this subsection we decribe how both terms and formulas can be represented as jungles. First we show how to extract a term from each node of a jungle. Indeed, by the second condition in the definition of a jungle, each node has either exactly one outgoing tentacle, or none. In the former case it is the root of a sub-jungle which represents a term, while in the latter case it represents a variable in itself. Analogously, an atomic formula can be extracted from every edge colored by a predicate symbol.

4.2 Definition *(from nodes to terms, from edges to atomic formulas)*

Let G be a jungle. The set Var$_G$ of *variables of G* is defined as

$Var_G = \{v \in V_G \mid outdegree_G(v) = 0\}$.

The function *term*$_G$ associates to each node in G a term in $T_\Sigma(Var_G)$, and is defined as follows:

- $term_G(v) = v$ if $v \in Var_G$
- $term_G(v) = op(term_G(v_1), \ldots, term_G(v_n))$ if there exists $e \in E_G$ with $s_G(e) = v$, $t_G(e) = \langle v_1 \ldots v_n \rangle$, and $m_G(e) = op \in \Sigma_n$.

The function *form*$_G$ associates to each edge colored by a predicate symbol an atomic formula over $T_\Sigma(Var_G)$, and it is defined as

- $form_G(e) = pred(term_G(v_1), \ldots, term_G(v_m))$ if $m_G(e) = pred \in \Pi_m$, and $t_G(e) = \langle v_1, \ldots, v_n \rangle$.

The (multi-) set of all formulas which can be extracted from a jungle G is denoted by *FORM*$_G$, i.e.,

- $FORM_G = \{form_G(e) \mid e \in E_G \text{ and } m_G(e) \in \Pi\}$. ♦

4.3 Example *(a jungle of category Jungle*$_{\Sigma,\Pi}$*)*

In the following jungle we have function symbols *cons* $\in \Sigma_2$, and *nil* $\in \Sigma_0$ (a constant)[3], and the predicate symbols ***reverse*** $\in \Pi_2$ and ***append*** $\in \Pi_3$ (notice that the edges with bold labels have no ingoing tentacle, thus by

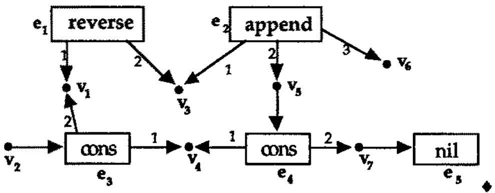
[1] For a hypergraph G, its *underlying bipartite graph* U(G) includes all nodes and hyperarcs of G as nodes, and the tentacles of G as arcs. Then G is acyclic iff U(G) is acyclic.

[2] Informally, the *outdegree* of a node is the number of tentacles outgoing from it.

[3] Throughout the paper we use functional notation for the list constructors. A PROLOG programmer should read *nil* as [] , and *cons(X, Y)* as [X | Y].

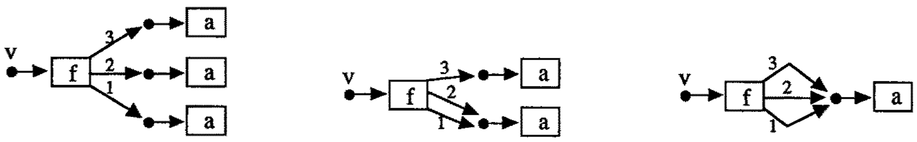
Definition 4.1 their labels are predicate symbols). There are four variable nodes (v_1, v_3, v_4 and v_6), and the functions *term* and *form* are defined as follows:

term(v_i) = v_i for i = 1, 3, 4, 6	*term*(v_2) = cons(v_4, v_1)
term(v_5) = cons(v_4, nil)	*term*(v_7) = nil
form(e_1) = **reverse**(v_1, v_3)	*form*(e_2) = **append**(v_3, cons(v_4, nil), v_6)



♦

Although from nodes and (predicate colored) edges of a jungles one can extract in a unique way terms and atomic formulas respectively, each term or formula can have many (possibly non isomorphic) representations as jungles. In fact common subterms can either be collapsed in a unique representation, or can be replicated for each occurrence[4]. For example, the term f(a,a,a) can be represented in five ways, three of which are depicted here (more precisely, in all the jungles below we have *term*(v) = f(a,a,a)).



Among the many possible representations of a term (or formula) as a jungle, there are two which are special, as described in the following definition.

4.4 Definition *(representing terms and formulas as jungles)*

Let t be a term in $T_\Sigma(X)$. Then its *variable-collapsed tree* [HKP88] is its most redundant representation as jungle, in which two subterms are collapsed iff they are occurrences of the same variable. We denote the function which assigns to each term its variable-collapsed tree by j: $T_\Sigma(X) \to \mathbf{Jungle}_{\Sigma,\Pi}$. Function j can be extended in the obvious way to sets of terms and/or formulas.

On the other hand, we denote by J: $T_\Sigma(X) \to \mathbf{Jungle}_{\Sigma,\Pi}$ the function which assigns to a term t its *fully collapsed tree* J(t), i.e., its less redundant representation where there are no distinct nodes v and v' such that *term*(v) = *term*(v'). J can be extended to sets of terms and/or formulas, too. Note that if X is the set of variables occurring in t, then X is exactly the set of variable nodes in j(t) and J(t). ♦

For example, among the three possible representations of the term f(a,a,a) depicted above, the leftmost is j(f(a,a,a)), while the rightmost is J(f(a,a,a)). It is easy to check that for a given term t, j(t) and J(t) are uniquely defined (up to isomorphism), and that these two representations enjoy the following property: if G is any

[4] In this discussion we consider jungles which include just the nodes and edges needed to represent a given term or formula.

representation of the term t, then there is exactly one morphism from j(t) to G, and exactly one from G to J(t). In other words, j(t) is *initial* and J(t) is *final* among all the representations of t.

It must be stressed that the results presented in the following sections do not depend on the actual representation chosen for formulas and terms. This fact provides a degree of freedom which can be exploited, for example, for increasing the efficiency of an implementation.

Let us continue the analysis of the correspondence between the category $\mathbf{Jungle}_{\Sigma,\Pi}$ and the world of terms and formulas over (Σ, Π). We show that from a morphism in $\mathbf{Jungle}_{\Sigma,\Pi}$ a term substitution can be extracted, and that pushouts in $\mathbf{Jungle}_{\Sigma,\Pi}$ are strictly related to most general unifiers.

4.5 Definition *(the substitution associated to a morphism)*

Let G and G' be jungles, and $h: G \to G'$ be a jungle morphism. Then the node components of h (i.e., h_V) induces a substitution σ_h: $Var_G \to T_\Sigma(Var_{G'})$, defined as

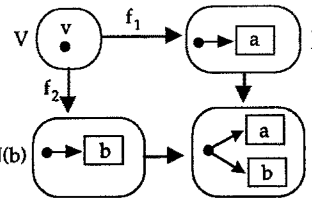
$$\sigma_h(x) = term_{G'}(h_V(x)).$$

As a consequence, the morphism h maps every term or formula of G to an instantiation of it in G'. In particular, we define the extension of h to formulas of G, h_F: $FORM_G \to FORM_{G'}$, as

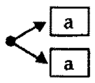
$$h_F(form_G(e)) = form_{G'}(h_E(e)). \ \blacklozenge$$

It is important to note that the correspondence between jungle morphisms and term substitutions is preserved by composition, in the following sense: if $h: G \to G'$ and $k: G' \to G''$ are jungle morphisms then $\sigma_{k \circ h} = \sigma_k \circ \sigma_h$, where $\sigma_k \circ \sigma_h$: $Var_G \to T_\Sigma(Var_{G''})$ is the composition of term substitutions. Analogously to the fact that the representation of terms as jungles is not unique, given a jungle G and a term substitution σ: $Var_G \to T_\Sigma(X)$, there exist in general many pairs $\langle G', h: G \to G' \rangle$ such that $\sigma \equiv \sigma_h$.

Although $\mathbf{Jungle}_{\Sigma,\Pi}$ is a sub-category of $\mathbf{HGraph}_{\Sigma,\Pi}$ (i.e., the category of hypergraphs with edge labels in (Σ, Π)), not every pair of morphisms has a pushout in $\mathbf{Jungle}_{\Sigma,\Pi}$ (but the pushout always exists in $\mathbf{HGraph}_{\Sigma,\Pi}$, cf. Example 2.6). For example, let V be the jungle containing just one variable node v, and let f_1: $V \to J(a)$ and f_2: $V \to J(b)$ be two morphisms (J is as in Definition 4.4). Then the pushout of $\langle f_1, f_2 \rangle$ exists in $\mathbf{HGraph}_{\Sigma,\Pi}$ (as it is depicted in the diagram below), but not in $\mathbf{Jungle}_{\Sigma,\Pi}$. In fact, the hypergraph closing the square in the picture is not a jungle because it does not satisfy the second condition of Definition 4.1.



Even if the pushout of two arrows exists in $\mathbf{Jungle}_{\Sigma,\Pi}$ it does not necessarely coincides with the pushout of the same two arrows in $\mathbf{HGraph}_{\Sigma,\Pi}$. For example, if we consider the pair of morphisms $\langle f_1, f_1 \rangle$, its pushout object in $\mathbf{Jungle}_{\Sigma,\Pi}$ is J(a), while its pushout in $\mathbf{HGraph}_{\Sigma,\Pi}$ is the hypergraph



The next result shows that the existence of a pushout in $\mathbf{Jungle}_{\Sigma,\Pi}$ can be stated in terms of the existence of suitable term unifiers.

4.6 Proposition *(existence of pushouts in $Jungle_{\Sigma,\Pi}$)*

1) Let r: K → R and d: K → D be two morphisms in $\mathbf{Jungle}_{\Sigma,\Pi}$. Let σ_r: $Var_K \to T_\Sigma(Var_R)$ and σ_d: $Var_K \to T_\Sigma(Var_D)$ be the associated substitutions. Then the pushout of ‹r, d› exists in $\mathbf{Jungle}_{\Sigma,\Pi}$ iff there exist two substitutions θ: $Var_R \to T_\Sigma(X)$ and θ': $Var_D \to T_\Sigma(X)$ which "unify" $\langle\sigma_r, \sigma_d\rangle$, in the sense that $\theta \circ \sigma_r = \theta' \circ \sigma_d$.

$$\begin{tikzcd} K \arrow[r, "r"] \arrow[d, "d"'] & R \arrow[d, "g"] \\ D \arrow[r, "f"] & H \end{tikzcd}$$

2) If ‹g, f› is a pushout of ‹r, d› in $\mathbf{Jungle}_{\Sigma,\Pi}$, as in the above diagram, then $\langle\sigma_g, \sigma_f\rangle$ is the most general unifier of $\langle\sigma_r, \sigma_d\rangle$, i.e., for each pair of substitutions $\langle\theta, \theta'\rangle$ such that $\theta \circ \sigma_r = \theta' \circ \sigma_d$, there exists a unique σ such that $\sigma \circ \sigma_g = \theta$ and $\sigma \circ \sigma_f = \theta'$.

Proof sketch. For point 1) the *only if* part is obvious (consider the substitutions associated to the pushout morphisms and apply the commutativity property), while the *if* part follows from the fact that in category $\mathbf{Jungle}_{\Sigma,\Pi}$ there exist all colimits over *consistent* diagrams (i.e., diagrams for which a cocone exists: cf. [Ke91]). Point 2) follows from the observation that the universal property of pushouts perfectly corresponds to the defining property of mgu's. ♦

It is worth noting that the relationship between most general unifiers and categorical universal constructions has been stressed in many places. For example, mgu's are characterized in [RB85] (resp. [Go88]) in terms of coequalizers (resp. equalizers, due to the dual approach), and also as pullbacks in [AM89]; in these papers terms and substitutions are represented as arrows of a category. On the contrary, our characterization is much closer to the one in [PEM87], where terms are objects and mgu's are pushouts.

The following result gives a sufficient condition for the existence and the uniqueness of pushout complements in the category $\mathbf{Jungle}_{\Sigma,\Pi}$: because of the additional structure these conditions are slightly different from the corresponding ones for hypergraphs, stated in [Ha89].

4.7 Proposition *(existence of PO-complement in $Jungle_{\Sigma,\Pi}$)*

Let l: K → L be an injective jungle morphism.

1) If g: L → G (as in the diagram below), then the pushout complement of ‹l,g› exists iff

• *dangling condition:* all the nodes of L which are mapped by g to a node of G connected to an edge which is not in the image of g, are in the image of l. Formally,

$\forall v \in V_L\ (\exists e \in E_G \setminus gL \text{ s.t. } (e = s_G(g_V(v)) \vee e \in t_G(g_V(v))) \Rightarrow \exists v' \in V_K \text{ s.t. } l_V(v') = v.$

• *identifying condition:* all pairs of distinct nodes (or predicate-colored edges) of L which are identified by g are in the image of l. Formally,

$\forall v, v' \in V_L\ (v \neq v' \wedge g_V(v) = g_V(v')) \Rightarrow \exists v'' \in V_K$ such that $l_V(v'') = v$,

$\forall e, e' \in E_L\ (e \neq e' \wedge m_L(e) \in \Pi \wedge g_E(e) = g_E(e')) \Rightarrow \exists e'' \in E_K$ such that $l_E(e'') = e$.

$$\begin{tikzcd}
L \arrow[d, "g"] & K \arrow[l, "l"'] \\
G &
\end{tikzcd}$$

2) For any morphism $g: L \to G$, such that the above conditions are satisfied, the pushout complement of $\langle l,g \rangle$ is *unique* if for each node $v \in Var_K$, $l_V(v) \in Var_L$.

3) If l is a bijection on nodes, in any pushout complement ($K \to H \to G$) of $\langle l, g \rangle$ the morphism $H \to G$ is a bijection on nodes, too.

Proof sketch. 1) The *dangling* and *identifying* conditions are the standard conditions for the existence of pushout complements for graphs, hypergraphs and other structures (cf. [Eh87, Ha89, EHKP90]). Here the identifying condition is slightly relaxed: it is not necessary (for the existence of at least one pushout complement) for Σ-colored edges, because they are in one-to-one correspondence with their source nodes. 2) If $l_V(v)$ is a variable of L whenever v is a variable of K, then it can be shown that the identifying condition holds for Σ-colored edges, too, and thus the PO-complement is unique. 3) The nodes of the pushout complement are the pushout complement in **Set** of the restriction to nodes of the above diagram. Thus since l_V is a bijection, $(H \to G)_V$ has to be a bijection too. ♦

This result will be exploited in the next subsection, where program clauses will be represented as jungle rewriting rules in such a way that the two conditions just stated are always satisfied.

4.2 Representing clauses as jungle rewriting rules

After understanding how to represent a collection of terms or atomic formulas as jungles, we show how a program clause can be represented by a *jungle rewriting rule*, i.e., a hypergraph rewriting rule with the following shape, but where arrows and objects are in category $\mathbf{Jungle}_{\Sigma,\Pi}$:

$$\begin{tikzcd}
L & K \arrow[l, "l"'] \arrow[r, "r"] & R
\end{tikzcd}$$

4.8 Definition *(representing clauses as rewriting rules)*

Let $C = H$:- $B_1, \ldots, B_n$ be a program clause, and let $H = p(t_1, \ldots, t_m)$. Then the representation of C as jungle rewriting rule, J(C), is defined as follows:

- L is the fully collapsed tree of H, i.e., L = J(H).
- K is obtained from L by removing the unique edge colored by the predicate symbol p. Morphism l is the obvious inclusion.
- R is the fully collapsed representation of the union of the atomic formulas $B_1, \ldots, B_n$, and the terms $t_1, \ldots, t_m$. Morphism r is the obvious inclusion. ♦

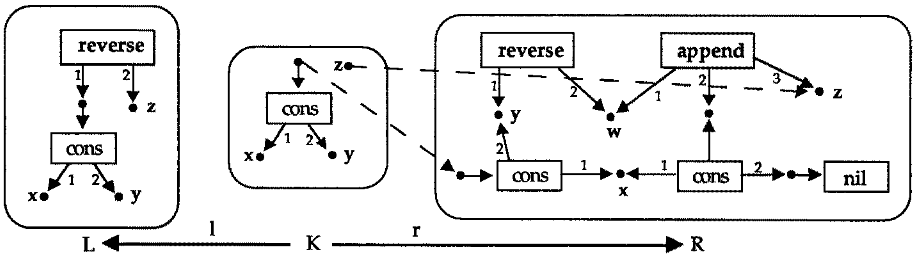
It could appear unnatural that the arguments $t_1, \ldots, t_m$ of the predicate p in the head of the clause H are represented also in the interface jungle K and in the right hand side R. However, this is necessary for two reasons. First, by Proposition 4.7 2), this guarantees the uniqueness of the PO-complement when applying the rewriting rule in a direct rewriting step. Second, terms $t_1, \ldots, t_m$ could be mapped to a shared term in a goal G. If we remove one of them, say t_i, from both K and R, in a rewriting step we could delete from the goal G not only the occurrence of t_i from the selected atomic formula, as it would be correct, but also some occurrences of the same term in other formulas of the goal, which is obviously incorrect.

4.9 Example *(jungle representation of a clause)*

The following diagram shows the jungle production representing the clause

reverse(cons(x, y), z) :- reverse(y, w), append(w, cons(x, nil), z)

The morphism l: K → L is the obvious inclusion, while the morphism r: K → R is uniquely determined by the two dotted arrows.



♦

5 Resolution as jungle pushout + rewriting

We recall here the definition of resolution step given in Section 3. Suppose we have the following goal G and clause C:

G: $G_1, \ldots, G_n$.

C: H:- $B_1, \ldots, B_m$.

Then G' is the resolvent of G and C if

- (unification step) there exists G_i and a substitution θ which is the mgu of G_i and H;
- (rewriting step) G' is $(G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\ \theta$.

We will now show that both the above steps can be faithfully modelled in the jungle rewriting framework introduced in the above sections. In particular, the unification step will be represented by a pushout in the category $\mathbf{Jungle}_{\Sigma,\Pi}$, while the rewriting step will be expressed as a double pushout in the same category.

5.1 Unification as jungle pushout

As a byproduct of Proposition 3.6 in Section 3.1, which states the conditions for the existence of the pushout object in the category $\mathbf{Jungle}_{\Sigma,\Pi}$, we have the following corollary.

5.1 Corollary *(pushout and unification)*

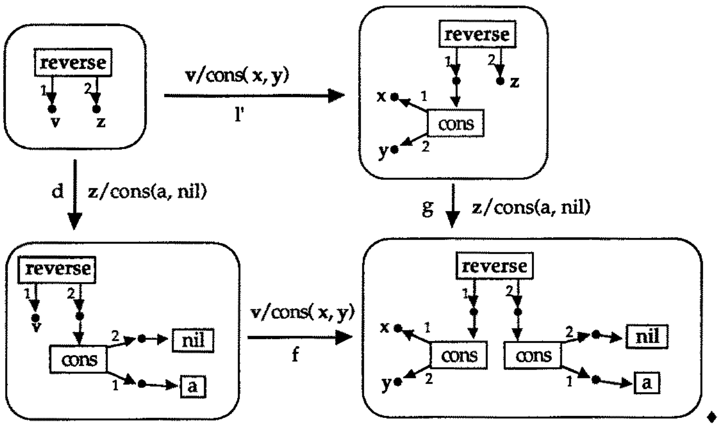
Suppose that in the diagram below W is an arbitrary jungle, while L' is a jungle consisting of only one arc *(e)* colored by $p \in \Pi_m$ with m tentacles going to m *distinct* nodes. Also, let L be a jungle representing a single atomic formula, i.e., containing only one predicate colored hyperarc, whose color is p, and let l': L' → L be the obvious morphism. Then the pushout of l' and d exists iff the atomic formula $d_F(\mathit{form}_{L'}(e))$ in W unifies with the atomic formula $l'_F(\mathit{form}_{L'}(e))$ in L. In this case $\mathit{FORM}_Q = (\mathit{FORM}_W)\theta'$, where θ' is the restriction to Var_W of $\theta = \mathrm{mgu}(d_F(\mathit{form}_{L'}(e)), l'_F(\mathit{form}_{L'}(e)))$.

$$\begin{tikzcd} L' \arrow[r, "l'"] \arrow[d, "d"'] & L \arrow[d, "g"] \\ W \arrow[r, "f"] & Q \end{tikzcd}$$ ♦

Let us rephrase the statement of the Corollary for the case we are interested in, that is for the unification performed during a resolution step: if L represents the head of clause C (i.e., L = J(H)), W represents a goal G (i.e., $FORM_W$ = G), and d selects the atomic goal G_i of G (i.e., $d_F(form_{L'}(e)) = G_i$), then the above pushout exists iff clause C is applicable to the goal G, being θ the mgu of H and G_i. In this case, Q represents Gθ (i.e., $FORM_Q$ = Gθ), and g_F maps H to $G_i\theta$.

5.2 Example *(unification step)*

The diagram below shows the unification step performed in order to apply the clause of Example 4.9 to the goal *reverse(v, cons(a, nil))*. The diagram forms a pushout in the category $\mathbf{Jungle}_{\Sigma,\Pi}$, and the substitutions associated to each morphisms are made explicit. For sake of simplicity, global names are given to the variables.



5.2 Goal rewriting as jungle rewriting

We showed in Section 4.2 how to represent a program clause C as a jungle rewriting rule J(C). We will now show that the rewriting of a goal G via C can be represented by the application of J(C) to J(G).

Consider the following double pushout diagram.

$$\begin{tikzcd} L \arrow[d, "g"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\ Q & D \arrow[l] \arrow[r] & Q' \end{tikzcd}$$

Let us assume that $(L \xleftarrow{l} K \xrightarrow{r} R)$ = J(C) for some clause C, and that Q has been obtained as a pushout object as described in the previous subsection, and thus it represents the instantiated goal Gθ. We know, by the construction of J(C), that l is bijective over nodes, and that the hypotheses of point 2) of Proposition 4.7 are satisfied. Therefore the pushout complement D exists and it is unique. It is easy to check that, if $G\theta = (G_1, \ldots, G_n)\theta$ and the morphism g: $L \to Q$ maps L to $G_i\theta$, as described above, then D represents the "context" goal $(G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n)\theta$.

Now, let us consider the second pushout. By applying Proposition 4.6, we can easily show that this pushout always exists. In fact, by construction, the substitution σ_r associated to arrow r is the identity substitution, and thus it is possible to unify it with any other substitution (in particular with the one induced by the arrow $K \to D$), in the sense of Proposition 4.6. Moreover, it can be easily shown that $FORM_{Q'} = (G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta$, i.e., Q' represents exactly the resolvent G' of G and C.

Thus we can conclude that a resolution step can be represented by a triple pushout construction, as follows.

$$\begin{tikzcd}
L' \arrow[r] \arrow[d, "d"'] \arrow[dr, phantom, "PO1"] & L \arrow[d] \arrow[dr, phantom, "PO2"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "PO3"] & R \arrow[d] \\
W \arrow[r, "f"] \arrow[rrr, bend right, "k = g_v \circ h_v^{-1} \circ f_v"] & Q & D \arrow[l, "h"'] \arrow[r, "g"] & Q'
\end{tikzcd}$$

The above discussion provides an informal proof of the following theorem.

5.3 Theorem *(resolution step as triple pushout)*

Let $C = (p(t_1, \ldots, t_k) \text{ :- } B_1, \ldots, B_m)$ be a clause, $J(C) = (L \xleftarrow{l} K \xrightarrow{r} R)$ be its jungle representation (as in Definition 4.8), and let $L' \to L$ be the obvious injection of $L' \equiv J(p(x_1, \ldots, x_k))$ in L (where $x_1, \ldots, x_k$ are fresh, distinct variables). Moreover let $G = (G_1, \ldots, G_n)$ be a goal. Then there is a resolution step from G to G' via C and θ if and only if for each jungle W such that $FORM_W = G$ there is a morphism d: $L' \to W$ such that the three pushout diagram above can be constructed.

In this case we say that *J(C) can be applied to W through d yielding Q'*. Moreover we have that

- $FORM_Q = G\theta$, where $\theta = \text{mgu}(H, G_i)$;
- $FORM_D = (G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n)\theta$;
- $FORM_{Q'} = G'$, where $G' = (G_1, \ldots, G_{i-1}, B_1, \ldots, B_m, G_{i+1}, \ldots, G_n)\theta$; thus Q' is a jungle representation of the resolvent of G and C.
- Since h: $D \to Q$ is a bijection on nodes (by Proposition 4.7), its node component has an inverse h_V^{-1}. Let $k = g_V \circ h_V^{-1} \circ f_V$; then σ_k is the restriction of θ to the variables of G. ♦

As formally stated above, PO_1 exists iff H and G_i unify. After the construction of PO_1, the construction of PO_2 and PO_3 is always possible. This reflects the resolution in logic programming, where the unification step is the only one involving possible failure and nondeterminism.

Obviously, such a representation of resolution steps in terms of jungle pushout + direct rewriting can be straightforwardly extended to entire refutations. In our framework, the termination condition of a derivation (corresponding to an empty goal) can be stated as the generation of a jungle including just arcs colored over Σ. By composing the substitutions computed at each step (that is σ_k of the above Theorem), one gets exactly the answer substitution computed by the refutation, corresponding to a morphism from the nodes of the jungle representing the initial goal to the terms represented by the last jungle of the derivation.

The result proved in this section is similar to the one presented in [CMREL91], where a different representation of program clauses as graph productions is used. The two solutions manifest different advantages. Indeed, the representation proposed in [CMREL91] (which first transforms the clauses in a *canonical form*) allows to mimic a resolution step with a double pushout construction, and thus a refutation corresponds to a derivation in the traditional sense of graph-grammars. However, the unification and rewriting parts of a resolution steps are performed simultaneously, unlike the usual behaviour of interpreters for logic languages. On the contrary, with the representation proposed in this paper, the triple pushout construction used to mimic a resolution step perfectly matches its operational definition, although a new notion of *derivation* should be defined for the grammars representing logic programs, where each step is composed by three (instead of two) pushouts.

6 Manipulating clauses through algebraic constructions

In this section we briefly suggest how some well known results in the graph grammar theory can be applied to the representation of logic programs described above.

The representation of a resolution step through three pushouts in the category $\mathbf{Jungle}_{\Sigma,\Pi}$ allows one to exploit general results of category theory for proving the correctness of some operations on clauses. In the following examples we will show how new (not necessarely definite) clauses can be generated from the original clauses of a logic program. These clauses can be added to the program without changing its semantics, in the sense that the effect of their application to a goal can be obtained by one or more applications of the original clauses.

6.1 Specializing clauses

As a first example, it could be noticed that in the jungle representation of a resolution step (as described in the previous section) the lower part of the diagram, containing the arrows which represent the derivation of the goal G' from G via C and θ[5], i.e.,

$$\begin{tikzcd} G \arrow[r, "\theta"] & G\theta & D \arrow[l] \arrow[r] & G' \end{tikzcd}$$

has exactly the same shape of the upper part of the diagram, which contains the representation of clause C and the inclusion of the predicate symbol in the head of C, i.e.,

$$\begin{tikzcd} J(C) = L' \arrow[r] & L & K \arrow[l, "l"'] \arrow[r, "r"] & R \end{tikzcd}$$

The following theorem states that we can safely take the goal derivation as a new, original (in general non definite) clause of the program, and that whenever we can apply it to a goal G_1, we get the same result we would obtain by the application of the original clause C to G_1. First we need a well known result of category theory.

[5] Sometimes in the rest of the paper, we will improperly denote the jungle representation of a goal G by G itself, and the arrow inducing a substitution θ by θ itself.

6.1 Lemma *(property of pushouts)*

In the following diagram, if squares (1) and (2) are pushouts, then also the outermost square is a pushout [ML71].

$$\begin{tikzcd}
\bullet \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & \bullet \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & \bullet \arrow[d] \\
\bullet \arrow[r] & \bullet \arrow[r] & \bullet
\end{tikzcd}$$

♦

6.2 Theorem *(specializing clauses)*

If G' can be derived from G via C and θ (where $\theta = \theta_1 \cup \theta_2$, as in the diagram below), then

1) if $g_1: G \to G_1$ is a morphism, and $\langle \sigma_1, \sigma_2 \rangle$ is the pushout of $\langle g_1, \theta_1 \rangle$ (square 4 below), then the rewriting rule $(G'' \leftarrow H \to G')$ can be applied to G''_1, producing the graph G'_1, in the sense that the pushout complement 5 and the pushout 6 can always be constructed.
2) if the conditions of point 1) are satisfied, then G'_1 can be derived from G_1 via C and $\theta' = \sigma_1 \cup (\sigma_2 \circ \theta_2)$.

$$J(C) = \begin{tikzcd}
L' \arrow[r] \arrow[d, "g"'] \arrow[dr, phantom, "1"] & L \arrow[d, "\theta_2"] \arrow[dr, phantom, "2"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "3"] & R \arrow[d] \\
G \arrow[r, "\theta_1"] \arrow[d, "g_1"'] \arrow[dr, phantom, "4"] & G'' \arrow[d, "\sigma_2"] \arrow[dr, phantom, "5"] & H \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "6"] & G' \arrow[d] \\
G_1 \arrow[r, "\sigma_1"'] & G''_1 & H_1 \arrow[l] \arrow[r] & G'_1
\end{tikzcd}$$

Proof sketch. The result can be easily proved by applying Lemma 6.1 to the pairs of pushouts 1 and 4, 2 and 5, and 3 and 6.♦

The second line in the above diagram can be interpreted as a specialization of the original clause C. In general, it is applicable to a goal G_1 only if it is 'more instantiated' than goal G, otherwise a morphism g_1 cannot be found.

6.2 Unfolding clauses

The unfolding of clauses is another technique that can be used to enrich a logic program by improving its 'efficiency' and without changing its semantics. Let $C \equiv H :- A_1, \ldots, A_n$ and $C' \equiv H' :- B_1, \ldots, B_m$ be two clauses of a logic program P, and θ be an mgu of H' and an atomic goal of C, say A_i. Then the *unfolding of C and C' (via A_i, θ)* is a clause C" having as body the resolvent of $A_1, \ldots, A_n$ and C', and as head the head of C instantiated by θ, i.e.,

$$C'' \equiv H\theta :- (A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n)\theta$$

For example, consider the standard program for the *append* predicate:

append(nil, x, x),

append(cons(y, z), w, cons(y, v)) :- append(z, w, v).

The first clause applies when we have to concatenate two lists the first of which is empty. In that case the resulting list is just the second one. The second clause applies instead when the first list contains at least one element, represented by the variable y. The unfolding of the two clauses is

append(cons(y, nil), x, cons(y, x)).

which is a new clause (with empty body) applicable only when the first list has exactly one element.

A relevant fact of (positive) definite clauses is that they are closed w.r.t. unfolding, i.e., the unfolding of two clauses is again a definite clause. This is not true for many extensions of Horn Clause Logic, including the so called *concurrent* logic languages (cf. [Le88]).

We show now that the unfolding of definite clauses corresponds (in the jungle rewriting framework) to a specific version of the well known *concurrency theorem* for graph grammars [Eh83, EHKP90]. More precisely, we first describe how to construct a new jungle production from two given productions representing program clauses (cf. Definition 4.8), and then we show that the new production can be applied to a jungle iff the two given productions can be applied to it in the given order.

6.3 Proposition *(concatenating jungle productions)*

Let $J(C) \equiv L \leftarrow K \rightarrow R$ and $J(C') \equiv L' \leftarrow K' \rightarrow R'$ be the jungle productions representing two definite clauses C and C' respectively (cf. Definition 4.8), and let $P' = J(p(x_1, \ldots, x_n))$ be the jungle containing just one hyperedge colored with the predicate symbol of the head of C', connected to n distinct nodes. Moreover, let $k: P' \rightarrow R$ be a jungle morphism and $P' \rightarrow L'$ be the obvious mapping. If the pushout (3) of k and $P' \rightarrow L'$ and the pushout complement (2) of $K \rightarrow R \rightarrow W$ exist, then the following construction is uniquely determined (up to isomorphisms) and defines a new jungle production $L'' \leftarrow K'' \rightarrow R''$.

$$\begin{tikzcd}
 & & & & & P' \arrow[dl, "k"'] \arrow[dr] & & & & & \\
L \arrow[dd] & & K \arrow[ll] \arrow[rr] \arrow[ddr] & & R \arrow[ddr] & (3) & L' \arrow[ddl] & & K' \arrow[ll] \arrow[rr] \arrow[ddl] & & R' \arrow[dd] \\
 & (1)\ \mathrm{PO} & & (2)\ \mathrm{PO\text{-}C} & & \mathrm{PO} & & (4)\ \mathrm{PO\text{-}C} & & (5)\ \mathrm{PO} & \\
L'' & & & W' \arrow[lll] \arrow[rr] & & W & & W'' \arrow[ll] \arrow[rrr] & & & R'' \\
 & & & & & (6)\ \mathrm{PB} & & & & & \\
 & & & & & K'' \arrow[uull] \arrow[uurr] & & & & &
\end{tikzcd}$$

Proof sketch. By hypothesis squares (2) and (3) are pushouts, the pushout complement (4) exists and it is unique by Proposition 4.7, the pushouts (1) and (5) exist by Proposition 4.6, and since it can be shown that all pairs of arrows with common target have a pullback [ML71] in $\mathbf{Jungle}_{\Sigma,\Pi}$, square (6) can be constructed as a pullback. The statement follows by uniqueness of universal constructions. Finally, the new rule $L'' \leftarrow K'' \rightarrow R''$ is obtained by taking the composite arrows $L'' \leftarrow W' \leftarrow K''$ and $K'' \rightarrow W'' \rightarrow R''$.♦

The requirement of the existence of pushout (3) in the last proposition is clear: it means that the head of C' has to unify with the atom of the body of C selected by k. The second condition (existence of the pushout (2)) is not so evident from the logic programming point of view, and is discussed at the end of the section.

The next theorem states that the application of the new production L" ← K" → R" to a jungle is equivalent to the sequential application of J(C) and J(C').

6.4 Theorem *(concurrency theorem for jungles)*

Let J(C), J(C'), and L" ← K" → R" and be as in Proposition 6.3. Then

1) If the jungle production L" ← K" → R" can be applied to a jungle G yielding jungle H (in the sense of Theorem 5.3), then J(C) can be applied to G yielding G', and J(C') can be applied to G' yielding H' isomorphic to H.

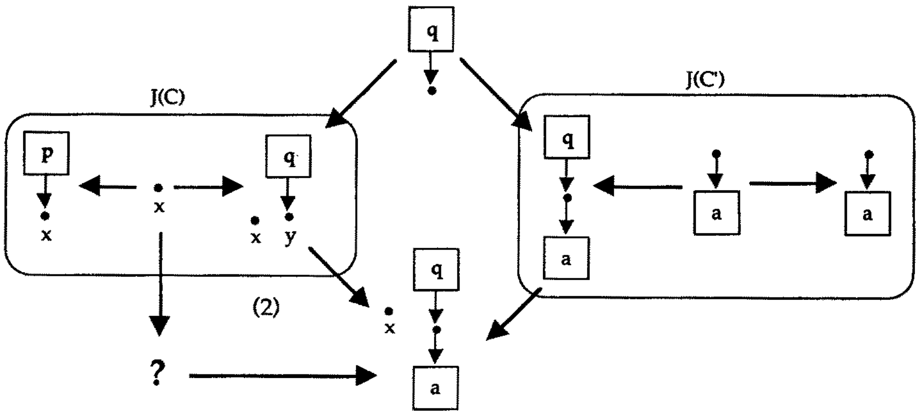
2) Viceversa, if J(C) can be applied to G yielding a jungle G', and J(C') can be applied to G' yielding H, and the predicate-colored edge of G' rewritten by J(C') is the image in G' of kP', then production L" ← K" → R" can be applied to G yielding a jungle H' isomorphic to H.

Proof sketch. The theorem is a (weaker) version (for the specific case of jungle rewriting) of the *concurrency theorem* that has been proved in literature for many rewriting systems based on pushout constructions. Recently a very general version of that theorem has been proved in [EHK90] for a class of High Level Replacement Systems, namely for the so-called HLR2-categories. These are categories equipped with a distinguished class of morphisms M, and satisfying various conditions of existence of limits and colimits, and of closure w.r.t. suitable constructions. It is cumbersome but not too difficult to check that $\mathbf{Jungle}_{\Sigma,\Pi}$ is indeed a HLR2-category, if M is chosen as the collection of all injective morphisms such that the associated substitution is a renaming of variables. Thus the concurrency theorem follows by the arguments in [EHK90]. ♦

Finally, we briefly comment the second hypothesis of Proposition 6.3, that is the requirement of the existence of the pushout complement (2). This condition can be interpreted as follows from the logic programming viewpoint: the mgu of the head of clause C' and the atom selected by k in the body of clause C cannot instantiate nor identify variables local to the body of G (i.e., that do not appear in its head). For example, the clauses

C ≡ p(x) :- q(y).
C' ≡ q(a).

do not satisfy this requirement, since the mgu of q(y) and q(a) instantiates y, which is local to the body of C. In this situation the construction of Proposition 6.3 fails, as shown in the diagram below: the pushout complement (2) does not exists because the dangling condition is not satisfied.



However, the unfolding of these two clauses is clearly possible, and produces the new fact

q(y).

Therefore there is an apparent incongruence between the logic programming technique of unfolding and its formalization through the construction of Proposition 6.3, since the first is always possible while the second is not. This problem can be solved by not requiring the existence of the pushout complement (2), but just the existence of a commutative square (2) such that the bottom arrow is in the class M. In this case the construction can always be performed. In the example above, the jungle that can be used instead of the pushout complement in (2) is contains just one variable node, mapped onto node x.

7 Conclusions

This paper can be considered as a preliminary work towards the investigation of the relationship between logic programming and graph grammars. We described how a logic program can be faithfully represented as a set of jungle rewriting rules, while a query can be represented by a jungle. Actually, programs and queries have many representations as jungles, all of them correct. The choice of a particular one concerns only the efficiency of its manipulation. A resolution step is modelled by two algebraic constructions: first a pushout, which models the unification step (instantiating the goal), then a double pushout which models the actual rewriting. The refutation of a goal is thus described as a sequence of such steps starting from the jungle representing the goal, and ending in a jungle with no predicates (representing the empty goal).

The algebraic framework that we use can be fruitfully exploited to define composition operations on clauses, and elegantly prove their correctness. Moreover, the rich collection of results about parallelism and concurrency in the graph-grammar theory could be exploited in the logic programming framework, in order to formally analyze and prove properties of parallel execution frameworks.

Acknowledgements

Many of the ideas presented in this paper originated by fruitful discussions with Hartmut Ehrig, Michael Löwe, and Ugo Montanari.

8 References

[AM89] Asperti, A., Martini, S., *Projections instead of variables, A category theoretic interpretation of logic programs*, Proc. 6th Int. Conf. on Logic Programming, Lisboa, Portugal, 1989.

[CER79] Claus, V., Ehrig, H., Rozenberg, G., (Eds.) *Proceedings of the 1stInternational Workshop on* Graph-Grammars and Their Application to Computer Science and Biology, LNCS 73, 1979.

[CMREL91] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Löwe, M., *Logic Programming and Graph Grammars*, to appear in [EKR91].

[Eh83] Ehrig, H., *Aspects of concurrency in graph grammars*, in [ENR83], pp. 58-81.

[Eh87] Ehrig, H., *Tutorial introduction to the algebraic approach of graph-grammars*, in [ENRR87], pp. 3-14.

[EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, F. Parisi-Presicce, *Parallelism and Concurrency in High-Level Replacement Systems*, Technical Report, Technische Universität Berlin, September 1990.

[EKR91] Ehrig, H., Kreowski, H.-J., Rozenberg, G., (Eds.) *Proceedings of the 4th International Workshop on* Graph-Grammars and Their Application to Computer Science, LNCS, 1991, to appear.

[ENR83] Ehrig, H., Nagl, M., Rozenberg, G., (Eds.) *Proceedings of the 2nd International Workshop on* Graph-Grammars and Their Application to Computer Science, LNCS 153, 1983.

[ENRR87] Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A., (Eds.) *Proceedings of the 3rd International Workshop on* Graph-Grammars and Their Application to Computer Science, LNCS 291, 1987.

[Go88] Goguen, J.A., *What is Unification? A Categorical View of Substitution, Equation and Solution*, SRI Research Report SRI-CSL-88-2R2, SRI International, Menlo Park, California, 1988.

[Ha89] Habel, A., *Hyperedge Replacement: Grammars and Languages*, Ph.D. Thesis, University of Bremen, 1989.

[HK87] Habel, A., Kreowski, H.-J., *May we introduce to you: hyperedge replacement*, in [ENRR87], pp. 15-26.

[HKP88] Habel, A., Kreowski, H-J., Plump, D., *Jungle evaluation*, in Proc. Fifth Workshop od Specification of Abstract Data Types, LNCS 332, 1988, pp. 92-112.

[Ke91] J.R. Kennaway, *Graph rewriting in some categories of partial morphisms*, in [EKR91].

[Kr87] Kreowski, H.-J., *Is parallelism already concurrency? Part 1: Derivations in graph grammars*, in [ENRR87], pp. 343-360.

[Le88] G. Levi, *Models, Unfolding Rules and Fixpoint Semantics*, in Proc. 5th Int. Conf. Symp. on Logic Programming, Seattle, MIT Press, pp. 1649-1665, 1988.

[Ll87] Lloyd, J.W., *Foundations of Logic Programming*, Springer Verlag, 1984, (Second Edition 1987).

[ML71] Mac Lane, S., *Categories for the Working Mathematician*, Springer Verlag, New York, 1971.

[PEM87] Parisi-Presicce, F., Ehrig, H., Montanari, U., *Graph Rewriting with Unification and Composition*, in [ENRR87], p. 496-514.

[RB85] Rydeheard, D.E., Burstall, R.M., *The Unification of Terms: A Category-Theoretic Algorithm*, Internal Report UMCS-85-8-1, Dept. Comp. Sci., University of Manchester, August 1985.

[RM88] Rossi, F., Montanari, U., *Hypergraph Grammars ad Networks of Constraints versus Logic Programming and Metaprogramming*, in Proc. META88, MIT Press, Bristol, June 1988.

[RM90] Rossi, F., Montanari, U., *Constraint Relaxation as Higher Order Logic Programming*, to appear in Proc. META90, Leuven, April 1990. Also Proc. GULP '88, in italian.