Typed lambda-terms in categorical attributed graph transformation

Bertrand Boisvert, Louis Féraud, Sergei Soloviev
IRIT
Université Paul Sabatier
Toulouse, France
{boisvert,feraud,soloviev}@irit.fr

This paper deals with model transformation based on attributed graph rewriting. Our contribution investigates a single pushout approach for applying the rewrite rules. The computation of graph attributes is obtained through the use of typed λ-calculus with inductive types. In this paper we present solutions to cope with single pushout construction for the graph structure and the computations functions. As this rewrite system uses inductive types, the expressiveness of attribute computations is facilitated and appears more efficient than the one based on Σ-algebras. Some examples showing the interest of our computation approach are described in this paper.

1 Introduction.

There is currently a need of rigorous support to model based software engineering. In model-driven software engineering, models are mostly described using a graphical syntax (UML, SDL, ...). Models are composed of a structural part which can be represented as a graph and of attributes which are informations attached to vertices or edges of the graph. Thus, models can be formalized as attributed graphs and model transformation as attributed graph transformation. An attributed graph transformation is composed of a rewrite of the structural part and of computations on its attributes. Thus, we need a formal framework that can express these two types of transformation.

Graph rewriting systems based on category theory have been widely used to deal with the transformation of structural part. One of the challenges of attributed graph rewriting systems concerns the implementation of attribute computations. Most of the existing systems based on category theory adopt the standard algebraic approach where graphs are attributed using algebraic data types represented by Σ-algebras [9, 13]. However, the implementation of computations with algebraic data types meets many difficulties, and for the sake of efficiency considerations and convenient uses, these systems do not generally implement the whole attribute computations but rely on programs written in a host-language [1].

In our earlier work [15, 16, 18] we suggested to use inductive types and lambda terms in combination with a modification of the double pushout approach [17] called DPoPb ("double pushout-pullback" approach). Our goal was to use the well developed double pushout approach to implement rewriting of the structural part of graphs and to use the expressive power of λ-terms and inductive types to describe and facilitate attribute computations. But the construction of the double pushout imposed restricting constraints on computation functions mostly due to the usage of total maps and the obligation to split computations into two parts. That is why we now present a new approach based on single pushout.

The first section of this paper introduces the main approaches of graph rewriting based on category theory, and particularly the single pushout approach on which our approach is based. Second we define our category of attributed graphs, and then explain how to apply a rewrite rule by the computation of a weak pushout. Finaly we present examples.

Francisco Durán and Vlad Rusu (Eds.):
Second International Workshop on Algebraic Methods
in Model-Based Software Engineering 2011 (AMMSE'11)
EPTCS 56, 2011, pp. 33–47, doi:10.4204/EPTCS.56.3

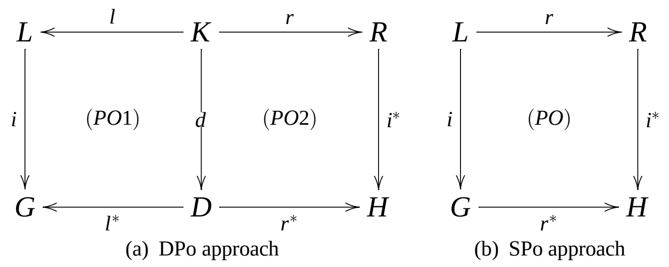
2 Categorical graph rewriting.

In graph rewriting systems based on category theory, we usualy define a category whose objects are graphs and morphisms are graph homomorphisms. A transformation rule is composed of at least two graphs called the left-hand side (usually noted L) and right-hand side (usually noted R). The left-hand side describes which subgraph a graph G must contain in order that the transformation could be applied to it, and the right-hand side describes how this part will look like after the transformation. Morphisms between left-hand side and right-hand side describe which parts of graphs will be deleted, transformed or added. To apply a rule to some subgraph of a larger graph G, we need first to embed the left-hand side as a subgraph of G. The embedding is represented by an inclusion $L \xrightarrow{i} G$. Cf Figure 1(a) and 1(b).

There are two principal categorical approaches to graph rewriting: double pushout (abbreviated DPo, concieved by H. Ehrig and his colleagues [7], [17]) and single pushout (abbreviated SPo, mainly devellopped by Löwe [12], [17]). The main difference is that in DPo morphisms are total maps and in SPo morphims are partial maps. This implies different forms of rules.

In the DPo approach a rule is defined by 3 graphs and 2 total morphisms: $L \xleftarrow{l} K \xrightarrow{r} R$. The morphism l indicates what vertices or edges should be erased (the ones who are not in the image of l) and the morphism r indicates what vertices or edges should be transformed (those who are in K), and added (those who are not in the image of r). The application of the rule is done by a computation of a pushout-complement (adding the arrows $K \xrightarrow{d} D$ and $D \xrightarrow{l^*} G$ and then a pushout (the arrows $R \xrightarrow{i^*} H$ and $D \xrightarrow{r^*} H$). Cf Figure 1(a).

In the SPo approach, a rule is defined by one partial morphim $L \xrightarrow{r} R$. Vertices and edges not included in the domain of r will be deleted, the ones in the domain of r will be transformed and those which are not in the image of r will be added. The application of the rule is done by the computation of one pushout (adding the arrows $G \xrightarrow{r^*} H$ and $R \xrightarrow{i^*} H$). Cf Figure 1(b)



$$\begin{tikzcd}
L \arrow[d, "i"'] \arrow[dr, phantom, "(PO1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(PO2)"] & R \arrow[d, "i^*"] \\
G & D \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

(a) DPo approach

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "i"'] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "i^*"] \\
G \arrow[r, "r^*"'] & H
\end{tikzcd}$$

(b) SPo approach

Figure 1: Classical categorical graph rewriting approaches

Because not all pushout-complements necessarily exist in the categories of graphs, there exist "application conditions" in DPo approach. As a consequence, rules that create dangling edges are forbidden in the DPo approach while in SPo approach dangling edges are removed when the rule is applied. If necessary, it is possible to add application conditions in the SPo approach as well. Thus the SPo approach is more general than the DPo approach, but SPo approach remained less developed due, in our opinion, mostly to historical reasons and to the fact that computation of pushout in categories of partial maps is more difficult than in categories of total maps.

Both approaches met many difficulties on the level of attribute computations. Our experience with the DPoPb approach [16, 15, 18] and the usage of λ-terms for attributes was encouraging but the construction of a double pushout imposed us some constraints due to the use of total maps and the obligation to split computation into two parts. The new approach we present here is more direct and natural, free of application conditions and we have no more constraints on the computational level. The rewriting process involves structure transformation and attribute computations. In this paper, we would stress on attribute computations and lighten the structure rewrite. We hope that the combined use of SPo and λ-terms will permit to overcome many known difficulties of attribute computations.

To develop a categorical graph rewriting system we must define a category (objects and morphisms) and then explain how to apply a rule (in our case by the computation of a pushout).

A pushout of two morphisms $L \xrightarrow{r} R$, $L \xrightarrow{i} G$ is a couple of morphisms ($G \xrightarrow{r'} H$, $R \xrightarrow{i'} H$) such that:

- $i' \circ r = r' \circ i$
- for every other couple of morphisms ($R \xrightarrow{h} H'$, $G \xrightarrow{g} H'$) such that $h \circ r = g \circ i$ it exists a unique morphism c such that the diagram below commutes:

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "i"'] & R \arrow[d, "i'"] \arrow[rdd, "h"] & \\
G \arrow[r, "r'"] \arrow[rrd, "g"'] & H \arrow[rd, dashed, "c"] & \\
 & & H'
\end{tikzcd}$$

As a consequence, the existence of pushout implies the uniqueness of the object H up to isomorphism (cf. [6, 12]). If we have the two properties in the definition of pushout but not the unicity of c, the construction is called a weak pushout.

3 Category of Attributed Graphs.

We shall denote Gr^T the category of graphs we consider below.

Objects. Objects of Gr^T are oriented attributed graphs. We shall assume that the vertices and edges are sets (noted $V(G)$ and $E(G)$ for a graph G) of natural numbers with $V(G) \cap E(G) = \emptyset$ and that a standard (lexicographic) ordering on vertices and edges is defined and noted $<$. This will help us to avoid ambiguity in the definition of morphisms, and in any case this assumption is standard when implementations are considered. The attributes will be λ-terms. The system of lambda calculus in this paper is the simply typed lambda calculus with surjective pairing, terminal object and inductive types, (see for example [3]). We shall denote the function type by $A \to B$ and the product type for types A and B by $A \times B$. The presence of inductive types permits to define all ordinary types of attributes, like $Bool$, Nat, etc., as well as more complex types like lists, binary trees, ω-trees, etc. We prefer to include pairings in the syntax directly instead of defining it using inductive types.

We decide to have exactly one attribute per node or edge. Pairing permits to embed different datas into this unique attribute. We can see each attribute as a tuple containing all informations attached to a node or an edge. The n-tuple $< t_1, ..., t_n >$ is considered as an abbreviation of the term $< ... < t_1, t_2 >$

$, ..., t_n >$. If $A_1, ..., A_n$ are types of $t_1, ..., t_n$ respectively, the type of this tuple will be written as $A_1 \times ... \times A_n$ instead of $(...(A_1 \times A_2) \times ...A_n)$. We shall use trivial attribute $0 : T$ (T - terminal object) to represent the absence of attributes. Thus we have a bijection between the set of nodes and edges and the set of attributes which permits to simplify some proofs.

If G is an attributed graph, $V(G)$ is the set of its vertices, $E(G)$ the set of its edges, $att(v)$ where $v \in V(G) \cup E(G)$ is the corresponding attribute (λ-term).

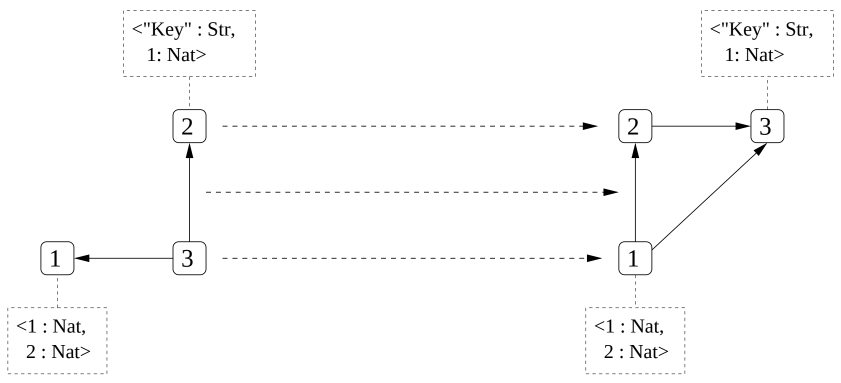
Three-level morphisms. Let G, H be two attributed graphs. We shall assume that all λ-terms considered below are typed in the same context Γ. This context may be fixed for the whole category, or at least sufficient for all graphs and terms in consideration. The terms are not necessarily closed. The equality of terms is understood in ordinary sense as equality w.r.t. α, β, η and also ι conversion for recursion[1]. Morphisms $f : G \to H$ are defined using the following three-level construction:

1. The "structural part" f_{str} is a partial graph homomorphism (without attributes) of G to H (cf. Fig. 2(a)).
2. The "attribute dependency relation" f_{adr} is a relation between the sets $V(G) \cup E(G)$ and $V(H) \cup E(H)$ induced by computation functions. For each $v \in V(H) \cup E(H)$ its preimage (i.e. the set of all its antecedents) is $[v]_{f_{adr}} \subseteq V(G) \cup E(G)$ and represents all attributes of graph G that we can use to compute v. (We could consider here instead of vertices and edges corresponding attributes, cf. Fig. 2(b).)
3. The "computational part" is represented, by the λ-term $f_{cmp}(v)$, for each element $v \in V(H) \cup E(H)$. These λ-terms will be called computation functions. They are functions that take as argument attributes of graph G defined by f_{adr}. More precisely, let $v \in V(H) \cup E(H)$. Let $att(v) = t : A$ be the corresponding attribute. Let $[v]_{f_{adr}} = \{u_1, ..., u_k\}, u_1 < ... < u_k$ (we use here the fact that the vertices are natural numbers) and
$$att(u_1) = t_1 : A_1, ..., att(u_k) = t_k : A_k$$
be the attributes of the antecedents. Now the term $s_v = f_{cmp}(v)$ has the type
$$A_1 \to ... \to A_k \to A$$
and should satisfy the following property:
$$s_v(t_1, ..., t_k) = t$$
(of course, many arguments may be "dummy"). In particular, if $[y]_{f_{adr}} = \emptyset$ then $s_v = t$. (See Fig. 2(c).)

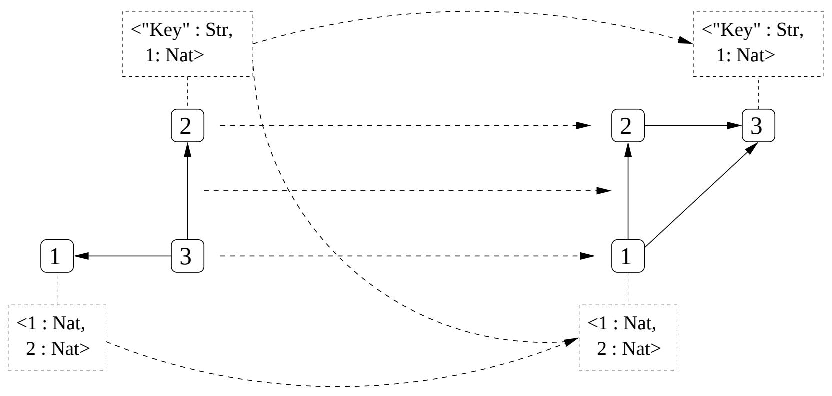
Equality of morphisms. Two morphisms $f, g : G \to H$ are equal if

1. $f_{str} = g_{str}$;
2. the relations $f_{adr} = g_{adr}$;
3. and for each $v \in V(H) \cup E(H)$ their computation functions f_{cmp_v} and g_{cmp_v} are equals on their arguments in G. More precisely, let $[v]_{f_{adr}} = [v]_{g_{adr}} = \{u_1, ..., u_k\}$ be the arguments of f_{cmp_v} and g_{cmp_v}; we have:
$$f_{cmp_v}(u_1, ..., u_n) = g_{cmp_v}(u_1, ..., u_n)$$
as λ-terms.

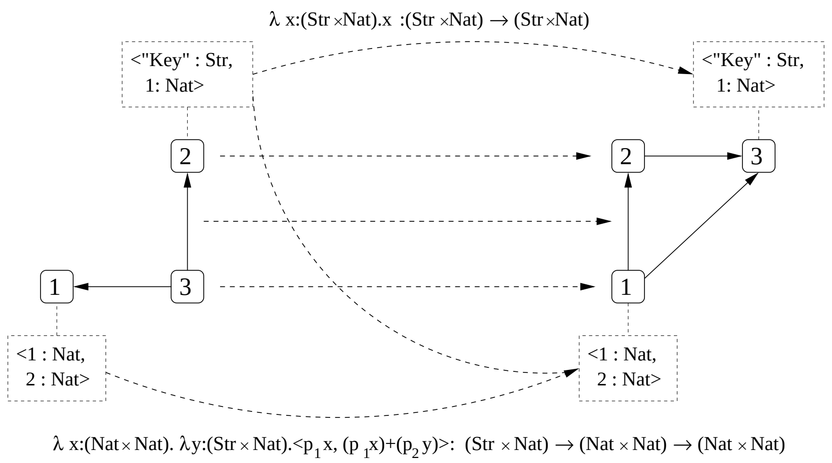
[1]In principle, other forms of equality could be considered.



(a) Structural part



(b) Structural part + attribute dependency relation


$\lambda x{:}(Str \times Nat).x : (Str \times Nat) \to (Str \times Nat)$
<"Key" : Str, 1: Nat>
<"Key" : Str, 1: Nat>
2
2
3
1
3
1
<1 : Nat, 2 : Nat>
<1 : Nat, 2 : Nat>
$\lambda x{:}(Nat \times Nat).\, \lambda y{:}(Str \times Nat).\langle p_1 x, (p_1 x)+(p_2 y)\rangle : (Str \times Nat) \to (Nat \times Nat) \to (Nat \times Nat)$

(c) Structural part + attribute dependency relation + computational part

Figure 2: Three-level attributed graph morphism

Remarks:

- One may notice that the equality of morphisms does not imply the equality of computations functions and two morphisms can be equal and have different computation functions.
- If the attributes of G are variables then the equality of functions on values is equivalent to the equality of computation functions. (This choice of variables as attributes is natural if we consider rule schemes instead of instance of the rules).
- Given two morphisms $f,g : G \to H$, the equality of first two levels $f_{str} = g_{str}$ and $f_{adr} = g_{adr}$ implies $f = g$ because the values of the attributes of H are the same.
- Taking into account the way how the rules are applied, it is in good accord with the intuition. A mosphism $r : L \to R$ is used for the formulation of a rule (or rule scheme) and it is natural that the values (attributes of R) are known. The computation occurs in the application of a rule (construction of a pushout).

Composition.

1. On the level of structure, we take the composition $g_{str} \circ f_{str}$;
2. On the level of attribute dependency relations we take the composition of relations $(g_{adr} \circ f_{adr})$. One may note that:
$$[w]_{(g\circ f)_{adr}} = [w]_{g_{adr}\circ f_{adr}} = \cup_{v\in[w]_{g_{adr}}} [v]_{f_{adr}}$$
3. On the level of computation functions (λ-terms) the composition is defined using composition of λ-terms.

   More precisely, let:

   - $t = g_{cmp}(w)$;
   - $[w]_{g_{adr}} = \{v_1,...,v_k\}, v_1 < ... < v_k$;
   - $t_1 = f_{cmp}(v_1),...,t_k = f_{cmp}(v_k)$;
   - $[v_1]_{f_{adr}} = \{u_{11},...,u_{1n_1}\},...,[v_k]_{f_{adr}} = \{u_{k1},...,u_{kn_k}\}$.

   The intersections of the antecedents may be not empty, so let $u_1 < ... < u_p$ be the distinct elements (vertices or edges) of the union $[v_1]_{f_{adr}} \cup ... \cup [v_k]_{f_{adr}}$. Let $A_1,...,A_p$ be the types of attributes $att(u_1),...,att(u_p)$ respectively, and $x_1 : A_1,...,x_p : A_p$ term variables not belonging to the context Γ. Since each of the elements u_{ij} corresponds to exactly one of $u_1,...,u_p$, we have $u_{ij} = u_m$ for some $m, 1 \le m \le p$, and for term variables we may put $x_{ij} = x_m$.

   Now we define:
$$(g \circ f)_{cmp}(w) =_{df} \lambda x_1 : A_1...\lambda x_p : A_p.(t(t_1x_{11}...x_{1n_1})...(t_kx_{k1}...x_{kn_k})).$$

Identity morphisms. For an attributed graph G, we call Id_G its identiy morphism defined by:

1. for the structural part, we take the identity graph homomorphism;
2. for the attribute dependency relation, we take the identity relation;
3. for the computation functions, for each $v \in V(G) \cup E(G)$, let A be the type of $att(v)$, $(Id_G)_{cmp}(v) = \lambda x : A.x : A \to A$.

Theorem. Attributed graphs and graph morphisms described above form a category.

Proof: Composition is associative due to associativity of the composition of graph homomorphisms, and associativity of the composition of relations. For λ-terms composition is associative too because of confluence and the fact that all simply typed λ-terms are strongly normalizable. Thus any evaluation strategy will terminate on a same simply typed λ-term. It is easy to verify that for every morphism $f : G \to H$ we have $f \circ Id_G = f$ and $Id_H \circ f = f$.

4 Weak pushout computation in category Gr^T.

As we said, to apply a rule $L \xrightarrow{r} R$ to a graph G we must find an embedding $R \xrightarrow{i} G$ and then compute the pushout of r and i. But in our attributed graphs, the attributes of R may contain free variables that are instanciated in G. So we shall see r as a rule "scheme". The application consists in general in two steps. First we take an instance of $L \xrightarrow{r} R$ obtained by substitution of λ-terms for certain free variables. Afterwards we try to "embed" the left side into G.

Injective attributed graph morphism: Let $f : G \to H$ be an attributed graph morphism. f is injective if:

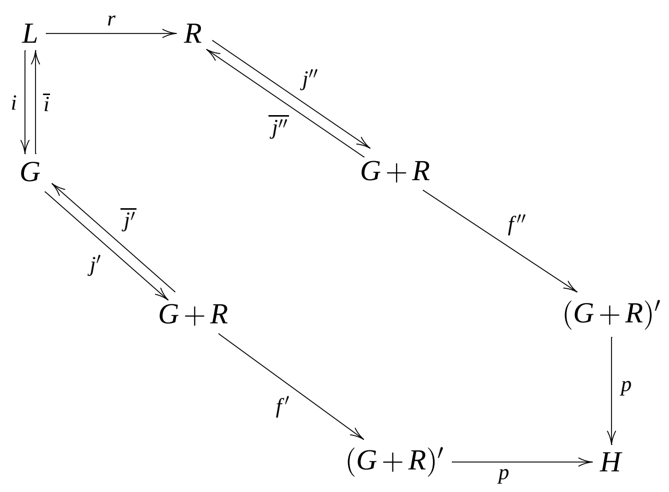
1. f_{str} is an injective partial graph homomorphism (i.e. $\forall v_1, v_2 \in V(G) \cup E(G).(f_{str}(v_1) = f_{str}(v_2) \Rightarrow v_1 = v_2)$);
2. $f_{adr} = f_{str}$;
3. for all $v' \in V(H) \cup E(H)$:
   - if $[v']_{f_{adr}}$ is empty, $f_{cmp}(v') = att(v')$,
   - if $[v']_{f_{adr}}$ is not empty (thus $[v']_{f_{adr}}$ is a singleton that we denote $\{v\}$ because f_{adr} is injective), $f_{cmp}(v') = \lambda x : A.x : A \to A$ where A is the type of $att(v)$.

Canonical retraction of a total injective attributed graph morphism: Let $f : G \to H$ be a total injective attributed graph morphism. A retraction of f (or a left inverse) is an attributed graph morphism $\overline{f} : H \to G$ such that $\overline{f} \circ f = Id_G$.

With this definition, we have not necessarily $f \circ \overline{f} = Id_H$, and $\overline{f}$ is not unique in general. That's why we give a canonical construction to obtain a retraction of f. This construction is defined by:

1. for every $v' \in V(H) \cup E(H)$ if $[v']_{f_{str}}$ is empty, v' has no image by $\overline{f}_{str}$; if $[v']_{f_{str}}$ is not empty (thus $[v']_{f_{str}}$ is a singleton that we denote $\{v\}$ because f_{str} is injective), $\overline{f}_{str}(v') = v$.
2. $\overline{f}_{adr} = \overline{f}_{str}$
3. for each $v \in V(G) \cup E(G)$:
   - if $[v]_{\overline{f}_{adr}}$ is empty, $\overline{f}_{cmp}(v) = att(v)$;
   - if $[v]_{\overline{f}_{adr}}$ is not empty (thus $[v]_{\overline{f}_{adr})}$ is a singleton that we denote $\{v'\}$), $\overline{f}_{cmp}(v) = \lambda x : A.x : A \to A$ where A is the type of $att(v')$.

As f is a total injection, it is easy to see that $\overline{f} \circ f = Id_G$.



$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, shift right, "i"'] & R \arrow[dr, shift left, "j''"] & & \\
G \arrow[u, shift right, "\bar{i}"'] \arrow[dr, shift right, "j'"'] & & G+R \arrow[ul, shift left, "\overline{j''}"] \arrow[dr, "f''"] & \\
& G+R \arrow[ul, shift right, "\overline{j'}"'] \arrow[dr, "f'"'] & & (G+R)' \arrow[d, "p"] \\
& & (G+R)' \arrow[r, "p"'] & H
\end{tikzcd}$$

Figure 3: Construction of weak pushout

Construction of a weak pushout The construction of a (weak) pushout in case of application of a rule is inspired by the paper by Löwe and others [17], but there will be differences due to our definition of attributed graphs and graph morphisms.

The “starting point” is the pair of morphisms ($L \xrightarrow{r} R, L \xrightarrow{i} G$) where i is injective and total attributed graph morphism as definded above. We want to compute the weak pushout ($R \xrightarrow{i'} H, G \xrightarrow{r'} H$) of this pair.

The definition of pushout in the paper by Löwe [17] uses coequalizers. We will have in mind this construction, but will give here a more restricted and straightforward definition, without detailed study of coequalizers in the category Gr^T.

First step to define a pushout using coequalizers in the category of graphs would be to take the coproduct $G+R$ of G and R (coproduct being here just the disjoint union). Next step would be to factorize it by certain equivalence relation (creating $(G+R)'$ which contains equivalence classes), and then to complete the construction using composition with certain morphism p from factor object to pushout object H.

We shall define each of the morphisms r' and i' as a composition of three morphisms (Cf figure 3) in order to have

$$r' = G \xrightarrow{j'} (G+R) \xrightarrow{f'} (G+R)' \xrightarrow{p} H$$

and

$$i' = R \xrightarrow{j''} (G+R) \xrightarrow{f''} (G+R)' \xrightarrow{p} H$$

The objects and morphisms in these diagrams are defined in several steps.

- On the level of structure $G+R$ is disjoint union of the graphs G and R;
- on the level of attributes each element of G and R in $G+R$ has the same attribute as in G and R;
- j' and j'' are inclusions respectively of G and R into $G+R$, thus they are total injective attributed graph morphisms.

To continue, we define first the equivalence relation $\sim_1$ on the elements of the graph structure $G+R$.

- let’s put $a \sim_1 b$ for $a, b \in G+R$ if $\exists x \in L.(j'(i(x)) = a \wedge j''(r(x)) = b)$

- then the relation $\sim$ is defined as reflexive, symmetric and transitive closure of $\sim_1$.
- notice that the elements of $G+R$ which are not the images of elements of $G-i(dom(r))$ form equivalence classes consisting of single element (itself).

The elements of $(G+R)'$ are defined as equivalence classes of elements of $G+R$. It is easily checked that this definition is consistent with the incidence relation and the map sending each element of $G+R$ to its equivalence class is a (total) graph homomorphism. This map will be structural part of f' and f''.

Moreover, each equivalence class with respect to $\sim$ containing an image of an element of R may be seen as a "span", consisting of the image of this element of R under j'' and the images of its antecedent via r under $j' \circ i$. In particular, each equivalence class contains exactly one image of an element of R. As a consequence, the composition $f''_{str} \circ j''_{str}$ is injective.

It permits also to define the attribute part of $(G+R)'$. Each equivalence class that contains an image of an element of R has the same attribute as this element has in R. Other equivalence classes (that have the form $\{j'(y)\}, y \in G, y \neq i(x)$ for some $x \in L$) keep the same attribute as in G.

The definitions of relational part and computation functions of f' and f'' are different.

For f'' the relation f''_{adr} connects the elements of R with corresponding equivalence classes (it is bijective on the R-part). There is no connections on the G-part. The computation functions are identities.

Remark. The composition $f'' \circ j''$ is injective, in particular $(f'' \circ j'')_{str}$ is an injective total graph homomorphism, $(f'' \circ j'')_{adr} = (f'' \circ j'')_{str}$ and computation functions are identities.

Now we may define f' as follows:

- $\forall v \in img(j' \circ i)$:
$$f' = G+R \xrightarrow{\overline{j'}} G \xrightarrow{\overline{i}} L \xrightarrow{r} R \xrightarrow{j''} G+R \xrightarrow{f''} (G+R)'.$$
- for the elements of $G-i(L)$, f' is like the identity.

As usual (cf. [17]) H is defined now as for coequalizer construction. Let $L_0 = dom(r)$. In our case H will be a subgraph of $(G+R)'$. The incidence relation in $(G+R)'$ is inherited from R and G. The elements on H (on the level of graph structure) are:

1. all the equivalence classes of the form $\{x_1,\dots,x_k,z\}$ $(x_1,\dots,x_k \in j'(i(L_0)), z \in j''(r(L))$;
2. all the equivalence classes of the form $\{z\}$, $z \in j''(R-r(L))$;
3. all the equivalence classes of the form $\{x\}$, $x \in j'(G-i(L))$ that are not dangling edges [17].

The attributes for the equivalence classes of the first two types are inherited from R and for the third from G.

The morphism p is defined as follows. Its structural part is identity on all elements of $(G+R)'$ that remain in H. We have also $p_{adr} = p_{str}$, and all computation functions are identities.

Now i', r' and H are defined such that $i' \circ r = r' \circ i$.

Let $h : R \to H'$ and $g : G \to H'$ be two other morphisms such that $h \circ r = g \circ i$. As i' is injective, we can use the canonical retraction $\overline{i'}$ and take for c (cf figure 4) $h \circ \overline{i'}$ and for elements who are not in the domain of $\overline{i'}$ we extend c in order to make it in accord with g. The commutativity on the level of computation functions follows from the definition of equality of attributed graph morphisms (cf section 3). Thus the diagram commutes but in general the unicity of c is not guaranteed, so we have a weak pushout.

$$\begin{tikzcd}
L \arrow[r, "r"] \arrow[d, "i"'] & R \arrow[d, "i'"] \arrow[ddr, "h"] & \\
G \arrow[r, "r'"] \arrow[drr, "g"'] & H \arrow[dr, dashed, "c"] & \\
& & H'
\end{tikzcd}$$

$$r' = p \circ f' \circ j'$$
$$i' = p \circ f'' \circ j''$$

Figure 4: Definition of Pushout

5 Examples.

To illustrate our transformation approach we present in this section two examples: the first one compares the graph grammar to compute $n!$ in our framework with the graph grammar given on the *AGG* website [1], i.e. the graph grammar corresponding to the framework based on Σ-algebras. And the second one presents computation on attributes representing infinite trees which is not possible by using Σ-algebras. Let us mention also some examples that we do not develop in this paper but that can be easily developped using our approach:

- graph cloning [2];
- information balance between attributes and structure;
- computation on functional attributes.

5.1 Computation of $n!$.

This example shows the advantage of our approach at computational level. Figure 5 presents the example of computing the factorial of a number n using two different graph rewriting systems. The first one is based on Σ-algebras and is a copy of an example given on the AGG website [1], and the other is based on our approach using λ-terms.

If we use classical approach based on Σ-algebras, we need three rules in two layers (layers define a priority of application on the rules [6] and are depicted by Roman ciffers on the figures) (cf figure 5(a)):

1. the first rule is used $n-2$ times and creates a chain with all values between n and 2 (the looping edge is used to specify which attribute must be decremented);
2. the second rule is used to terminate the excecution of the first rule (it consists in removing the looping edge);
3. the third rule is used $n-2$ times to multiply all numbers between n and 2.

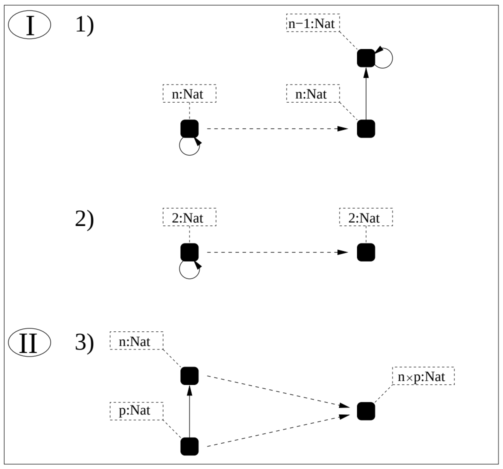
Thus to compute $n!$ we must apply $2n-3$ rules. This number of rule computations is due to the fact that in graph rewriting systems each rule can modelize the application of certain operations of a Σ-algebra, but putting to work recursion in Σ-algebras is difficult. Thus, the computation of a factorial is not straightforward.

If we use our framework based on λ-terms, only one rule is necessary to compute the factorial of n because recursion operators exist, thus it is easy to write a λ-term that compute $n!$ (Cf figure 5(b)). Of course computation of recursive functions requires many steps but it is included in a standard computational framework based on inductive types and optimized for such computations.

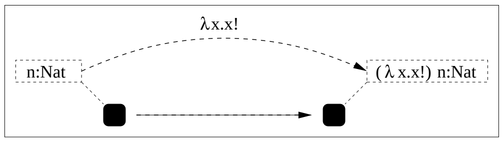
This reduction of the number of rules needed to compute on attributes has two advantages:

- the simplicity of the graph grammar;
- the application of a rule is expensive in terms of algorithmic complexity. To apply a rule we need to solve the problem of subgraph isomorphism (wich is NP-complete) to find a matching. If we simplify the left side of rules and if we reduce the number of applications we significantly reduce the time of computing of $n!$.

Resting on this example, it appears that the attribute computation is more expressive and certainly more efficient than the one based on Σ-algebras, particularly when addressing problems requiring recursion.



(a) $n!$ computation based on Σ-algebras



(b) $n!$ computations based on λ-terms. Here we write "$x!$" instead of the λ-term computing $x!$ to lighten the figure.

Figure 5: graph grammars to compute $n!$ in two different attributed graph rewriting systems

5.2 Managing infinity with functional attributes.

Another advantage of using λ-terms as attributes is the possibility to have complex data structures that can represent infinity. If we want to modelize infinite attributes we can do that by defining a function with an infinite data type as domain. For example the type T_ω of ω-trees (cf figure 6(a)) represents trees where nodes can have an infinity of subtrees. Using the recursion operators on inductive types we can define transformation on these infinite tree structures.

We may recall the form of recursive equations from Nat to any type A and from T_ω to B:

- $\psi(0) = a; \psi(succ(x)) = g(x, \psi(x))$ (for Nat)
- and $\phi(Leaf) = t; \phi(Succ_\omega(x) = g(x, \phi(x)); \phi(Lim(f)) = h(f, \phi \circ f)$ ($\circ$ denotes the composition).

In the syntax of λ-calculus the solution of these equation is denoted by

- $Rec^{Nat \to A}(a)(g)$
- and $Rec^{T_\omega \to B}(t)(g)(h)$.

Now the transformation of the trees we are considering may be written as follows.

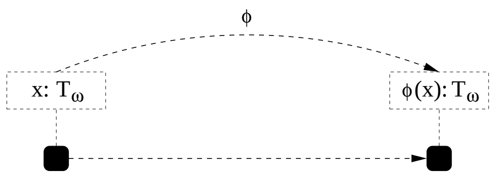
- Let d be defined by $d(0) = 0, d(Succ(x)) = Succ(Succ(d(x)))$, $d = Rec^{Nat \to Nat}(0)(\lambda x.\lambda y.Succ(Succ(y)))$, i.e. $d(x) = 2x$ in arithmetical notation.
- Let ϕ be defined by $\phi(Leaf) = Leaf, \phi(Succ_\omega(x)) = Succ(\phi(x)), \phi(Lim(f)) = Lim(f \circ d)$, using Rec, $\phi(f) = Rec^{T_\omega \to T_\omega}(Leaf)(\lambda x^{T_\omega}.Succ_\omega)(\lambda u.\lambda v.(u \circ d))$. This transformation selects (once) the branches with pair numbers at the first (infinite) branching.

Slightly more sophisticated transformation selects the branches with pair numbers at every infinite branching. There is only one modification. We define ϕ' by $\phi'(Leaf) = Leaf$, $\phi'(Succ_\omega(x)) = Succ(\phi'(x))$, $\phi'(Lim(f)) = Lim((\phi' \circ f) \circ d)$. Using Rec, $\phi'(f) = Rec^{T_\omega \to T_\omega}(Leaf)(\lambda x^{T_\omega}.Succ_\omega)(\lambda u.\lambda v.(v \circ d))$.

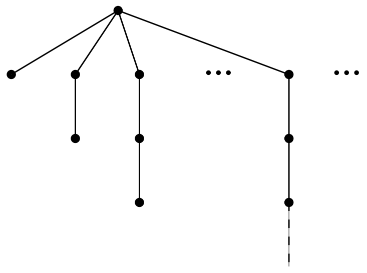
The figure 6(b) presents a rule that selects the branches with pair numbers at the first infinite branching (using ϕ defined above). The figure 6(c) presents an example of $\omega - tree$ and the figure 6(d) presents the result of the application of the rule on it.

$$T_\omega = Ind\alpha\{Leaf : \alpha,\; Succ_\omega : \alpha \to \alpha,\; Lim : (Nat \to \alpha) \to \alpha\}$$

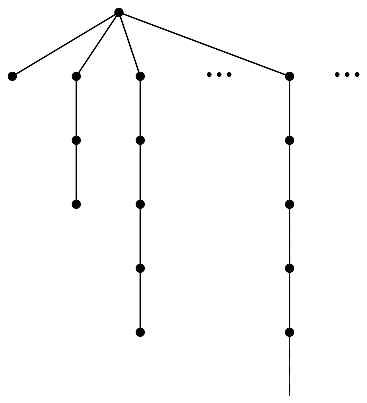
(a) Definition of the inductive type ω-trees



(b) Transformation rule



(c) Example of ω-tree defined by the term $Lim(Rec^{Nat \to T_\omega}(Leaf)(\lambda x^{Nat} \lambda y^{T_\omega}.Succ_\omega(y)))$. The length of the n-th branch is n.



(d) Result of the application of the rule 6(b) on a graph with the $\omega - Tree$ of figure 6(c) as attribute: only the branches with pair numbers are selected.

Figure 6: Computation on attributes representing infinite trees.

This is a very simple example but we can imagine a more complex example where leafs are elements of a complex type.

6 Conclusion.

The aim of this paper was to present a new attributed graph rewriting system based on the the SPo approach and whose main originality concerns the use of a typed λ-calculus to express attribute computations. On the structural parts our approach has the same characteristics than the classical SPo approach, but on the computation on attributes we have shown by examples that we can simplify the grammars, extend the expressivity of rules and certainly gain in efficiency of the computation.

Thanks to the expressive power of inductive types, it is now possible to dispatch some rewrite mechanism from structure to attribute computation and back (attributes can represent certain types of graphs e.g. trees).

Theoretically speaking, the SPo approach necessitates the definition and the construction of a weak pushout when dealing with attributes. A solution is presented in this paper. The next step of this work will concern the study of usual properies of any rewriting system such as confluence, termination, critical pairs analysis, etc. Note that for attribute computations these properties are well known properties of λ-calculus. In addition, we are now investigating another way to describe transformation of attributes, based on a calculus using deduction rules.

The possible domains of applications include all usual applications of graph transformations, e.g., verification and model transformations in programming, but more “tight” relationship between computational and structural parts will permit also the pursuit of much more specific goals.

References

[1] *AGG Homepage*, `http://tfs.cs.tu-berlin.de/agg/`.

[2] Sergey Baranov, Bertrand Boisvert, Louis Feraud & Sergei Soloviev (2011): *Typed Lambda Terms in Categorical Graph Rewriting*. In N.N. Vassiliev, editor: *The International Conference Polynomial Computer Algebra, April 18-22-2011, Saint-Petersburg, Russia, Euler International Mathematical Institute*, VVM Publishing, pp. 9–16.

[3] David Chemouil (2005): *Isomorphisms of simple inductive types through extensional rewriting*. *Math. Structures in Computer Science* 15(5), doi:`10.1017/S0960129505004950`.

[4] David Chemouil & Sergei Soloviev (2003): *Remarks on isomorphisms of simple inductive types* . In: *Mathematics, Logic and Computation , Eindhoven, 04/07/03-05/07/03*, Elsevier, ENTCS 85, 7, pp. 1–19, doi:`10.1016/S1571-0661(04)80760-6`.

[5] Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors (2002): *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*. *LNCS* 2505, Springer.

[6] H. Ehrig, K. Ehrig, U. Prange & G. Taentzer (2006): *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[7] Hartmut Ehrig (1978): *Introduction to the Algebraic Theory of Graph Grammars (A Survey)*. In: *Graph-Grammars and Their Application to Computer Science and Biology*, pp. 1–69.

[8] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce & Grzegorz Rozenberg, editors (2004): *Graph Transformations, Second International Conference, ICGT 2004, Rome, Italy, September 28 - October 2, 2004, Proceedings*. *LNCS* 3256, Springer, doi:`10.1007/b100934`.

[9] Hartmut Ehrig, Julia Padberg, Ulrike Prange & Annegret Habel (2006): *Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation*. *Fundam. Inf.* 74(1), pp. 1–29.

[10] Hartmut Ehrig, Ulrike Prange & Gabriele Taentzer (2004): *Fundamental Theory for Typed Attributed Graph Transformation*. In Ehrig et al. [8], pp. 161–177, doi:`10.1007/978-3-540-30203-2_13`.

[11] Annegret Habel, Reiko Heckel & Gabriele Taentzer (1996): *Graph Grammars with Negative Application Conditions*. *Fundam. Inform.* 26(3/4), pp. 287–313.

[12] Michael Löwe (1993): *Algebraic approach to single-pushout graph transformation*. *Theor. Comput. Sci.* 109, pp. 181–224, doi:`10.1016/0304-3975(93)90068-5`.

[13] Fernando Orejas (2011): *Symbolic graphs for attributed graph constraints*. *J. Symb. Comput.* 46, pp. 294–315, doi:`10.1016/j.jsc.2010.09.009`.

[14] Maxime Rebout (2008): *Une approche catégorique unifiée pour la récriture de graphes attribués*. Ph.D. thesis, Université Paul Sabatier.

[15] Maxime Rebout, Louis Féraud, Lionel Marie-Magdeleine & Sergei Soloviev (2009): *Computations in Graph Rewriting: Inductive types and Pullbacks in DPO Approach*. In Tomasz Szmuc, Marcin Szpyrka & Jaroslav Zendulka, editors: *IFIP TC2 Central and East European Conference on Software Engineering Techniques - CEE-SET, Krakow, Pologne, 12/10/09-14/10/09*, Springer-Verlag, pp. 164–177.

[16] Maxime Rebout, Louis Féraud & Sergei Soloviev (2008): *A Unified Categorical Approach for Attributed Graph Rewriting*. In E Hirsch & A Razborov, editors: *International Computer Science Symposium in Russia (CSR 2008), Moscou 07/06/2008-12/06/2008*, *LNCS* 5010, Springer-Verlag, pp. 398–410, doi:`10.1007/978-3-540-79709-8_39`.

[17] Grzegorz Rozenberg, editor (1997): *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, doi:`10.1142/9789812384720`.

[18] Hanh Nhi Tran, Christian Percebois, Ali Abou Dib, Louis Féraud & Sergei Soloviev (2010): *Attribute Computations in the DPoPb Graph Transformation Engine (regular paper)*. In: *4th International Workshop*

on Graph Based Tools (GRABATS 2010), University of Twente, Enschede, The Netherlands, 28/09/2010-28/09/2010, University of Twente, http://www.utwente.nl/en, p. (electronic medium).