A Heterogeneous Pushout Approach to Term-Graph Transformation*

Dominique Duval[1], Rachid Echahed[2], and Frédéric Prost[2]

[1] LJK, Université de Grenoble, France
Dominique.Duval@imag.fr
[2] LIG, CNRS, Université de Grenoble
B.P.53, F-38041 Grenoble, France
Rachid.Echahed@imag.fr, Frederic.Prost@imag.fr

Abstract. We address the problem of cyclic termgraph rewriting. We propose a new framework where rewrite rules are tuples of the form (L, R, τ, σ) such that L and R are termgraphs representing the left-hand and the right-hand sides of the rule, τ is a mapping from the nodes of L to those of R and σ is a partial function from nodes of R to nodes of L. The mapping τ describes how incident edges of the nodes in L are connected in R, it is not required to be a graph morphism as in classical algebraic approaches of graph transformation. The role of σ is to indicate the parts of L to be cloned (copied). Furthermore, we introduce a notion of *heterogeneous pushout* and define rewrite steps as heterogeneous pushouts in a given category. Among the features of the proposed rewrite systems, we quote the ability to perform local and global redirection of pointers, addition and deletion of nodes as well as cloning and collapsing substructures.

1 Introduction

Complex data-structures built by means of records and pointers, can formally be represented by *termgraphs* [2,15,18]. Roughly speaking, a termgraph is a first-order term with possible sharing and cycles. The unravelling of a termgraph is a rational term. Termgraph rewrite systems constitute a high-level framework which allows one to describe, at a very abstract level, algorithms over data-structures with pointers. Thus avoiding, on the one hand, the cumbersome encodings which are needed to translate graphs (data-structures) into trees in the case of programming with first-order term rewrite systems and, on the other hand, the many classical errors which may occur in imperative languages when programming with pointers.

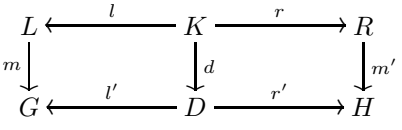
Transforming a termgraph is not an easy task in general. Many different approaches have been proposed in the literature which tackle the problem of termgraph transformation. The algorithmic approach such as [2,8] defines in detail every step involved in the transformation of a termgraph by providing the

* This work has been partly funded by the project ARROWS of the French *Agence Nationale de la Recherche.*

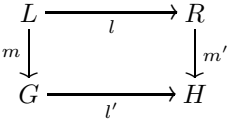
R. Treinen (Ed.): RTA 2009, LNCS 5595, pp. 194–208, 2009.

corresponding algorithm; for our purpose, this approach is too close to implementation techniques. In [1], equational definition of termgraphs is exploited to define termgraph transformation. These transformations are obtained up to bisimilar structures (two termgraphs are bisimilar if they represent the same rational term). Unfortunately, bisimilarity is not a congruence in general, e.g., the lengths of two bisimilar but different circular lists are not bisimilar.

A more abstract approach to graph transformation is the algebraic one, first proposed in the seminal paper [10]. It defines a rewrite step using the notion of pushouts. The algebraic approach is quite declarative. The details of graph transformations are hidden thanks to pushout constructs. There are mainly two different algebraic approaches, namely the double pushout (DPO) and the single pushout (SPO) approaches, which can be illustrated as follows:



Double pushout: a rewrite step



Single pushout: a rewrite step

In the DPO approach [10,5], a rule is defined as a span, i.e., as a pair of graph morphisms $L \leftarrow K \rightarrow R$. A graph G rewrites into a graph H if and only if there exists a morphism (a matching) $m : L \rightarrow G$, a graph D and graph morphisms d, m', l', r' such that the left and the right squares in the diagram above for a DPO step are pushouts. In general, D is not unique, and sufficient conditions may be given in order to ensure its existence, such as dangling and identification conditions. Since graph morphisms are completely defined, the DPO approach is easy to grasp, but in general this approach fails to specify rules with deletion of nodes, as witnessed by the following example. Let us consider the reduction of the term $f(a)$ by means of the rule $f(x) \rightarrow f(b)$. This rule can be translated into a span $f(x) \leftarrow K \rightarrow f(b)$ for some graph K. When applied to $f(a)$, because of the pushout properties, the constant a must appear in D, hence in H, although $f(b)$ is the only desired result for H, in the context of term rewriting.

In the SPO approach [17,12,13,9], a rule is a *partial* graph morphism $L \rightarrow R$. When a (total) graph morphism $m : L \rightarrow G$ exists, G rewrites to H if and only if the square in the diagram above for a SPO step is a pushout. This approach is appropriate to specify deletion of nodes thanks to partial morphisms. However, in the case of termgraphs, some care should be taken when a node is deleted. Indeed, deletion of a node causes automatically the deletion of its incident edges. This is not sound in the case of termgraphs since each function symbol should have as many successors as its arity.

In this paper, we investigate a new approach to the definition of rewrite steps for cyclic termgraphs. We are interested in rewrite steps such that H is obtained from G by performing one of the six following kinds of actions: (i) addition of new nodes, (ii) redirection of particular edges, (iii) redirection of all incoming edges of a particular node, (iv) deletion of nodes, (v) cloning of nodes, and (vi) collapsing of nodes. In order to deal with these features in a single framework,

we propose a new algebraic approach to define such rewrite steps. Our approach departs from the SPO and the DPO approaches. A rewrite rule is defined as a tuple (L, R, τ, σ) such that L and R are termgraphs, respectively the left-hand side and the right-hand side of the rule, τ is a mapping from the nodes of L into the nodes of R (τ needs not be a graph morphism) and σ is a partial function from the nodes of R into the nodes of L. Roughly speaking, $\tau(p) = n$ indicates that incoming edges of p are to be redirected towards n and $\sigma(n) = p$ indicates that node n should be instantiated as p (parameter passing). We show that whenever a matching $m : L \to G$ exists, then the termgraph G rewrites into a termgraph H. We define the termgraph H as an initial object of a given category, generalizing the definition of a pushout. We call it a *heterogeneous pushout*.

The paper is organized as follows. In section 2 we introduce the basic definitions of graphs and morphisms considered in the paper. In section 3, we give the definition of rewriting, and we illustrate our approach through several examples in section 4. A comparison with related work is done in section 5, and concluding remarks are given in section 6.

2 Graphs

In this section are given some basic definitions. We assume the reader is familiar with category theory. The missing definitions may be consulted in [14].

Throughout this paper, a signature Ω is fixed. Each operation symbol $\omega \in \Omega$ is endowed with an *arity* $\mathrm{ar}(\omega) \in \mathbb{N}$. For each set X, the set of strings over X is denoted X^*, and for each function $f : X \to Y$, the function $f^* : X^* \to Y^*$ is defined by $f^*(x_1 \dots x_n) = f(x_1) \dots f(x_n)$. In addition, we denote by **Set** the category of sets.

Definition 1 (Graph). *A* termgraph*, or simply a* graph $G = (\mathcal{N}, \mathcal{D}, \mathcal{L}, \mathcal{S})$ *is made of a set of* nodes $\mathcal{N}$ *and a subset of* labeled nodes $\mathcal{D} \subseteq \mathcal{N}$*, which is the domain for a* labeling function $\mathcal{L} : \mathcal{D} \to \Omega$ *and a* successor function $\mathcal{S} : \mathcal{D} \to \mathcal{N}^*$*, such that for each labeled node n, the length of the string $\mathcal{S}(n)$ is the arity of the operation $\mathcal{L}(n)$. For each labeled node n the fact that $\omega = \mathcal{L}(n)$ is written $n{:}\omega$, and each unlabeled node n may be written as $n{:}\bullet$, so that the symbol $\bullet$ is a kind of anonymous variable. A* graph homomorphism*, or simply a* graph morphism $g : G \to H$*, where $G = (\mathcal{N}_G, \mathcal{D}_G, \mathcal{L}_G, \mathcal{S}_G)$ and $H = (\mathcal{N}_H, \mathcal{D}_H, \mathcal{L}_H, \mathcal{S}_H)$ are graphs, is a function $g : \mathcal{N}_G \to \mathcal{N}_H$ which preserves the labeled nodes and the labeling and successor functions. This means that $g(\mathcal{D}_G) \subseteq \mathcal{D}_H$, and for each labeled node n, $\mathcal{L}_H(g(n)) = \mathcal{L}_G(n)$ and $\mathcal{S}_H(g(n)) = g^*(\mathcal{S}_G(n))$ (the image of an unlabeled node may be any node). This yields the category* **Gr** *of graphs.*

Definition 2 (Node functor). *The* node functor $|-| : \mathbf{Gr} \to \mathbf{Set}$ *maps each graph $G = (\mathcal{N}, \mathcal{D}, \mathcal{L}, \mathcal{S})$ to its set of nodes $|G| = \mathcal{N}$ and each graph morphism $g : G \to H$ to its underlying function $|g| : |G| \to |H|$. In this paper, it is also called the* underlying *functor.*

We may denote g instead of $|g|$ since the node functor is faithful, which means that a graph morphism is determined by its underlying function on nodes. The

faithfulness of the node functor implies that a diagram of graphs is commutative if and only if its underlying diagram of sets is commutative.

The *graphic functions* and the *strictly graphic functions*, as defined now, can be seen as "weak" graph morphisms. They will be used in section 3 to relate graphs involved in a rewrite step.

Definition 3 (Graphic functions). *Let G and H be graphs and $\gamma : |G| \to |H|$ a function. For each node n of G, γ is* graphic at n *if either n is unlabeled or both n and $\gamma(n)$ are labeled, $\mathcal{L}_H(\gamma(n)) = \mathcal{L}_G(n)$ and $\mathcal{S}_H(\gamma(n)) = \gamma^*(\mathcal{S}_G(n))$. And γ is* strictly graphic at n *if either both n and $\gamma(n)$ are unlabeled or both n and $\gamma(n)$ are labeled, $\mathcal{L}_H(\gamma(n)) = \mathcal{L}_G(n)$ and $\mathcal{S}_H(\gamma(n)) = \gamma^*(\mathcal{S}_G(n))$. For each set of nodes Γ of G, γ is* graphic (resp. strictly graphic) on Γ *if γ is graphic (resp. strictly graphic) at every node in Γ.*

Example 1. Let us consider the following graphs G_1 and G_2:

Let $\gamma : |G_1| \to |G_2|$ be the function defined by $\gamma = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto d\}$. It is easy to check that γ is strictly graphic on $\{1, 3\}$, is graphic but not strictly graphic on $\{1, 2, 3\}$, and is not graphic on $\{1, 2, 3, 4\}$.

It should be noted that the property of being graphic (resp. strictly graphic) on Γ involves the successors of the nodes in Γ, which may be outside Γ. In addition, it is clear that a function $\gamma : |G| \to |H|$ underlies a graph morphism $g : G \to H$ if and only if it is graphic on $|G|$. The next straightforward result will be useful.

Lemma 1. *Let G, H, H' be graphs and let $\gamma : |G| \to |H|$, $\gamma' : |G| \to |H'|$, $\eta : |H| \to |H'|$ be functions such that $\gamma' = \eta \circ \gamma$. Let Γ be a set of nodes of G. If γ is strictly graphic on Γ and γ' is graphic on Γ, then η is graphic on $\gamma(\Gamma)$.*

3 Rewriting

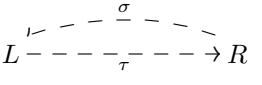
Roughly speaking, in the context of graph rewriting, a rewrite rule has a left-hand side graph L and a right-hand side graph R, and a rewrite step applied to a graph G with an occurrence of L returns a graph H with an occurrence of R, by replacing L by R in G. The meaning of this "replacement" is quite clear for the nodes, as well as for the edges of G that connect two nodes outside L. When a labeled node p in G outside L has its i-th successor p' inside L, then p must have some i-th successor n' in H. For this purpose, we introduce a function τ (τ for "target") from the nodes of L to the nodes of R, and we decide that n' must be $\tau(p')$. On the other hand, when a labeled node p in G inside L has an i-th successor p', then we may require that some node n' in H has the same label as p and has as its i-th successor either p', if it is outside L, or $\tau(p')$ otherwise; then

n' is called a τ-clone of p. Since each node in L may have an arbitrary number of clones (maybe no clone at all), and a node in R cannot be a clone of more than one node in L, this is specified thanks to a partial function σ (σ for "source") from the nodes of R to the nodes of L, which maps the clones of p to p. Partial functions are denoted with the symbol "$\rightharpoonup$", the domain of a partial function σ is denoted $\mathrm{Dom}(\sigma)$, and the composition of partial functions is defined as usual. The main result is theorem 1: under relevant definitions and assumptions, for each rewrite rule T and matching m there is a *heterogeneous pushout* of T and m, which can be built explicitly from a pushout of sets.

Definition 4 (Clones). *Let G and H be graphs and $\tau : |G| \to |H|$ a function. Then $p \in |H|$ is a* τ-clone *of $q \in |G|$ when: p is labeled if and only if q is labeled, and then $\mathcal{L}_H(p) = \mathcal{L}_G(q)$ and $\mathcal{S}_H(p) = \tau^*(\mathcal{S}_G(q))$. It is* not *required that $p = \tau(q)$.*

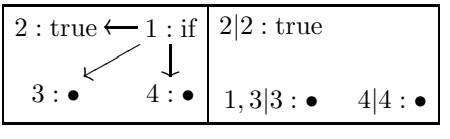
Definition 5 (Rewrite rule). *A* rewrite rule *is a tuple (L, R, τ, σ) made of two graphs L and R, a function $\tau : |L| \to |R|$ and a partial function $\sigma : |R| \rightharpoonup |L|$ such that each node n in the domain of σ is unlabeled or is a τ-clone of $\sigma(n)$. A* morphism of rewrite rules, *from $T = (L, R, \tau, \sigma)$ to $T_1 = (L_1, R_1, \tau_1, \sigma_1)$ is a pair of graph morphisms (m, d) with $m : L \to L_1$ and $d : R \to R_1$ such that $|d| \circ \tau = \tau_1 \circ |m|$, $d(\mathrm{Dom}(\sigma)) \subseteq \mathrm{Dom}(\sigma_1)$ and $|m| \circ \sigma = \sigma_1 \circ |d|$ on $\mathrm{Dom}(\sigma)$.*

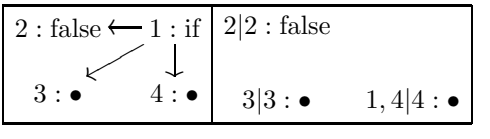
In this paper, the illustrations take place either in the category **Set** of sets or in a heterogeneous framework where the points stand for graphs, the solid arrows for graph morphisms and the dashed arrows for functions on nodes. So, a rewrite rule $T = (L, R, \tau, \sigma)$ will be illustrated as:



In order to ease the reading of the examples, a rule is depicted as $\boxed{L|R}$. In addition, each node n in R in the image of τ is named $n = x, y, \ldots|w$ where $x, y, \ldots$ are the names of the nodes in L such that $\tau(x) = \tau(y) = \ldots = n$ and where $\sigma(n) = w$. Whenever n is not in the image of τ then it is named $n = x|w$ where the name x is new (i.e., x does not appear in L) and where $\sigma(n) = w$. In both cases, the "$|w$" part is omitted when n is not in the domain of σ.

Example 2 (if-then-else). The following rewrite rules define the "if-then-else" operator as it behaves in classical imperative languages. We assume that the three arguments of an "if-then-else" expression may be shared.



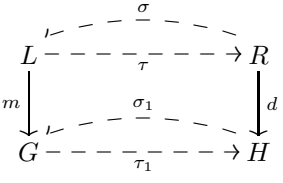


The definition of τ ensures that the "if-then-else" expression is replaced by its value $\tau(3) = 1, 3|3$ (resp. $\tau(4) = 1, 4|4$). The definition of σ indicates that

the value of the "if-then-else" is its second (resp. third) argument specified by $\sigma(1,3|3) = 3$ (resp. $\sigma(1,4|4) = 4$). Notice that if σ were defined as the empty function, the "if-then-else" expression would evaluate to an unlabeled node. In addition, in both rules, we have $\sigma(2|2) = 2$ which ensures, in case node 2 is shared, that its incident edges remain unchanged after a rewrite step. Finally, the reader may verify that these two rules are sufficient to handle all possible cases, even when the arguments of an "if-then-else" expression are shared, thanks to the conditions on matching substitutions given later in definition 8.

It can be noted that each graph morphism $t : L \to R$ determines a rewrite rule where $\tau = |t|$ and σ is defined nowhere. In this case, for each graph morphism $m : L \to G$ the pushout of t and m in the category **Gr**, when it exists, is the initial object in the category of cones over t and m. Let us adapt this definition to any rewrite rule $T = (L, R, \tau, \sigma)$ and any graph morphism $m : L \to G$.

Definition 6 (Heterogeneous cone). *Let $T = (L, R, \tau, \sigma)$ be a rewrite rule and $m : L \to G$ a graph morphism. A* heterogeneous cone over T and m *is a tuple (H, τ_1, d, σ_1) made of a graph H, a function $\tau_1 : |G| \to |H|$, a graph morphism $d : R \to H$ and a partial function $\sigma_1 : |H| \rightharpoonup |G|$ such that $T_1 = (G, H, \tau_1, \sigma_1)$ is a rewrite rule, $(m, d) : T \to T_1$ is a morphism of rewrite rules, τ_1 is graphic on $|G| - |m(L)|$ and n_1 is a τ_1-clone of $\sigma_1(n_1)$ for each n_1 in the domain of σ_1.*



A morphism of heterogeneous cones over T and m, *say $h : (H, \tau_1, d, \sigma_1) \to (H', \tau'_1, d', \sigma'_1)$, is a graph morphism $h : H \to H'$ such that $|h| \circ \tau_1 = \tau'_1$, $h \circ d = d'$, $h(\mathrm{Dom}(\sigma_1)) \subseteq \mathrm{Dom}(\sigma'_1)$ and $\sigma'_1 \circ |h| = \sigma_1$ on $\mathrm{Dom}(\sigma_1)$.*

This yields the category $\mathbf{C}_{T,m}$ of heterogeneous cones over T and m.

Definition 7 (Heterogeneous pushout). *Let $T = (L, R, \tau, \sigma)$ be a rewrite rule and $m : L \to G$ a graph morphism. A* heterogeneous pushout of T and m *is an initial object in the category $\mathbf{C}_{T,m}$ of heterogeneous cones over T and m.*

When a heterogeneous pushout exists, its initiality implies that it is unique up to an isomorphism of heterogeneous cones. In theorem 1 we prove the existence of a heterogeneous pushout of T and m under some injectivity assumption on m.

Definition 8 (Matching). *A* matching *with respect to a rewrite rule $T = (L, R, \tau, \sigma)$ is a graph morphism $m : L \to G$ such that if $m(p) = m(p')$ for distinct nodes p and p' in L then $\tau(p)$ and $\tau(p')$ are in $\mathrm{Dom}(\sigma)$ and $m(\sigma(\tau(p))) = m(\sigma(\tau(p')))$ in G.*

Proposition 1. *Let $T = (L, R, \tau, \sigma)$ be a rewrite rule and $m : L \to G$ a matching with respect to T. Then the pushout of τ and $|m|$ in* **Set***:*

$$\begin{array}{ccc} |L| & \xrightarrow{\tau} & |R| \\ {\scriptstyle |m|}\downarrow & & \downarrow{\scriptstyle \delta} \\ |G| & \xrightarrow[\tau_1]{} & \mathcal{H} \end{array}$$

satisfies $\mathcal{H} = \tau_1(\Gamma) + \delta(\Delta) + \delta(\Sigma)$ where $\Gamma = |G| - |m(L)|$, $\Sigma = \mathrm{Dom}(\sigma)$, $\Delta = |R| - \Sigma$ and: the restriction of $\tau_1 : \Gamma \to \tau_1(\Gamma)$ is bijective, the restriction of $\delta : \Delta \to \delta(\Delta)$ is bijective, and the restriction of $\delta : \Sigma \to \delta(\Sigma)$ is such that if $\delta(n) = \delta(n')$ for distinct nodes n and n' in Σ then $m(\sigma(n)) = m(\sigma(n'))$ in G. In addition, there is a unique partial function $\sigma_1 : \mathcal{H} \rightharpoonup |G|$ with domain $\delta(\Sigma)$ such that $|m| \circ \sigma = \sigma_1 \circ \delta$.

Proof. Clearly $\mathcal{H} = \tau_1(\Gamma) + \delta(|R|)$ and the restriction of $\tau_1 : \Gamma \to \tau_1(\Gamma)$ is bijective. If $\delta(n) = \delta(n')$ for distinct nodes n and n' in R, then there is a chain from n to n' made of pieces like this one:

$$\widetilde{n} \xleftarrow{\ \tau\ } p \xmapsto{\ |m|\ } p_1 \xleftarrow{\ |m|\ } p' \xmapsto{\ \tau\ } \widetilde{n}'$$

with $\widetilde{n}, \widetilde{n}' \in |R|$, $p, p' \in |L|$, $p_1 \in |G|$, and it can be assumed that $\widetilde{n} \neq \widetilde{n}'$ and $p \neq p'$. Since m is a matching, $\widetilde{n}$ and $\widetilde{n}'$ are in Σ and $m(\sigma(\widetilde{n})) = m(\sigma(\widetilde{n}'))$. The decomposition of $\mathcal{H}$ follows. Now, let $n_1 \in \delta(\Sigma)$ and let us choose some $n \in \Sigma$ such that $n_1 = \delta(n)$. If σ_1 exists, then $\sigma_1(n_1) = \sigma_1(\delta(n)) = m(\sigma(n))$. On the other hand, if $n' \in \Sigma$ is another node such that $n_1 = \delta(n)$, then we have just proved that $m(\sigma(n)) = m(\sigma(n'))$, so that $m(\sigma(n))$ does not depend on the choice of n, it depends only on n_1. So, there is a unique $\sigma_1 : \mathcal{H} \rightharpoonup |G|$ as required, it is defined by $\sigma_1(n_1) = m(\sigma(n))$ for any $n \in \Sigma$ such that $n_1 = \delta(n)$.

Proposition 2. *Let $m : L \to G$ be a matching with respect to a rewrite rule $T = (L, R, \tau, \sigma)$. The pushout of τ and $|m|$ in* **Set***, with σ_1 as in proposition 1, underlies a heterogeneous cone over T and m.*

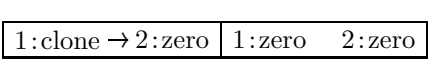
Proof. First, let us define a graph H with set of nodes $\mathcal{H}$. By exploiting proposition 1, and with the same notations, a graph H with set of nodes $\mathcal{H}$ is defined by imposing that τ_1 is strictly graphic on Γ, that δ is strictly graphic on Δ, and that each node $n_1 \in \delta(\Sigma)$ is a τ_1-clone of q_1, where $q_1 = \sigma_1(n_1)$. Now, let us prove that δ underlies a graph morphism $d : R \to H$. Since δ is graphic on Δ, we have to prove that δ is also graphic on Σ. Let $n \in \Sigma$ and $n_1 = \delta(n)$. If n is unlabeled there is nothing to prove, otherwise let $q = \sigma(n)$, then q is labeled, $\mathcal{L}_R(n) = \mathcal{L}_L(q)$ and $\mathcal{S}_R(n) = \tau^*(\mathcal{S}_L(q))$. Then $m(q) = m(\sigma(n)) = \sigma_1(\delta(n)) = q_1$, and from the fact that m is a graph morphism we get $\mathcal{L}_L(q) = \mathcal{L}_G(q_1)$ and $|m|^*(\mathcal{S}_L(q)) = \mathcal{S}_G(q_1)$. The definition of H imposes $\mathcal{L}_G(q_1) = \mathcal{L}_H(n_1)$ and $\tau_1^*(\mathcal{S}_G(q_1)) = \mathcal{S}_H(n_1)$. Altogether, $\mathcal{L}_R(n) = \mathcal{L}_H(n_1)$ and $\mathcal{S}_H(n_1) = (\tau_1^*(|m|^*(\mathcal{S}_G(q))) = \delta^*(\tau^*(\mathcal{S}_G(q))) = \delta^*(\mathcal{S}_R(n))$, so that indeed δ is also graphic on Σ. Finally, it is easy to check that this yields a heterogeneous cone over T and m.

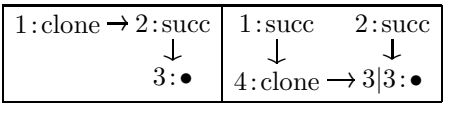
Theorem 1. *Given a rewrite rule $T = (L, R, \tau, \sigma)$ and a matching $m : L \to G$ with respect to T, the heterogeneous cone $(m, d) : T \to T_1$ over T and m defined in proposition 2 is a heterogeneous pushout of T and m.*

Proof. As in proposition 2, let $T_1 = (G, H, \tau_1, \sigma_1)$. Let us consider any heterogeneous cone $(m, d') : T \to T'_1$ over T and m, with $T'_1 = (G', H', \tau'_1, \sigma'_1)$. Since (m, d) underlies a pushout of sets, there is a unique function $\eta : |H| \to |H'|$ such that $\eta \circ |d| = |d'|$ and $\eta \circ \tau_1 = \tau'_1$. Let $\Sigma = \mathrm{Dom}(\sigma)$ and $\Sigma_1 = \mathrm{Dom}(\sigma_1)$. Because the node functor is faithful, the result will follow if we can prove that $\eta(\Sigma_1) \subseteq \Sigma'_1$ and $\sigma'_1 \circ \eta = \sigma_1$ on Σ_1, and that η underlies a graph morphism. We have $\eta(\Sigma_1) = \eta(d(\Sigma)) = d'(\Sigma) \subseteq \Sigma'_1$, and for each $n_1 \in \Sigma_1$, let $n \in \Sigma$ such that $n_1 = d(n)$, then on one hand $\sigma'_1(\eta(n_1)) = \sigma'_1(\eta(d(n))) = \sigma'_1(d'(n)) = m(\sigma(n))$ and on the other hand $\sigma_1(n_1) = \sigma_1(d(n)) = m(\sigma(n))$, hence as required $\sigma'_1(\eta(n_1)) = \sigma_1(n_1)$. In order to check that η underlies a graph morphism $h : H \to H'$, we use the decomposition of $\mathcal{H}$ from proposition 1 and the construction of the heterogeneous cone (m, d) in proposition 2. It follows immediately from lemma 1 that η is graphic on $\tau_1(\Gamma)$ and also on $d(\Delta)$. Let us prove that η is graphic on Σ_1. Let $n_1 \in \Sigma_1$, $q_1 = \sigma_1(n_1)$ and $n'_1 = \eta(n_1)$. Then $q_1 = \sigma'_1(n'_1)$ because $\sigma'_1 \circ \eta = \sigma_1$. So, n_1 is a τ_1-clone of q_1 and n'_1 is a τ'_1-clone of the same node q_1. This means that $\mathcal{L}_{H'}(n'_1) = \mathcal{L}_G(q_1) = \mathcal{L}_H(n_1)$ and that $\mathcal{S}_{H'}(n'_1) = (\tau'_1)^*(\mathcal{S}_G(q_1)) = \eta^*(\tau_1^*(\mathcal{S}_G(q_1))) = \eta^*(n_1)$. So, η is graphic on Σ_1, and since $d(\Sigma) \subseteq \Sigma_1$, it follows that η is graphic on $d(\Sigma)$. Altogether, η is graphic on the whole of $|H|$, which means that $\eta = |h|$ for a graph morphism $h : H \to H'$. This concludes the proof.

Definition 9 (Rewrite step). *Given a rewrite rule $T = (L, R, \tau, \sigma)$ and a matching $m : L \to G$ with respect to T, the corresponding* rewrite step *builds the graph morphism $d : R \to H$, obtained from the heterogeneous pushout of T and m.*

Example 3 (Cloning data-structures). Here are two rules for cloning natural numbers, encoded with succ and zero. These rules can be generalized to any data-structure as presented in section 5.



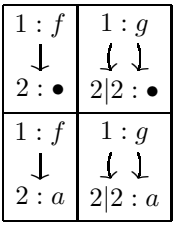


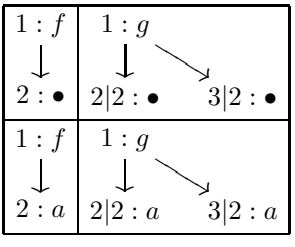
The first rule takes care of the cloning of *zero*. Given a matching $m : L \to G$, since $\tau(1) = 1$ this rule redirects all edges in G with target $m(1 : \mathrm{clone})$ to edges in H with target $d(1 : \mathrm{zero})$, and since $\tau(2) = 2$ the edges in G with target $m(2 : \mathrm{zero})$ remain "unchanged" in H, in the sense that their target is $d(2 : \mathrm{zero})$. The second rule takes care of the cloning of the non-zero naturals. Since $\sigma(3|3) = 3$, the label and successors of the node $m(3)$ in G will be "the same as" the label and successors of the node $d(3|3)$ in H. Notice that, in this case, it would not be allowed to define $\sigma(4) = 2$ because node 4 in R is labeled by clone and node 2 in L is labeled by succ, thus breaking the τ-clone condition.

Let us represent a rewrite step from G to H performed by a rewrite rule (L, R, τ, σ) as in the figure opposite, with the same notations as above regarding node names in the right-hand side of the rule. When the matching, m, is injective on nodes we denote $m(p) = p$.

L	R
G	H

Example 4. Let us consider the "rule" $f(x) \to g(x, x)$. In our framework, this rule may be translated to several different rules according to the way $g(x, x)$ is represented as a termgraph and to the way the functions τ and σ are defined. For instance, here are two different rules and their application to the termgraph $1 : f(2 : a)$, with the matching preserving the names of the nodes. The second rule provides two clones $2|2 : a$ and $3|2 : a$ in H to the node $2 : a$ in G.





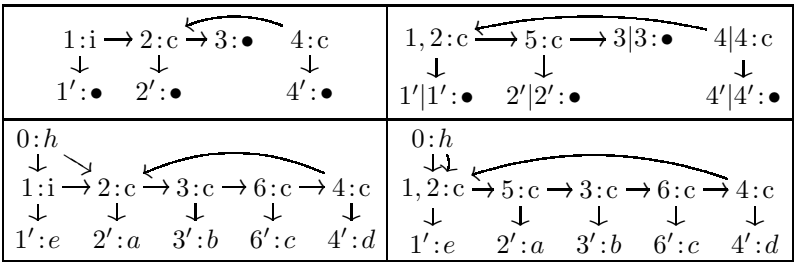
Since the rewriting of termgraphs with heterogeneous pushouts relies on a pushout on the underlying nodes (proposition 1), and also because of the condition on matching (definition 8), there is an obvious relation between the size of the rewritten termgraph and the size of the original termgraph, for each rewrite step. Moreover, this relation can be inferred at the level of rules. So, one can analyze memory usage of a program simply by inspecting its rules. Proposition 3, where $\sharp$ denotes the cardinal, states this formally. Therefore, if the size of a termgraph is considered as the measure of the memory used (thus putting aside unreachability issues), then it is possible to statically compute, for each rule separately, the amount of memory needed, or freed, by a rewriting step.

Proposition 3. *Let $T = (L, R, \tau, \sigma)$ be a rewrite rule, $m : L \to G$ a matching and $d : R \to H$ the result of the rewrite step. Then $\sharp|H| - \sharp|G| = \sharp|R| - \sharp|L|$.*

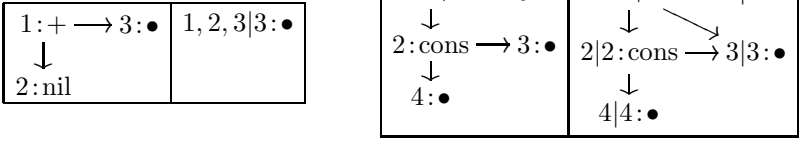
4 Examples

In this section, we provide several examples illustrating our framework. For better readability, when rewriting a termgraph G into H, the description of σ_1 in the graph H will now be omitted.

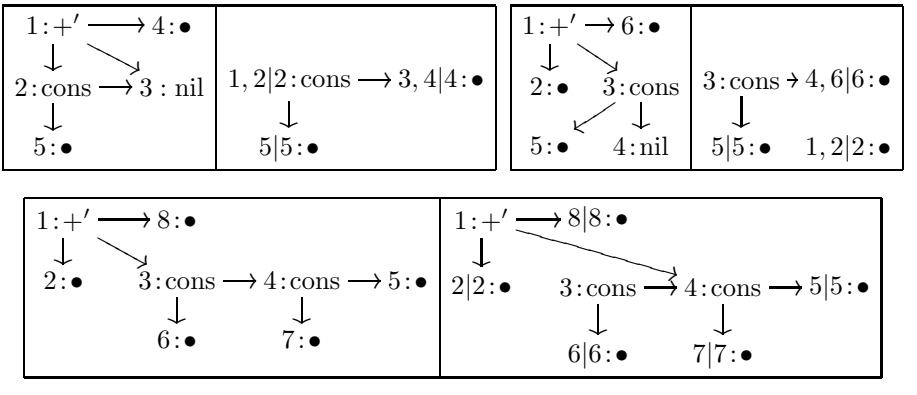
Example 5 (Insertion in a circular list). Here is a rule for the insertion of an element at the head of a circular list of size greater than one. In the left-hand side of this rule, node 2 is the head of the list, and node 4 is the last element of the list. The pointer from node 4 to the head of the list is moved from 2 (in L) to a new node $1, 2$ (in R). The definition of τ is such that all pointers to the head of the list are moved from 2 to $1, 2$. We apply the rule on a circular list of four items. We note c and i for cons and insert, respectively.



Example 6 (Appending linked lists). We now consider two rules for the operation "+" which appends, in place, two linked lists. The lists are built with the constructors cons and nil.



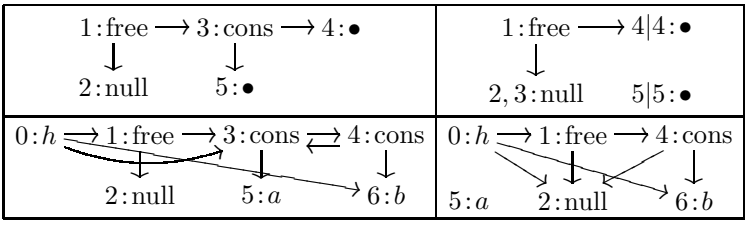
The first rule above takes care of the base case, when the first argument is nil. The second rule says that when the first argument of + is a non-empty list, then an auxiliary function "+′" of arity 3 is called. The role of this function is to go through the first list until its end and to concatenate the two lists by pointer redirection. The following three rules define the operation +′.



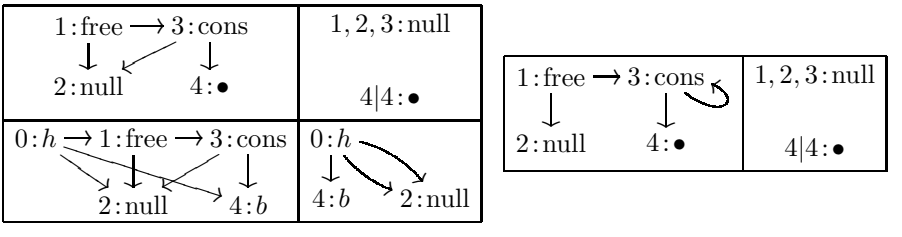
The first rule considers the case when the first list consists of one element. The second rule defines the case when the last element of the first list is reached. In this case, the edge $3 \to 4$ in L is redirected as $3 \to 4, 6|6$ in R, which is the head of the second list to append. The overall result of the operation +′ is the node $\tau(1) = 1, 2|2$, which is the head of first list. The third rule performs the traversal of the first list.

Example 7 (Memory freeing). In this example we show how we can free the memory used by the "cons" nodes of a circular list. As we are concerned with

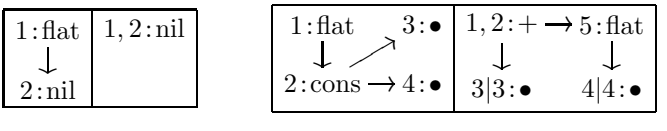
termgraphs where every function symbol has a fixed arity, it is not allowed to create dangling pointers nor to remove useless pointers. This constraint is expressed by the fact that every node in a left-hand side L must have an image in the right-hand side R by τ. The operation free has two arguments. The first one is a particular node labeled by a constant null, it is dedicated to be the target of the edges which were pointing to the freed nodes. The second argument of free is the list of cells to be freed. Here is a rule for defining the operation free in the case of a list with at least two elements. We illustrate its application on a list of length two. Notice that pointers incoming to nodes 3 and 5 are redirected towards node 2 in H.



For lists with one element, there are two cases to be considered. The first rule specifies the case where the last element of the list is obtained after freeing other elements of the list. We illustrate the rewrite step on the graph obtained earlier (up to renaming of nodes). The second rule deals with the special case of circular lists of size one.



Example 8 (Memory usage analysis). Predicting memory usage of an algorithm can be of great interest for many applications, especially those involved within embedded systems. There exist several methods to analyse memory usage. For example, in [11], a very powerful method based on a type system enriched with resource annotations has been proposed. The authors succeeded in analysing several examples, but failed to tackle a program which flattens a list of lists. As an application of proposition 3, we show how our framework can be used to analyse the memory usage of such an algorithm. The program which flattens a list of lists, consists of the following rules, the first one is for the base case (the empty list) and the second rule deals with a non-empty list of lists. Here $+$ stands for the append operator and $+'$ for its auxiliary operator, as in example 6.



Thus, memory usage by flattening a list can be analyzed considering seven rules: two for flat, two for + and three for $+'$. Inspection of the recursion rules for the three functions shows that the number of nodes is unchanged. The halt case rule for flat frees one memory cell. The halt case rules, both for + and $+'$, free two memory cells. Now, simple reasoning on the rules involved in the evaluation of flat shows that flat(ℓ) frees exactly $2|\ell| + 1$ memory cells, where $|\ell|$ is the size of the list ℓ. Indeed, there is one halt case for flat, and for each element of ℓ there is one halt case for + or another one for $+'$.

5 Related Work

Cloning is also one of the features of the sesqui-pushout approach to graph transformation [4]. In this approach, a rule is a span $L \leftarrow K \rightarrow R$ and the application of a rule to a graph G can be illustrated by the same figure as for a DPO step (as in the introduction), where the right-hand side is a pushout as in the DPO approach but the left-hand side is a pullback, and moreover it is a final pullback complement. The graphs considered in [4] are defined as $G = (V, E, \mathtt{src} : E \rightarrow V, \mathtt{tgt} : E \rightarrow V)$ where V and E are the sets of vertices and edges respectively, and the connections of edges are defined by the functions `src` (source) and `tgt` (target). Nodes are not endowed with arities and thus they may have an arbitrary amount of outgoing edges. This fact, together with the use of final pullback complements, makes the sesqui-pushout approach different from our framework. We illustrate this difference through the cloning of nodes. According to the sesqui-pushout approach, the cloning of a node is, roughly speaking, performed by copying a node together with all its incident edges (incoming and outgoing edges). In our framework, a node is copied only with its outgoing edges. Let us consider for instance the termgraph $h(f(2:a), 2)$ in which the subgraph $f(2:a)$ is supposed to be transformed into $g(2, 3)$ where nodes 2 and 3 are clones of $2:a$. Then according to our framework, this transformation can be achieved by means of the following rule. The application of this rule to $h(f(2:a), 2)$ yields the graph $h(g(2:a, 3:a), 2)$.

$$\boxed{2:\bullet \leftarrow 1:f} \quad \boxed{2|2:\bullet \leftarrow 1:g \rightarrow 3|2:\bullet}$$

Using the sesqui-pushout approach, we get a rule of the following shape.

$$\boxed{2:\bullet \leftarrow 1:f} \longleftarrow \boxed{K} \longrightarrow \boxed{2:\bullet \leftarrow 1:g \rightarrow 3:\bullet}$$

Indeed, K should encode the cloning of the instance of node 2 as well as the replacement of f by g, thus K should include at least three unlabeled nodes. The application of this rule to $h(f(2:a), 2)$ yields the graph $h(g(2:a, 3:a), 2, 3)$, where the operation h has three arguments, because each of the clones $2:a$ and $3:a$ requires the cloning of all incoming edges.

Cloning has also been subject of interest in [6]. The authors considered rewrite rules of the form $S := R$ where S is a star, i.e., S is a (nonterminal) node surrounded by its adjacent nodes together with the edges that connect them.

Rewrite rules which perform the cloning of a node have been given in [6, Def. 6]. These rules show how a star can be removed, kept identical to itself or copied (cloned) more than once. Here again, unlike our framework, the cloning does not care about the arity of the nodes and, as in the case of the sesqui-pushout approach, a node is copied together with all its incoming and outgoing edges. If we consider the termgraph $h(f(2:a),2)$ again and clone twice the node $2:a$, then according to [6] we get the graph $h(f(2:a,3:a),2,3)$ where both h and f have augmented their arity when copying the incoming edges to the clone $3:a$.

A categorical framework dedicated to *cyclic* termgraph transformation can be found in [3] where the authors propose, following [16], a 2-categorical presentation of termgraph rewriting. They almost succeed in representing the full operational view of termgraph rewriting as defined in [2], but they differ on rewriting circular redexes. For example, the application of the rewrite rule $f(x) \to x$ on the termgraph $n : f(n)$ yields the same termgraph (i.e., $n : f(n)$) according to [2] but yields an unlabeled node, say $p:\bullet$, according to [3]. With our definition of rewrite rules, it is possible to encode exactly the algorithmic approach [2] by simply stating that node $n, m|m$ is a clone of node m in the rule:

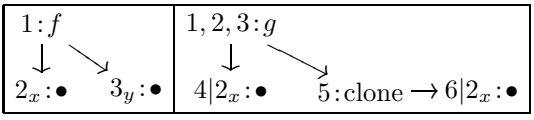
$n:f \longrightarrow m:\bullet$	$n, m\|m:\bullet$

In general, term rewriting and termgraph rewriting do not coincide [15]. However, thanks to its cloning facilities, our framework can simulate term rewriting in the case of left linear term rewrite systems. Indeed, a term rewrite rule $l \to r$ where l is a linear term (i.e., variables in l occur only once in l), can be transformed into a rule (L, R, τ, σ) where $L = \texttt{l2L}(l)$ and $R = \texttt{r2R}(r)$ are termgraphs corresponding respectively to l and r, by the transformations `l2L` and `r2R` defined below. Notice that the transformation of the right-hand side takes into account the cloning of variable instances, this happens when a right-hand side is not linear. The aim of τ, when simulating term rewriting, consists in indicating the replacement of the root of L by that of R, i.e., $\tau(\texttt{root}(L)) = \texttt{root}(R)$. The images via τ of the remaining nodes are not significant, so that $\tau(|L|) = \texttt{root}(R)$ is a possible choice for τ. The function σ indicates the parts that should be cloned, it plays an important role in encoding the use of variables in the right-hand side. Every occurrence of a variable x in r corresponds to a non labeled node $p:\bullet$ in R. In this case, by definition of rewrite rules, x appears in l and thus a corresponding non labeled node $p_x:\bullet$ appears in L, thus we state that p is a clone of p_x by $\sigma(p) = p_x$. Now we define the transformation functions `l2L` and `r2R`.

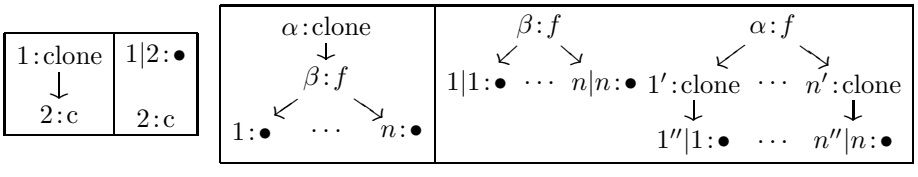
- If c is a constant then $\texttt{l2L}(c) = p:c$ and $\texttt{r2R}(c) = p:c$ where node p is new.
- If f is an operation and the t_i's are terms then $\texttt{l2L}(f(t_1,\ldots,t_n) = p : f(\texttt{l2L}(t_1),\ldots,\texttt{l2L}(t_n))$ and $\texttt{r2R}(f(t_1,\ldots,t_n) = p : f(\texttt{r2R}(t_1),\ldots,\texttt{r2R}(t_n))$ where node p is new.
- If x is a variable then
  - $\texttt{l2L}(x) = p_x:\bullet$ where node p_x is new

- $\texttt{r2R}(x) = p:\bullet$ for the first occurrence of x in r where node p is new and $\sigma(p) = p_x$, while $\texttt{r2R}(x) = q:\text{clone}(p:\bullet)$ otherwise, where nodes p and q are new and $\sigma(p) = p_x$.

For example, the term rewrite rule $f(x, y) \to g(x, x)$ is transformed as:



When applying a matching substitution m over the right-hand side r of a term rewrite rule, for every variable x of the left-hand side m constructs as many copies of the instance $m(x)$ as the number of occurrences of x in r. The aim of $q:\text{clone}(p:\bullet)$ in the definition of $\texttt{r2R}(x)$ is to mimic the application of a matching substitution on the right-hand side $m(r)$. The function clone builds a copy of its argument, it can be defined for all operators of a given signature as follows, where c is a constant and f is an operation of arity n. We write $\to^*_{\texttt{CLONE}}$ the rewrite relation induced by the following rules.



Proposition 4. *Let $\mathcal{R}$ be a left linear term rewrite system built over a signature Ω. Let $T(\mathcal{R})$ be the termgraph rewrite system made of the rewrite rules which define the function clone over the operation symbols in Ω, together with the transformations of the rules $l \to r$ in R. Let t be a ground term. If $t \to_{\mathcal{R}} t'$ then there exists a termgraph G_1 such that $\texttt{12L}(t) \to_{T(\mathcal{R})} G_1 \to^*_{\texttt{CLONE}} \texttt{12L}(t')$.*

The proof of this proposition is quite obvious, because the term t is assumed to be ground. However, reduction of non linear terms is not allowed by the given transformations. Indeed, variables cannot be cloned by transformation 12L (but only renamed). For example $\texttt{12L}(f(x, x))$ is the termgraph $1 : f(2_x : \bullet, 3_x : \bullet)$, which is not a sound translation of $f(x, x)$. For the instances of nodes 2_x and 3_x are not supposed to be isomorphic.

6 Conclusion

In this paper, we have proposed a new way to define termgraph rewrite rules, and we have defined a rewrite step as a heterogeneous pushout in an appropriate category. The proposed rewrite systems offer the possibility to transform cyclic termgraphs either by performing local edge redirections or global edge redirections, as defined following a DPO approach in [7], and in addition, it provides new features such as cloning or deleting nodes. Future work includes the generalization of the proposed systems to other graphs less constrained than termgraphs.

Acknowledgements

We would like to thank Andrea Corradini and the referees for their insightful comments on an earlier version of this paper.

References

1. Ariola, Z., Klop, J.: Equational term graph rewriting. Fundamenta Informaticae 26(3-4) (1996)
2. Barendregt, H., van Eekelen, M., Glauert, J., Kenneway, R., Plasmeijer, M.J., Sleep, M.: Term graph rewriting. In: de Bakker, J.W., Nijman, A.J., Treleaven, P.C. (eds.) PARLE 1987. LNCS, vol. 259, pp. 141–158. Springer, Heidelberg (1987)
3. Corradini, A., Gadducci, F.: A 2-categorical presentation of term graph rewriting. In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 87–105. Springer, Heidelberg (1997)
4. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 30–45. Springer, Heidelberg (2006)
5. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation - part I: Basic concepts and double pushout approach. In: Handbook of Graph Grammars, pp. 163–246 (1997)
6. Drewes, F., Hoffmann, B., Janssens, D., Minas, M., Eetvelde, N.V.: Adaptive star grammars. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 77–91. Springer, Heidelberg (2006)
7. Duval, D., Echahed, R., Prost, F.: Modeling pointer redirection as cyclic term-graph rewriting. ENTCS 176(1), 65–84 (2007)
8. Echahed, R.: Inductively sequential term-graph rewrite systems. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 84–98. Springer, Heidelberg (2008)
9. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic approaches to graph transformation - part II: Single pushout approach and comparison with double pushout approach. In: Handbook of Graph Grammars, pp. 247–312 (1997)
10. Ehrig, H., Pfender, M., Schneider, H.J.: Graph-grammars: An algebraic approach. In: FOCS 1973, The University of Iowa, USA, October 15-17, pp. 167–180. IEEE, Los Alamitos (1973)
11. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL 2003, pp. 185–197 (2003)
12. Kennaway, R.: On "on graph rewritings". Theor. Comput. Sci. 52, 37–58 (1987)
13. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theor. Comput. Sci 109(1&2), 181–224 (1993)
14. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Springer, Heidelberg (1998)
15. Plump, D.: Term graph rewriting. In: Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, vol. 2, pp. 3–61. World Scientific, Singapore (1999)
16. Power, A.J.: An abstract formulation for rewrite systems. In: Dybjer, P., Pitts, A.M., Pitt, D.H., Poigné, A., Rydeheard, D.E. (eds.) Category Theory and Computer Science. LNCS, vol. 389, pp. 300–312. Springer, Heidelberg (1989)
17. Raoult, J.C.: On graph rewriting. Theoretical Computer Science 32, 1–24 (1984)
18. Sleep, M.R., Plasmeijer, M.J., van Eekelen, M.C.J.D. (eds.): Term Graph Rewriting. Theory and Practice. J. Wiley & Sons, Chichester (1993)