Incorporating Proofs in a Categorical Attributed Graph Transformation System for Software Modelling and Verification[*]

Bertrand Boisvert, Louis Féraud and Sergei Soloviev
IRIT, Université Paul Sabatier, Toulouse, France

Keywords: Model Transformation, Graph Transformation, Attribute Computation, Single Pushout.

Abstract: This paper deals with model transformations based on attributed graphs transformation. Our approach is based on the categorical approach called Single Pushout. The principal goal being to strengthen the attribute computation part, we generalize our earlier approach based on the use of typed lambda-terms with inductive types and recursion to represent attributes and computation functions. The generalized approach takes terms in variable context as attributes and partial proofs as computation functions that permit to combine computation with proof development and verification. The intended domains of application are the development of cerified software models and semantics models for interactive proof development and verification.

1 INTRODUCTION

In Model Driven Engineering (abbreviated MDE), models are mostly described using a graphical syntax (UML, SDL, etc.). 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 transformations. An attributed graph transformation is composed of a rewrite of the structural part and of computations on its attributes.

When considering graph transformations, it can be noticed that a lot of them deal only with structures and not with attributes. For instance transforming UML class diagrams to relational models or UML activity diagrams to Petri nets require few attribute computations. In contrast, when dealing with formalisms such as timed automata, complex Petri nets (predicate Petri nets, Object Petri nets, colored Petri nets), internal program representation leads to sophisticated attribute computations. In our previous and current works we decided to focus on graph transformations requiring complex computations. So we developped a first approach based on a typed λ-calculus and now we moved a step further considering inference rules as a way to make attribute computations.

One of the challenges of attributed graph transformation 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 (Ehrig et al., 2006b), (Orejas, 2011). However, the computation with algebraic data types does not permit to represent certain computations (like computation of recursive functions or term matching), and meets efficiency problems when implemented.

In our earlier work, see (Rebout et al., 2011), (Rebout et al., 2008), (Tran et al., 2010) we suggested to use inductive types and lambda terms in combination with a modification of the double pushout approach (Rozenberg, 1997) called DPoPb ("double pushout-pullback" approach). As stated above, our goal was to use a well developed 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 strong constraints on computation functions mostly due to the usage of total maps and the obligation to split all the computations into two parts. That is why later we presented a new approach based on single pushout and λ-terms as computation functions (Boisvert et al., 2011a).

In this paper, we generalize this approach towards models incorporating proofs. Instead of λ-terms in the same fixed context we consider full typing judgements of the form $\Gamma \vdash t : A$ where Γ is a context, t is a term and A is a type. Instead of λ-terms as com-

[*]Part of this research has been supported by the Climt project, ANR-11-BS02-016-02

62 Boisvert B., Féraud L. and Soloviev S..
Incorporating Proofs in a Categorical Attributed Graph Transformation System for Software Modelling and Verification.
DOI: 10.5220/0004321200620074
In *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2013), pages 62-74
ISBN: 978-989-8565-42-6
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

putation functions we take partial proofs in the corresponding system of type theory. As a result, the scope of the approach is considerably extended. It can be used now not only to support attribute computations in graph transformation systems but also in computer-assisted verification and proof-development.

The next 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. Afterwards we define our category of attributed graphs, and then explain how to apply a rewrite rule by the computation of a weak pushout. Proofs are based on the ideas presented in (Boisvert et al., 2011a) but applied in more general setting. In section 5 we present examples. Section 6 contains an outline of future work. The paper is completed by an appendix that contains a brief description of the system of typed λ-calculus with inductive types used for presentation and examples as well as necessary notions of proof theory.

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 $L \overset{i}{\to} G$. 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 (Ehrig, 1978), (Rozenberg, 1997)) and single pushout (abbreviated SPo, mainly developped by Löwe (Löwe, 1993), (Rozenberg, 1997)). The main difference is that in DPo morphisms are total maps and in SPo morphisms are partial maps. This implies different forms of rules.

In the DPo approach a rule is defined by 3 graphs and 2 total morphisms: $L \overset{l}{\leftarrow} K \overset{r}{\to} R$. 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 $K \overset{d}{\to} D$ and $D \overset{l^*}{\to} G$ and then a pushout (the arrows $R \overset{i^*}{\to} H$ and $D \overset{r^*}{\to} H$). Cf Figure 1(a).

In the SPo approach, a rule is defined by one partial morphim $L \overset{r}{\to} R$. 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 $G \overset{r^*}{\to} H$ and $R \overset{i^*}{\to} H$). 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, see (Rebout et al., 2008), (Rebout et al., 2011), (Tran et al., 2010) and the use of λ-terms for attributes was encouraging but the construction of a double pushout still imposed some constraints due to the use of total maps and the obligation to split computation into two parts. The approach based on single pushout construction with λ-terms as attributes that we pursued afterwards (Boisvert et al., 2011a) was more direct and natural, free of application conditions and no specific constraints on the computational level. It permitted to strengthen attribute computations and lighten the structure rewrite. At the same time, while the relationship between λ-calculus and proof theory is well known (one may mention famous Curry-Howard isomorphism), the early version of our system could not be used directly in proof development and verification. In this paper, we generalize it in this direction.

3 CATEGORY OF ATTRIBUTED GRAPHS

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).

Let us recall that 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:

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

As a consequence, the existence of pushout implies the uniqueness of the object H up to isomorphism (cf. (Ehrig et al., 2006a), (Löwe, 1993)). If we have the two properties in the definition of pushout but not the unicity of c, the construction is called a weak pushout.

As in our previous work (Boisvert et al., 2011a), the system T of λ-calculus is used to define attributes and computation functions, but now we generalize both the notion of an attribute and of a computation function: instead of λ-terms in a fixed context, full typing judgements $\Gamma \vdash M : A$ with arbitrary context Γ and partial typing proofs are considered. The resulting category of attributed graphs will be denoted by Gr^{TP}. (For all notions that are not explained in the main part of our paper please see Appendix.)

To have better idea of the power of T, let us recall that it is a system of simply typed λ-calculus with surjective pairing, terminal object and inductive types. Type constructors include $\rightarrow$ for functional types, $\times$ for product (pairing) and *Ind* for inductive types. Pairing may be used to represent records, i.e., to "pack" multiple attributes into one. 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. Definition of inductive types includes structural recursion over each inductive type, this explains their particular interest in modeling computations.

Below Θ is the set of all typing judgements of T.

Objects. Objects of Gr^{TP} are attributed graphs. An attributed graph is defined as 5-tuple $G =< V_G, E_G, sr_G, tg_G, att_G >$, where the structural part (first 4 items) consists of the set of vertices V_G, the set of edges E_G and two functions source $sr_G : E_G \rightarrow V_G$ and target $tg : E_G \rightarrow V_G$ to connect edges to vertices. The elements of the set $V_G \cup E_G$ are called "elements of the graph". In this paper, we assume that $V_G \cup E_G$ is fully ordered (lexicographically)[2]. The function $att_G : V_G \cup E_G \rightarrow \Theta$ associates exactly the judgement $\Gamma \vdash M : A$ with each element of the graph.

To represent multiple attributes of a structural element, we use pairing to "pack" different data into one λ-term. So each attribute can be seen as an n-tuple containing all information attached to an element. The n-tuple $< M_1, ..., M_n >$ is considered as an abbreviation of the term $< ... < M_1, M_2 >, ..., M_n >$.

Certain inductive type(s) can be reserved to represent labels in ordinary sense. *E.g.*, let $F_n = \text{Ind}(\alpha)\{c_1 : \alpha | ... | c_n : \alpha\}$ be a finite type. In T, we have typing judgements $\Gamma \vdash c_i : F_n$ in any context Γ. If we want to use c_i in combination with other attribute $M : A$, we may use $< M, c_i >$ of type $A \times F_n$. The "absence of attributes" is represented by $0 : \top$.

Morphisms. Let G, H be two attributed graphs. A morphism $f : G \rightarrow H$ is defined in three parts:

1. The "structural part" noted f_{str} is a partial graph homomorphism (Rozenberg, 1997) from the stuctural part of G to the structural part of H (cf. Fig. 2).For each $v \in V_H \cup E_H$, its pre-image (*i.e.* the set of all its antecedents) is noted $[v]_{f_{str}} \subseteq V_G \cup E_G$.
2. The "attribute dependency relation" f_{adr} is a rela-

[2]In any case, it is close to ordinary practice when vertices are represented by natural numbers.

tion between the sets $V_G \cup E_G$ and $V_H \cup E_H$. For each $v \in V_H \cup E_H$, its pre-image is noted $[v]_{f_{adr}} \subseteq V_G \cup E_G$. Applying to its elements att_G, all attributes of graph G which are used to compute v can be obtained.

3. The "computational part" $f_{cmp}(v)$ is represented by a partial proof. The partial proofs $f_{cmp}(v)$ have to be "matched" with the attributes of G and H in the following sense.

   Let $v \in V_H \cup E_H$. Let $att(v) = \Gamma \vdash M : A$. Let $[v]_{f_{adr}} = \{u_1, ..., u_k\}, u_1 < ... < u_k$ (as before, we use the ordering on elements of G) and

   $$att(u_1) = \Gamma_1 \vdash M_1 : A_1, ..., att(u_k) = \Gamma_k \vdash M_k : A_k.$$

   The partial proof tree $p = f_{cmp}(v)$ should have exactly k active leaves with labels that are equal to the attributes $\Gamma_1 \vdash M_1 : A_1, ..., \Gamma_k \vdash M_k : A_k$ (in the order defined by the order of the leaves of the tree). The root of the tree should have a label that is equal to the attribute $att_H(v)$. Parameter leaves are not matched to anything. (See Fig. 2.)

Equality of Objects and Morphisms. For objects, we use identity on structural part and $\beta\eta\iota$-equality of judgements of T for attributes. For morphisms, $f = g$ requires the identity of f_{str} and g_{str}, f_{adr} and g_{adr}; for computation functions, for all v the equality of f_{cmp} and $g_{cmp}(v)$ w.r.t. $\beta\eta\iota$-equality is required (see definition 6.5 of the Appendix [3]).

Categorical Structure on Gr^{TP}. The identity id_G is defined using identity graph homomorphism as f_{str}, identity relation as f_{adr} and canonical identity partial proofs as $f_{cmp}(v)$. The composition of morphisms (only the level of partial proof trees is non-trivial) is defined using composition of partial proof trees, definition 6.6 of the Appendix.

Theorem 3.1. *Gr^{TP} defined above is a category.*

Proof. Composition is associative due to associativity of the composition of graph homomorphisms, and associativity of the composition of relations. For partial proofs composition is associative too because of confluence and the fact that T is 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 $Id_H \circ f = f$.

Remarks. This notion of equality is discussed in detail in (Boisvert et al., 2011a). Here we would like to remark that there is no reason to impose equivalence relation on partial proofs themselves since the system we describe is intended to study the properties of deductions, models etc. represented by attributed graphs. So it is more natural to impose an equivalence relation on attributes and attributed graphs as needed.

[3]We may obtain other interesting categorical structures with different kinds of equality of judgements and partial proofs, for example syntactic (graphical) equality of judgements.

4 RULE APPLICATION BY A WEAK PUSHOUT COMPUTATION

As in the SPo approach, in our approach each morphism $r : L \to R$ defines a transformation rule. The auxilliary notion of an embedding is necessary to indicate a "redex" - the part of the host graph to which the rule can be applied.

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

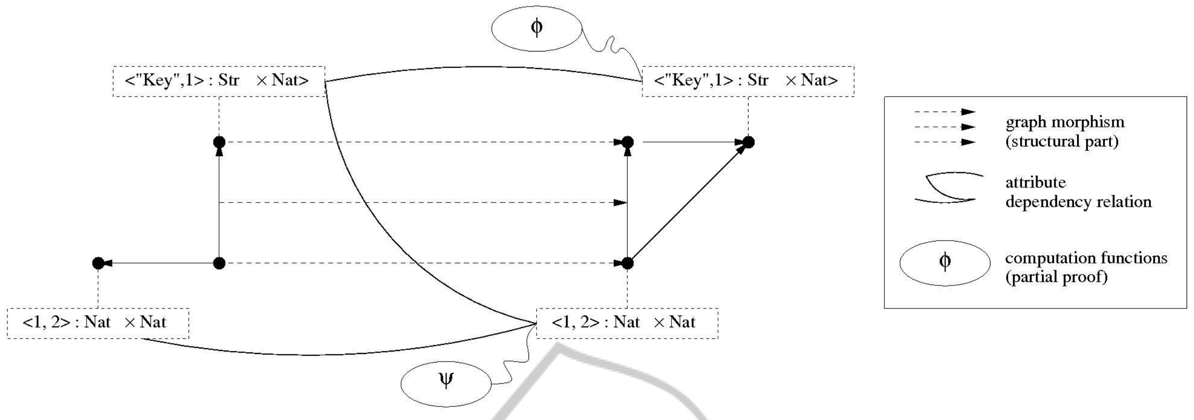
1. f_{str} is an injective partial graph homomorphism (i.e. $\forall v_1, v_2 \in V(G) \cup E(G).(f_{str}(v_1) = f_{str}(v_2) \Rightarrow v_1 = v_2)$);
2. $f_{adr} = f_{str}$;
3. for each $v' \in V(H) \cup E(H)$:
   - if $[v']_{f_{adr}}$ is empty, then $f_{cmp}(v')$ is the partial proof tree (cf. appendix) that has one node with the label $att(v')$. It is at the same time its root and its only leaf, which is not active.
   - if $[v']_{f_{adr}}$ is not empty (thus $[v']_{f_{adr}}$ is a singleton because f_{adr} is injective) then $f_{cmp}(v')$ is canonical identity partial proof for the attribute $att(v')$.

We shall call an *embedding* a total injective attributed graph morphism.

Canonical Retraction of an Embedding. Let $f : G \to H$ be an embedding. 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, *i.e.*, v' does not belong to the image of f_{str}, then v' does not belong to the domain of $\overline{f}_{str}(v')$; if $[v']_{f_{str}}$ is not empty then $[v']_{f_{str}} = \{v\}$ for some v because of injectivity and we pose $\overline{f}_{str}(v') = v$. Notice that $[v]_{\overline{f}_{str}} = \{v'\}$ by this definition.
2. $\overline{f}_{adr} = \overline{f}_{str}$
3. for each $v \in V(G) \cup E(G)$ $f_{cmp}(v)$ is canonical identity partial proof for the attribute $att(v)$.


ϕ
<"Key",1> : Str × Nat
<"Key",1> : Str × Nat
<1, 2> : Nat × Nat
<1, 2> : Nat × Nat
ψ
graph morphism (structural part)
attribute dependency relation
computation functions (partial proof)

$$\phi = \overset{*}{\vdash} <'' Key'', 1 >: Str \times Nat$$

$$M = \lambda x : Nat{\times}Nat.\lambda y : Str{\times}Nat. < p_1x, p_1x{+}p_2y >: Nat{\times}Nat \to Str{\times}Nat \to Nat{\times}Nat$$

$$\psi = \frac{\dfrac{\vdash M : Nat \times Nat \to Str \times Nat \to Nat \times Nat \quad \overset{*}{\vdash} < 1,2 >: Nat \times Nat}{M < 1,2 >: Str \times Nat \to Nat \times Nat} \quad \overset{*}{\vdash} <'' Key'', 1 >: Str \times Nat}{\vdash (M < 1,2 >) <'' Key'', 1 >: Nat \times Nat}$$

Figure 2: Attibuted graph morphism.

With this definition, it is easy to see that $\overline{f} \circ f = Id_G$.

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 (Rozenberg, 1997), 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 an embedding as definded above. We want to compute the weak pushout ($R \xrightarrow{i'} H$, $G \xrightarrow{r'} H$) of this pair.

The first step to define a pushout 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;

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

Figure 3: Construction of weak pushout.

- 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. (Rozenberg, 1997)) 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 $\{x_1,\ldots,x_k,z\}$ $(x_1,\ldots,x_k \in j'(i(L_0)), z \in j''(r(L))$;
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 (Rozenberg, 1997).

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 $h \circ \overline{i'}$ 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.

This is summarised in the following theorem.

Theorem 4.1. *Let $L \xrightarrow{r} R$ be an attributed graph morphism and $L \xrightarrow{i} G$ be an embedding in Gr^{TP}. There exists weak pushout of $L \xrightarrow{r} R$ and $L \xrightarrow{i} G$. Moreover, it may be assumed that the morphism i' in the (weak) pushout diagram is also an embedding.*

Composite Rules. The proof of the theorem above provides a canonical construction of the weak pushout, in particular the vertical arrow i' is an embedding like i. Such weak pushouts are sometimes called specific weak pushouts because in the proof we used the property that one of the two morphisms is an embedding.

This construction of specific weak pushout permits to compose the rules. Take two morphisms $L \xrightarrow{r} R$ and $R \xrightarrow{r'} R'$. We have also their composition $L \xrightarrow{r' \circ r} R'$. The fact that i' in the specific weak pushout above is also an embedding permits to construct the second weak pushout representing an application of the rule given by $R \xrightarrow{r'} R'$. It may be verified that the construction of specific weak pushout applied directly to $r' \circ r$ gives the same graph H' after transformation.

Possible Generalizations: Schematic Morphisms and Rule Schemas. The idea to use metavariables in the definitions of graph transformation rules is supported by the practice of proof theory. Various examples are possible, e.g., one may define the disjoint union of graphs using a rule schema (Boisvert et al., 2011b). We shall use below only a restricted case of rule schema based on the notion of schematic morphism which we shall define precisely. For definitions of partial proofs and schemas see Appendix, defintions 6.4 and 6.8.

Definition 4.1. *A schematic morphism is obtained if we replace partial proofs in the definition of morphism in Gr^{TP} by partial proof schemas. An instance of schematic morphism is any morphism in Gr^{TP} obtained by instantiation of metavariables. Let $r : L \to R$ be a schematic morphism. The rule schema in Gr^{TP}*

is the family of graph transformation rules defined by all instances of $r : L \to R$.

5 EXAMPLES

To illustrate our transformation approach we present in this section two detailed examples that may be of interest from the point of view of model transformations. The first one presents computation on attributes representing infinite trees. This can not be done using Σ-algebras. Possible applications include transformations of infinite models. The interest of dealing with infinite models is now taken into consideration (Combemale et al., 2012). A model can be infinite according to the width or the depth of the graph. In the example, we show how to deal with infinite width trees. Another example concerns coercive subtyping, that has important uses in software modeling and reuse (Soloviev and Luo, 2001).

Let us mention also some examples not developed in this paper that can be easily treated using our approach: (i) graph cloning, cf. (Boisvert et al., 2011b); (ii) generation and transformation of proofs in deductive systems, cf. (Boisvert et al., 2012) (e.g., Kleene-style premutations of rules (Kleene, 1952)); (iii) information transfer between attributes and structure; (iv) transformations of UML diagrams to relational models; (v) term graph rewriting (cf. (Barendregt et al., 1997)).

5.1 UML Diagram to Database Relational Model

Our approach permits to manage classical graph transformation problems like UML diagram to database relational model transformation. This problem is described in (). In (Taentzer et al., 2005) the transformation using the the DPo approach ... In our approach it is possible to manage this classical problem. In this paper we present only an example of rule because it would take several pages to present them all. The figure 4 presents the rule "Class2Table" (Taentzer et al., 2005). As we use typed λ-calculus, it necessary to define the types used in this example:

- the type *Type* is defined as a finite type (see appendix 6) representing all "classes" of objects manipulated by the transformation:

  $$Type = Ind(\alpha)\{Class : \alpha | Table : \alpha | Column : \alpha | \ldots\}$$[4]

- the type *Connector* is also defined as a finite type representing all temporary nodes that permit to connect elements of the class diagram to elements of the relational database model (see (Taentzer et al., 2005)):

  $$Connector = Ind(\alpha)\{C2T : \alpha | A2C : \alpha | A2F : \alpha | \ldots\}$$

- the type *Name* is the type describing the Name of the classes or tables. It is more or less like String.
- the type Bool is used as the value of the attribute *is_persistent* of a node of *Type Class*.[5]

We do not describe only one rule of this problem here because it would take several pages to write them all, and there is no complex attribute computation in these rules. Thus our approach has no advantage on other approach to manage this example. It is possible to define all the other rules presented in (Taentzer et al., 2005) in the same way.

5.2 Managing Infinity with Functional Attributes

The use of λ-terms as attributes permits to manage complex data structures that can represent infinity.

As an example, the type T_ω which represents trees with infinite branching [6] can be defined as follows:

$$\begin{aligned} T_\omega = Ind\alpha\{ & 0 : \alpha, \\ & S : \alpha \to \alpha, \\ & L : (Nat \to \alpha) \to \alpha\} \end{aligned}$$

Using the standard recursion operators on inductive types (see appendix), we can define complex ω-trees and computations on these infinite tree structures. The figure 5 presents a simple example of ω-tree defined by the term $L(Rec_{Nat \to T_\omega}(0)(\lambda x^{Nat} \lambda y^{T_\omega}.S(y)))$. It is also possible to define computations that transform these terms.

It is possible to write a function that takes as argument an infinite tree, and give as results the trees whith branches with pair numbers at every infinite branching:

$$Rec_{T_\omega \to T_\omega}(0)(\lambda x^{T_\omega}.S)(\lambda u.\lambda v.(v \circ d))$$

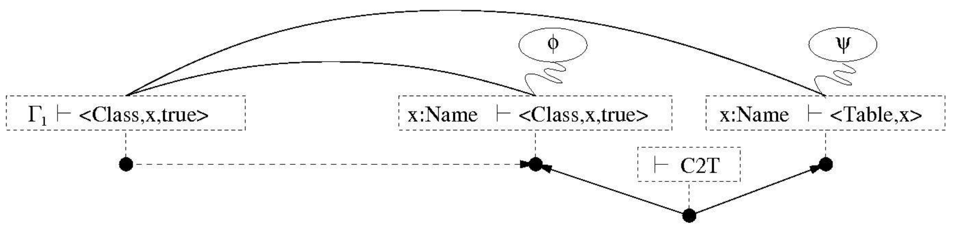
5.3 Coercive Subtyping

The notion of coercion was introduced to represent explicitly the transformation of the elements of the

[4]the constant Column and other constants are used in other rules that are not described in this paper

[5]In this example the attributes of a node have no "name", because they are stored in a tuple, and the position in the tuple permits to identify the different attributes. We do like this because we respect strictly on formalism, but in principle it would be possible to add names for the different elements of a tuple

[6]nodes have an infinite number of subtrees


ϕ
ψ
$\Gamma_1 \vdash$ <Class,x,true>
x:Name $\vdash$ <Class,x,true>
x:Name $\vdash$ <Table,x>
$\vdash$ C2T

$$\phi = x : Name \overset{*}{\vdash} < Class, x, true >: Type \times Name \times Bool$$

$$M = \lambda u : Name. < Table, (p_2(p_1 u)) >: Type \times Name \times Bool \to Type \times Name$$

$$\psi = \frac{x : Name \vdash M : Type \times Name \times Bool \to Type \times Name \qquad x : Name \overset{*}{\vdash} < Class, x, True >}{x : Name \vdash M < Class, x, True >: Type \times Name}$$

Figure 4: Example of a rule.

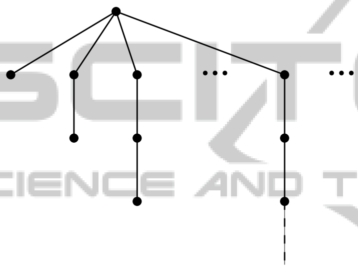


Figure 5: Example of ω-tree defined by the term $L(Rec_{Nat \to T_\omega}(0)(\lambda x^{Nat} \lambda y^{T_\omega}.S(y)))$. The length of the n-th branch is n.

subtype into the elements of the supertype. It is common knowledge that the representation of the elements of a datatype is often changed when we pass to a larger datatype, even if from mathematical point of view it is merely an inclusion.

In coercive subtyping the subtyping relation $A < B$ is interpreted as existence of a certain definable term $c : A \to B$, with motivation of giving operational semantics to calculi with subtyping and inheritance (see, e.g., (Breazu-Tannen et al., 1991)).

In practice, certain "basic coercions" are defined and other coercions are derived using appropriate rules. For example, to the transitivity of subtyping relation corresponds composition of coercions, from two subtyping relations $A < B$ and $C < D$ one can derive $B \to C < A \to D$, respectively, from the coercions $c_1 : A \to B$ and $c_2 : C \to D$ one may derive a coercion $c : (B \to C) \to (A \to D)$.

In the calculus with inductive types basic coercions are usually certain coercions between inductive types, for example the type $Bool = Ind(\alpha)\{T : \alpha | F : \alpha\}$ is the subtype of $Nat = Ind(\alpha)\{0 : \alpha | S : \alpha \to \alpha\}$ (with coercion $c(T) = S(0) : Nat, c(F) = 0 : Nat$).

The set of coercions may be represented by an acyclic attributed graph where attributes of the nodes represent corresponding inductive types. To do that we may use free variables, e.g., to represent the type A we take the axiom $x : A \vdash x : A$ as the attribute. Two nodes corresponding to the types A and B are connected by an arc if the types are in subtyping relation, and the attribute of this arc is the coercion $\vdash c : A \to B$ (coercion terms representing basic coercions have no free variables).

The set of coercions is coherent if composition of coercion terms along two paths with the same source and target is equal. The graph is completed to transitive closure (concerning the attributes, coherence permits to do it without contradiction). Practical usefulness of this is clear, since the coercions "implementing" the subtyping relation can be directly taken from the graph.

One of the main results obtained in (Soloviev and Luo, 2001) was that coherence of the set basic coercions implies coherence of the set of all derived coercions. The main consequence was that the subtyping extension of the consistent type theory without subtyping remains consistent.

Here we shall consider as an example two graph-rewriting rules (besides already mentioned transitivity) that may be used to extend already obtained coercion graph. They include the following derivations used to define new coercions.

$$d = \cfrac{\cfrac{\cfrac{x : A \overset{*}{\vdash} x : A}{x : A, z : C \vdash x : A}}{x : A \vdash \lambda z : C.x : C \to A}}{\vdash \lambda x : A.\lambda z : C.x : A \to (C \to A)}$$

$$d' = \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overset{*}{\vdash} c : A \to B}{z : C \vdash c : A \to B}}{f : C \to A, z : C \vdash c : A \to B} \qquad \cfrac{f : C \to A, z : C \vdash f : C \to A \quad f : C \to A, z : C \vdash z : C}{f : C \to A, z : C \vdash fz : A}}{f : C \to A, z : C \vdash c(fz) : B}}{f : C \to A \vdash \lambda z : C.(c(fz)) : C \to B}}{\vdash \lambda f : C \to A.\lambda z : C.(c(fz)) : (C \to A) \to (C \to B)}$$

Here in fact d is an ordinary derivation and d' is a partial derivation, active leaves (we refer to Ap-

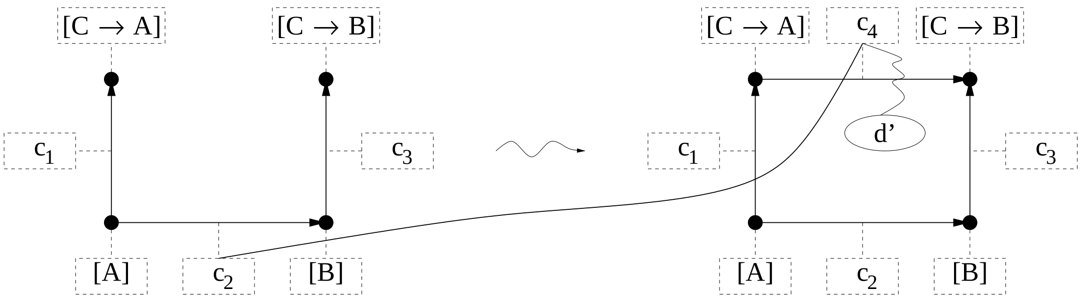


Figure 7: Rule 2.

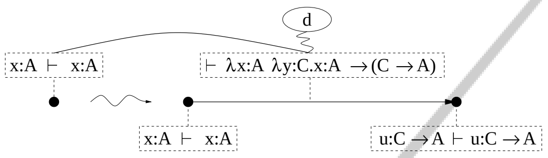


Figure 6: Rule 1.

pendix) are labeled by $*$. If A, B, C are considered as the metavariables for arbitrary types and c as a metavariable for arbitrary coercions, then we have schemas of (partial) derivations instead of concrete (partial) derivations. The instances will be obtained if we take, e.g., *Bool* instead of A, *Nat* instead of B and C, and concrete coercion $c : Bool \to Nat$ mentioned above.

Of course, other rules to introduce new coercions are possible, for example, a "contravariant" rule to pass from $A < B$ to $B \to C < A \to C$ and the rules for product types $A \wedge B$.

Below we give an example of graph transformations using d and d' to define computation functions.

The rules are given in Fig. 6 and 7. In Fig. 7, $[A]$ denotes $x : A \vdash x : A$, $[B]$ denotes $x' : B \vdash x' : B$ etc., c_1 denotes some coercion $\vdash c_1 : A \to B$, c_2 denotes $\vdash \lambda x : A.\lambda y : C.x : A \to (C \to A)$, c_3 denotes $\lambda.x' : B.\lambda y' : C.x' : B \to (C \to B)$, and c_4 denotes $\vdash \lambda f : C \to A.\lambda z : C.(c(fz)) : (C \to A) \to (C \to B)$. In Fig. 7 the left side may be assumed to be already obtained by applications of the first rule[7].

6 CONCLUSIONS

The aim of this paper was to generalize our previously defined graph transformation system . As in (Boisvert et al., 2011a), it is based on the SPo approach and its main originality concerns the use of a partial deductions to express attribute computations. Type theory incorporates both λ-calculus and reductions as computational mechanism, and has at the same time deduction rules for typing. The use of inductive types permits to include user-defined datatypes, as in many modern programming languages, and recursion over these types.

That gives to our approach a great expressive power that permit to manage classical model transformations like UML to Relational database model transformation, and also more sophisticated examples like computation on functions, on infinite data structures, or on proofs.

At the same time, the presence of proofs permits to establish connection between development and transformation of software models and software certification and verification.

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. In comparison with our previous papers, we presented a more powerful way to describe transformation of attributes, using not only computation with λ-terms but deduction rules.

The possible domains of applications include all usual applications of graph transformations, e.g., verification and model transformations in software engineering. Note that thanks to the approach described above it is now possible to deal with certain infinite models. More "tight" relationship between computation, graph structure and proofs will permit also the pursuit of much more specific goals, in particular in the domain of computer-assisted reasoning and verification (Luo, 1994),(Soloviev and Luo, 2001).

As a principal example of deductive system based on type theory we considered in this paper the simply typed λ-calculus with inductive types and pairing. All the constructions, though, can be easily modified to be used in case of higher order and dependent type systems in proof assistants, as well as for purely logical systems and applications.

A former experiment (Tran et al., 2010) of implementation in Haskell language constitutes the basis for building a sofware environment devoted to model

[7]The labels added using pairing may be used to avoid repeated application of transformation rules to the same arguments.

transformations using our new approach. This is a natural practical extension of our current work.

REFERENCES

Baar, T., Strohmeier, A., Moreira, A. M. D., and Mellor, S. J., editors (2004). *UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004. Proceedings*, volume 3273 of *LNCS*. Springer.

Barendregt, H., van Eekelen, M., Glauert, J., Kennaway, J., Plasmeijer, M., and Sleep, M. (1997). Term graph rewriting. *PARLE Parallel Architectures and Languages Europe*, pages 141–158.

Bézivin, J., Rumpe, B., Schürr, A., and Tratt, L. (2005). Model transformations in practice workshop. In *MoDELS Satellite Events*, pages 120–127.

Boisvert, B., Féraud, L., and Soloviev, S. (2011a). Typed lambda-terms in categorical attributed graph rewriting. In *2nd WorKshop on Algebraic Methods in Model-Based Software Engineering TOOLS 2011, June 30th, 2011, Zurich, Switzerland* . Electronic Proceedings in Theoretical Computer Science.

Boisvert, B., Féraud, L., and Soloviev, S. (2011b). Typed lambda-terms in categorical graph rewriting. In *The International Conference Polynomial Computer Algebra, April 18-22, Saint-Petersburg, Russia, Euler International Mathematical Institute*.

Boisvert, B., Féraud, L., and Soloviev, S. (2012). Graph Transformations, Proofs, and Grammars. In *Int. Conf. Phylosophy, Mathematics, Linguistics, Aspects of Interaction, May 22-25, Saint-Petersburg, Russia, Euler International Mathematical Institute*.

Breazu-Tannen, V., Coquand, T., Gunter, C., and Scedrov, A. (1991). Inheritance and implicit coercion. *Information and Computation*, 93:172–221.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In Luck, E. and Overbeek, R., editors, *Proceedings of the 9th International Conference on Automated Deduction (CADE)*, number 310 in LNCS, pages 111–120. Springer, Argonne.

Chemouil, D. (2005). Isomorphisms of simple inductive types through extensional rewriting. *Math. Structures in Computer Science*, 15(5):875–917.

Combemale, B., Thirioux, X., and Baudry, B. (2012). Formally Defining and Iterating Infinite Models. In France, R., Kazmeier, J., Atkinson, C., and Breu, R., editors, *Proceedings of the 15th international conference on Model driven engineering languages and systems (MODELS'12)*, volume 7590 of *LNCS*, pages 119–133, Innsbruck, Austria. Springer.

Diestel, R. (2010). *Graph Theory*. Springer-Verlag, fourth edition.

Ehrig, H. (1978). Introduction to the algebraic theory of graph grammars (a survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, pages 1–69.

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

Ehrig, H., Padberg, J., Prange, U., and Habel, A. (2006b). Adhesive high-level replacement systems: A new categorical framework for graph transformation. *Fundam. Inf.*, 74(1):1–29.

Gentzen, G. (1934-35). Untersuchungen über das logische Schliessen. In *I, II, Math. Z. 39*, pages 176–210, 405–443.

Kleene, S. C. (1952). Permutability of inferences in Gentzen's calculi LK and LJ. *Mem. Amer. Math. Soc.*, pages 1–26.

Löwe, M., editor (1993). *Algebraic approach to single pushout graph transformation, TCS*, volume 109.

Luo, Z. (1994). *Computation and Reasoning: A Type Theory for Computer Science*. International Series of Monographs on Computer Science. Oxford University Press, USA.

Luo, Z. (2008). Coercions in a polymorphic type system. *Math. Structures in Computer Science*, 18(4):729–751.

Orejas, F. (2011). Symbolic graphs for attributed graph constraints. *J. Symb. Comput.*, 46:294–315.

Rebout, M. (2008). *Une approche catégorique unifiée pour la récriture de graphes attribués*. PhD thesis, Université Paul Sabatier, Toulouse, France.

Rebout, M., Féraud, L., Marie-Magdeleine, L., and Soloviev, S. (2011). Computations in Graph Rewriting: Inductive types and Pullbacks in DPO Approach. In Szmuc, T., Szpyrka, M., and Zendulka, J., editors, *Advances in Software Engineering Techniques, CEE-SET 2009, Krakow, Poland, October 2009*, volume 7054 of *LNCS*, pages 150–163. Springer-Verlag.

Rebout, M., Féraud, L., and Soloviev, S. (2008). A Unified Categorical Approach for Attributed Graph Rewriting. In Hirsch, E. and Razborov, A., editors, *International Computer Science Symposium in Russia (CSR 2008), Moscou 07/06/2008-12/06/2008*, volume 5010 of *LNCS*, pages 398–410. Springer-Verlag.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific.

Soloviev, S. and Luo, Z. (2001). Coercion completion and conservativity in coercive subtyping. *Annals of Pure and Applied Logic*, 113–1:297–322.

Taentzer, G., Ehrig, K., Guerra, E., Lara, J. D., Levendovszky, T., Prange, U., Varro, D., and et al. (2005). Model transformations by graph transformations: A comparative study. In *Model Transformations in Practice Workshop at Models 2005, MONTEGO*, page 5.

Tran, H. N., Percebois, C., Abou Dib, A., Féraud, L., and Soloviev, S. (2010). Attribute Computations in the DPoPb Graph Transformation Engine (regular paper). In *GRABATS 2010, University of Twente, Enschede, The Netherlands, 28/09/2010-28/09/2010*, page (electronic medium), http://www.utwente.nl/en. University of Twente.

APPENDIX

Typed λ-Calculus with Inductive Types

The system of λ-calculus in this paper is the simply typed λ-calculus with surjective pairing, terminal object and inductive types. For details see (Chemouil, 2005), here we recall the principal definitions concerning this system that will be named T in the other sections of this paper.

Definition 6.1. *Types are either atomic types or defined by using a type constructor.*

Atomic types are:

- *the constant type* $\top$*;*
- *a finite or infinite set* $\mathcal{S} = \{\alpha, \beta, \ldots\}$ *of* type variables*;*

Type constructors are:

- $\rightarrow$ *for functional types, which constructs* $A \rightarrow B$ *for any types A and B*
- $\times$ *for product types, which constructs* $A \times B$ *for any types A and B*
- *Ind, defined as follows: let* $\mathcal{C}$ *be an infinite set of* introduction operators *(constructors of* elements *of inductive types), with* $\mathcal{C} \cap \mathcal{S} = \varnothing$*. an inductive type with n constructors* $c_1, \ldots, c_n \in \mathcal{C}$*, each of them having the arity* k_i *(with* $1 \leq i \leq n$*), has the form:*

$$\begin{aligned} \mathrm{Ind}(\alpha)\{c_1 : A_1^1 \rightarrow \ldots \rightarrow A_1^{k_1} \rightarrow \alpha; \\ \ldots; \\ c_n : A_n^1 \rightarrow \ldots \rightarrow A_n^{k_n} \rightarrow \alpha\}, \end{aligned}$$

Here, every $A \equiv A_i^1 \rightarrow \ldots \rightarrow A_i^{k_i} \rightarrow \alpha$ *is an inductive schema, i.e.,* A_i^j *is:*

- *either a type not containing* α*;*
- *or a type of the form* $A_i^j \equiv C_1 \rightarrow \ldots \rightarrow C_m \rightarrow \alpha$*, where* α *does not appear in any* $C_{\ell \in 1..m}$ *(such* A_i^j *are called* strictly positive operators*).*

Here $Ind(\alpha)$ binds *the variable* α*.*

Example 6.1. *(Definition of types Bool, Nat and* T_ω*, the type of* ω*-trees.)*

$$\begin{aligned} F_n &= \mathrm{Ind}(\alpha)\{c_1 : \alpha | \ldots | c_n : \alpha\} \\ Bool &= \mathrm{Ind}(\alpha)\{T : \alpha | F : \alpha\} \\ Nat &= \mathrm{Ind}(\alpha)\{0 : \alpha | \mathrm{S} : \alpha \rightarrow \alpha\} \\ T_\omega &= \mathrm{Ind}(\alpha)\{0_\omega : \alpha | \mathrm{S} : \alpha \rightarrow \alpha | \mathrm{L}_\omega : (Nat \rightarrow \alpha) \rightarrow \alpha\}. \end{aligned}$$

Definition 6.2. *Let* $\mathcal{V}$ *be an infinite set of variables* $\mathcal{V}$ *(with* $\mathcal{V} \cap \mathcal{S} \cap \mathcal{C} = \varnothing$*). The set of* λ-terms *is generated by the following grammar rules:*

$$M \ ::= c \,|\, \mathcal{R}_{B,D} \,|\, x \,|\, (\lambda x : B \cdot M) \,|\, (M\, M) \,|\, < M\, M >$$

where $x \in \mathcal{V}$*,* $c \in \mathcal{C}$*, B and D are arbitraty types, and* $\mathcal{R}_{B,D}$ *is the standard recursion operator (for details, see (Luo, 1994), (Chemouil, 2005)).*

All terms and types are considered up to α-conversion, *i.e.*, renaming of bound variables. Context Γ is a set of term variables with types $x_1 : A_1, \ldots, x_n : A_n$ ($x_1, \ldots, x_n$ distinct). Γ, Δ denotes the union of contexts Γ, Δ (we assume that Γ, Δ have no common term variables). The expression $\Gamma \vdash M : A$ is called typing judgement (or sequent). Its meaning is "term M has type A in the context Γ".

Definition 6.3. *Here are the following typing axioms and rules for the terms defined above (*A, B, D *denote arbitrary types,* Γ *arbitrary context).*

Axioms.

- $\Gamma, x : A \vdash x : A,\ \Gamma \vdash 0 : \top;$
- *For each inductive type* $C = \mathrm{Ind}(\alpha)\{c_1 : A_1 | \ldots | c_n : A_n\}$ *and* $1 \leq i \leq n$
$$\Gamma \vdash c_i : A_i[B/\alpha]$$
(for example, if $C = Nat$*, we shall have* $\Gamma \vdash 0 : Nat$ *and* $\Gamma \vdash S : Nat \rightarrow Nat$*);*
- *For C as above and any type D the axiom*[8]*:*
$$\Gamma \vdash \mathcal{R}_{C,D} : \Upsilon_C(A_1, D) \rightarrow \ldots \rightarrow \Upsilon_C(A_n, D) \rightarrow C \rightarrow D.$$

Typing Rules.

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash < M\, N >: A \times B}(pair)$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash p_1 M : A}(p_1) \quad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash p_2 M : B}(p_2)$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A \cdot M) : A \rightarrow B}(\lambda)$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\, N) : B}(app)$$

Remark 6.1. *(i) The constant* $\mathcal{R}_{C,D}$ *is called the* recursor *from C to D. Notice that applying it (using the rule app) to the terms* $M_1 : \Upsilon_C(A_1, D), \ldots, M_n : \Upsilon_C(A_n, D)$ *we define the function* $\mathcal{R}_{C,D} M_1 \ldots M_n : C \rightarrow D$*. The following derived rule is often included:*

$$\frac{\Gamma \vdash M_i : \Upsilon_C(A_i, D)(1 \leq i \leq n)}{\Gamma \vdash (\mathcal{R}_{C,D}\, M_1 \ldots M_n) : C \rightarrow D}(elim)$$

(ii) Usually the following **structural rules** *are included in T (they are admissible w.r.t. other rules):*

$$\frac{\Gamma \vdash M : B}{\Gamma, x : A \vdash M : B}(wkn) \quad \frac{\Gamma, x : A, x' : A \vdash M : B}{\Gamma, x : A \vdash [x/x']M : B}(contr)$$

[8] $\Upsilon_C(A, D)$ are certain auxilliary types used to define recursion from C to D. They correspond to the types of functions that appear in standard recursive equations over C. For example, if $C = D = Nat$ and $A = Nat \rightarrow Nat$ (the type of successor S), then $\Upsilon_{Nat}(A, Nat) = Nat \rightarrow Nat \rightarrow Nat$.

$$\frac{\Gamma \vdash N : A \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash [N/x]M : B}(subst)$$

Here $[N/x]$ denotes substitution with renaming of bound variables to avoid capture.

Normalization and Equality. The terms of the system T are considered up to equality generated by conversion relation. The α-conversion (renaming of bound variables) was already mentioned. Other conversions are[9] : (i) β-conversion $(\lambda x : A.M)N = [N/x]M$; (ii) η-conversion $\lambda x : A.(Mx) = M$ (where x must not be free in M); (iii) and ι conversion for recursion. The ι-conversion corresponds to one step in recursive computation. It applies to the terms of the form $(\mathcal{R}_{C,D}M_1...M_n)(c_iN)$, i.e. when the function defined by recursion applied to the term beginning by one of the introduction operators. For example, for $(\mathcal{R}_{Nat,Nat}ag)0 \rightarrow_\iota a$ and $(\mathcal{R}_{Nat,Nat}ag)(Sn) \rightarrow_\iota (\mathcal{R}_{Nat,Nat}ag)((gn)(Sn))$ (here $a : Nat$ is "initial value" and $g : Nat \rightarrow Nat \rightarrow Nat$ defines inductive step. The exact general definition may be found in (Chemouil, 2005), p.884.

T is confluent and strongly normalizing with respect to $\beta\eta\iota$-reductions (directed conversions). Detailed description and normalization theorems for T can be found in (Chemouil, 2005). Thus, the equivalence relation on terms based on conversion (often called $\beta\eta\iota$-equality) is decidable.

Proof Trees and Partial Proofs

An inference rule in proof theory is a couple $\dfrac{P}{C}$ where P is a list of premises, possible subject to some constraints. As examples one may take the rules of the system T above. Usually in proof theory the presentation of rules is *schematic*, that is, the metavariables like Γ, Δ are used to represent arbitrary contexts, A, B to represent arbitrary types *etc.* The presentation below is generic, *i.e.*, all the definitions can be modified to accomodate a change of logical system, if only the system has tree-form derivations build by application of deduction rules to their premises.

Trees are a special case of graphs, and proof trees are a special case of attributed graphs, but in any case the trees below should be considered as part of metatheory and not the objects of the category of attributed graphs defined in this paper. Applications of trees to computations and data structures are usually straightforward, in difference from graphs in general[10]. Our generalization of the definition of graph transformation systems is based on the notion of partial proof. Below the reader may assume that the partial proofs are taken in the system T described above but the definition will apply to any other deductive system with appropriately defined derivable objects.

According to standard definitions (cf. (Diestel, 2010)), a tree is a connected directed acyclic graph $J = (V, E)$ in which a single node is designated as root and there is a unique path from the root to any other node. If $(x, y) \in V$, we say that y is a child of x and y is the parent of x. A leaf has no children. Since J has no directed cycles, the transitive closure E^* of E defines a partial strict order on V. There is a path from x to y in J iff $(x, y) \in E^*$.

An ordered tree is the tree where outgoing edges of any $v \in V$ are numbered $0, 1, \ldots$. Thus, to any $(x, y) \in E^*$ corresponds a unique sequence of natural numbers. The lexicographic ordering of these sequences beginning at the root r permits to extend this partial order to the unique linear order on V. In particular there is the natural ordering of the leaves.

The definitions below are modified definitions from (Bundy, 1988) adapted to our case.

Definition 6.4. *A partial proof is an ordered tree with the following properties: (i) each node is labelled with a sequent and the rule of inference which is applied to this sequent (backwards) to produce the (labels of) the node's children; (ii) the final sequent (or the goal) is the sequent at the root of the tree; (iii) for the leaves, no rule of inference is specified; (iv) we shall further distinguish "active" and "parameter" leaves. Active leaves are marked by $*$.*

Our purpose is to use partial proofs for attribute transformations, and thus, the difference between axioms and other sequents is not relevant. In some cases we may impose additional restrictions, e.g., that the parameter leaves are axioms or that all leaves have *derivable* sequents as labels.

The use of partial proofs instead of λ-terms to represent computation functions permits more flexibility concerning the choice of equality in the category of graph transformations. The definitions and results below will be valid for any equality (equivalence relation) on the set of sequents (logical formulas, judgements). Usually (but not necessarily) the judgements of the system T are considered up to $\beta\eta\iota$-equality. Another choice may be syntactic (graphic) equality of terms and types (with γ and Γ' equal as sets).

Next notion we are going to define is the equality of partial proofs. In fact, the main requirement is that the good properties of composition must be assured.

[9]We omit the contexts and types of terms.

[10]The possibility to "embed" them into this category seems obvious, but to our opinion it may be considered as an invitation to study a hierarchy of graphs and attributes with alternating layers. The possibility seems interesting but it is out of scope of this paper.

Definition 6.5. *Partial proofs are equal if they have the same number of active leaves, the labels of corresponding active leaves (in order defined by the trees) and the labels of two roots are equal* [11].

The notion of composition of partial proofs is inspired by the notion of composition of multivariable functions.

Definition 6.6. *Let $l_1 < ... < l_k$ be all active leaves (see definition above) of the partial proof P. Let P_1,..., P_k be partial proofs and $r_1, ..., r_k$ their roots. Let for all i, $1 \leq i \leq k$ the labels of l_k and r_k be equal. The composition $P * (P_1, ..., P_k)$ is obtained by identification of each l_i with its label and r_i with its label (assuming other nodes disjoint). Order relations are extended to the new tree in natural way. The active leaves are now the union of the active leaves of $P_1, ..., P_k$ and the root is the root of P.*

The result is another partial proof. This composition is associative w.r.t. the equality defined above.

Definition 6.7. *The canonical identity partial proof for the sequent (formula, judgement) S is the tree with one node (which is the root and the one active leaf at the same time) that has S as its label.*

Schemas of Partial Proofs

It is common in proof theory to use axiom and rule schemas instead of individual axioms and rules. In the schemas the meta-variables may be used. The formulations of axioms and rules of the system T above are schematic. There may be metavariables of different kinds, *e.g.*, metavariables for terms, contexts, and even for arbitrary variables as in the axiom schemas or the rule (λ) above[12].

Definition 6.8. *(Partial Proof Schema.) A partial proof schema is an ordered tree with the following properties: (i) each node is labelled with a meta-level sequent. (ii) each node except the leaves is labeled also with the rule of inference which is applied to this sequent (backwards) to produce the node's children (and the children of course must be the meta-level sequents matching the premises of this rule). (iii) the final meta-level sequent (or the goal) is the meta-level sequent at the root of the tree, some of the leaves are marked as active by $*$.*

[11] An alternative definition would be to require the equality of the whole ordered trees and *all* corresponding labels.

[12] Essentially, this practice is similar to the use of nonterminals in the formal grammars.