Electronic Communications of the EASST
Volume 32 (2010)

Proceedings of the
Fourth International Workshop on
Graph-Based Tools
(GraBaTs 2010)

Attribute Computations in the DPoPb Graph Transformation Engine

Hanh Nhi Tran, Christian Percebois, Ali Abou Dib, Louis Féraud, Sergei Soloviev
IRIT, University of Toulouse
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
{tran, percebois, aboudib, feraud, soloviev}@irit.fr

14 pages

Guest Editors: Juan de Lara, Daniel Varro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Attribute Computations in the DPoPb Graph Transformation Engine

Hanh Nhi Tran, Christian Percebois, Ali Abou Dib, Louis Féraud, Sergei Soloviev

IRIT, University of Toulouse
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
{tran, percebois, aboudib, feraud, soloviev}@irit.fr

Abstract: One of the challenges of attributed graph rewriting systems concerns the implementation of attribute computations. Most of the existing systems adopt the standard algebraic approach where graphs are attributed using sigma-algebras. However, 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. In previous works we introduced the Double Pushout Pullback (DPoPb) framework which integrates attributed graph rewriting and computation on attributes in a unified categorical approach. This paper discusses the DPoPb's theoretical and practical advantages when using inductive types and lambda-calculus. We also present an implementation of the DPoPb system in the Haskell language which thoroughly covers the semantics of this graph rewriting system.

Keywords: attributed graph rewriting, attribute computation, algebraic graph transformation, Haskell language.

1 Introduction

The last decade shows a great interest in graph rewriting in MDE [15] as a model transformation technique. For these applications, it is important to use attributed graphs. There are several works on attributed graph transformations (see *e.g.* [11][12][13][3][8]) mostly based on the algebraic data types to implement attribute computations. In the algebraic approach, attributes are given as values within some Σ-algebras. Therefore, information is directly integrated in the graph structure by creating a new “attribute node” for each value of an algebraic sort. This approach is theoretically sound but shows some limits on the expressiveness of attribute computation and is especially difficult to be completely implemented. Thus most of the existing systems resting on the standard algebraic approach hardly respect the theoretical foundation. Consequently, the attributed graphs rewriting in these systems is validated theoretically but not practically.

In [5] and [6] we introduced DPoPb, a unified categorical model of attributed graph transformations using inductive types for attribute values and lambda-terms for computations. Keeping the same conceptual scheme as in the DPO constructions, the goal of DPoPb is to put the attribute computation to work in a more uniform way by staying within the same theory for implementing computations. We argue in this paper that the inductive types and lambda-calculus make attribute computations in DPoPb more expressive than in the DPO-standard model [1] and facilitate the implementation of attributed graph rewriting system. This claim is justified by an implementation of DPoPb engine in Haskell [2] which conforms entirely the theoretical model hence allows validating theoretically and practically the DPoPb attributed graphs rewriting.

The paper is organized as follows. In Section 2 we analyse the differences between the theoretical solutions for attribute computations in the DPoPb and the standard algebraic approach represented by the HLR framework [3]. Section 3 discusses the implementation of HLR and DPoPb graph rewriting systems. In Section 4, we sketch out the development of the DPoPb prototype in Haskell to validate the theoretical foundation. Finally, in the last section we discuss some current and future works to improve the DPoPb approach.

2 Attribute computations in the DPO and DPoPb approaches

In this section we compare the solutions for attribute computations in the HLR approach [3] with the DPoPb's one. We first outline attribution computations in each approach, and then we use an example to show their differences.

2.1 Attribute Computation in HLR framework

HLR framework [3] is representative for the algebraic approach attributed graph rewriting. This work is now very well-known so we just give an outline of the approach for attribute computation.

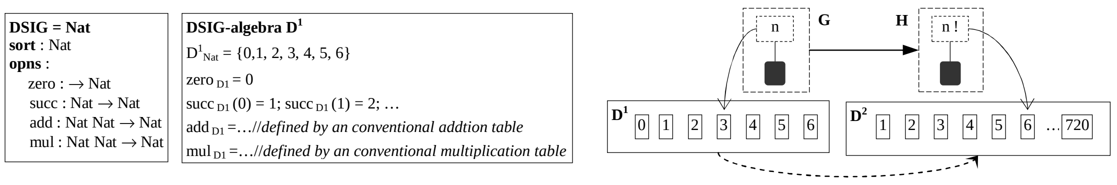
In order to model attributed graphs with attributes for nodes and edges, HLR extend the classical notion of graphs to E-graphs. An E-graph G has two different kinds of nodes, implementing the graph and data nodes, and three kinds of edges, the usual graph edges and special edges used for the node and edge attribution. An E-graph morphism f_G is defined then as a classical graph morphism. Let $DSIG = (S_D,OP_D)$ be a data signature with attribute value sorts $S'_D \in S_D$. An attributed graph $AG = (G,D)$ consists of an E-graph G together with a DSIG-algebra D such that $_{s\in S'_D} D_s = V_D$ where V_D is the set of data nodes of graph G. For two attributed graphs $AG^1 = (G^1,D^1)$ and $AG^2 = (G^2,D^2)$, an attributed graph morphism $f : AG^1 \rightarrow AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \rightarrow G^2$ and an algebra homomorphism $f_D : D^1 \rightarrow D^2$ such that the square in Fig. 1 commutes for all $s \in S'_D$, where the vertical arrows are inclusions.

$$\begin{tikzcd}
\mathrm{D_G}^1 \arrow[r, "f_{D,s}"] \arrow[d] & \mathrm{D_G}^2 \arrow[d] \\
\mathrm{V_D}^1 \arrow[r, "f_{G,V_D}"] & \mathrm{V_D}^2
\end{tikzcd}$$

Fig. 1. The algebra homomorphism f_D in the attributed graph morphism f: AG$^1 \rightarrow$ AG^2

Given a data signature DSIG as depicted, attributed graphs and attributed graph morphisms form the category *AGraphs* and graph rewrites can be realized by constructing the pushout of the category using the double pushout or simple pushout approach. In this section we consider the double pushout approach in HLR. A transformation rule $p: L \leftarrow K \rightarrow R$ is given by three attributed graphs (with variables) K, L, and R and two morphisms $l: K \rightarrow L$ and $r: K \rightarrow R$ which have to be injective on the graph structure and isomorphic over the Σ-algebra. In order to describe computations on the attributes, terms containing variables are to be used; *e.g.*, in the graph R, an attribute $x + y$ can be found if in the graph K the variables x and y are present.

To show how to combine graph transformations with computations on attributes in the HLR framework, we rest on an example for computing the value of $n!$. In this example, we use the signature *Nat* which defines the operators *zero*, *succ*, *add* and *mul* on the sort *Nat*. We can define two algebras D^1 and D^2 associated to the signature *Nat* to give two different semantics for *Nat*. For the illustration purpose, in this example, we use D^1 and D^2 which are different only on the carrier sets as shown in Fig. 2b. We can define then the attributed graph type $AG^1 = (G,D^1)$ to represent the graphs which can be attributed by the values from 0 to 6 and the attributed graph type $AG^2 = (G,D^2)$ to represent the graphs attributed by the values from 0 to 720.



DSIG = Nat
sort : Nat
opns :
zero : $\rightarrow$ Nat
succ : Nat $\rightarrow$ Nat
add : Nat Nat $\rightarrow$ Nat
mul : Nat Nat $\rightarrow$ Nat

DSIG-algebra D^1
$D^1_{Nat} = \{0,1, 2, 3, 4, 5, 6\}$
zero $_{D1}$ = 0
succ $_{D1}$ (0) = 1; succ $_{D1}$ (1) = 2; …
add $_{D1}$ =…*//defined by an conventional addtion table*
mul $_{D1}$ =…*//defined by an conventional multiplication table*

(a) Data signature and the algebra defining the semantics of the signature (b) Request Transformation to compute *n!*

Fig. 2. Calculating *n!* by graph rewriting and attribute computations in HLR

Given a value $n \leq 6$, let us suppose we compute the value of *n!* by graph rewriting in the system supporting the attributed graphs defined with the above *Nat* signature. We use the graph *G* of type AG^1 composed of one node having the attribute $n \in D^1_{Nat}$. The computation's result will be stored in the graph *H* of type AG^2 having also one node attributed by $n \in D^2_{Nat}$. Fig. 2b describes the needed transformation from *G* to *H*. However, with the signature *Nat* we cannot realize such a transformation because the factorial operator *!* is not defined. Consequently, we must decompose the computation of *n!* into many transformation steps, using the four rules represented in Fig. 3.

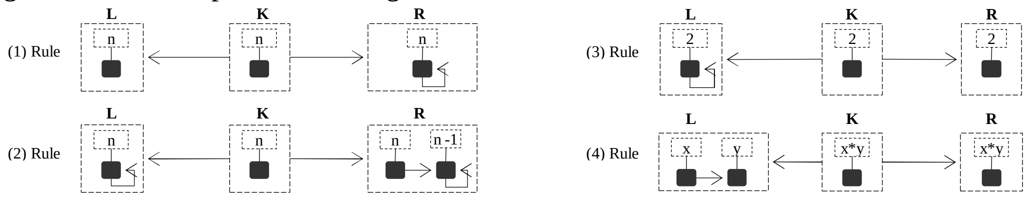


Fig. 3. Rules used for calculating *n!* in HLR

The first rule is applied once on the initial graph to prepare the list of number analyzing *n!*. Rule 2 is for delegating the computation of *n-1!* to a new node. This rule is applied until number 2 is reached. When the rule 2 is completed, a chain of nodes is created with the final node in still self-referred. To stop the delegation, the rule 3 is applied once to obtain a simple chain of number from *n* to 2. Rule 4 now is applicable: it takes the numbers of the last two nodes and multiplies them, then stores the result in the first of these two nodes, and deletes the very last one. Applying rule 4 as long as possible to obtain the result represented by only one node left, containing the computed result for *n!*. We can see that the computation of *n!* in this example with HLR system requires four transformation rules and *2n-3* steps.

For each application of a rule, the attributes of graph *L* must be updated to produce the graph *R*. In the literature, two main different approaches have been defined in order to specify attribute-value changes: relabeling attribute-nodes [11][26] and reconnecting attribute-value nodes [12][8]. In the relabeling issue, no built-in data types on labels are encoded and programs are in general based on rule scheme labelled with terms over algebras. On the opposite, in the reconnecting mode, a new edge is added each time the graph must reference a new attribute value. The example in Fig. 2b is illustrated with reconnecting scheme.

2.2 Attribute computation in the DPoPb approach

The DPoPb framework relies on the classical DPO approach for the structural part and uses type theory with inductive types for attribute computations. The precise definition of attributed graphs in the DPoPb approach can be found in [4][5][6]. Below we will give the essential information necessary to explain how attributes are implemented and computed in the DPoPb framework.

2.2.1 DPoPb Attributed Graphs and Attributed Graph Morphisms

In the DPoPb approach, an attributed graph is defined with two parts: a graph structure composed of labeled nodes and edges and a set of attributes associated to edges or nodes. DPoPb uses type theory to code attributed graphs: finite types to describe the structure of graphs and general inductive types to define data types and computations.

A morphism between two attributed graphs G and H is a 3-level structure morphism $f\colon G \rightarrow H$ defined by two components: the first component specifies the structural part of the morphism and the second one represents the attribute part of the morphism.

– The structural part, denoted f_S, a graph morphism from G to H, is the first level.
– The attribute part has two levels.
  - A relational level, denoted f_R. It includes the multirelation R between the attributes of H and G. For each attribute b of type B of the vertex s of G, a partition of the set $R_{\{s,b\}}$ of the H's attributes necessary to compute b. Each element of the partition (a subset of $R_{\{s,b\}}$) together with b defines a tree, and the set of all trees of f_R is called the forest of the morphism.
  - To each tree described above is associated a computation function represented by a lambda-term t in a way such that if the leaves of a tree are the attributes $a_1,\dots, a_n$ of the types $A_1,\dots, A_n$ respectively and its root is the attribute b of type B, then the type of the term is $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$. The conditions to be satisfied is that $t(a_1,\dots, a_n) = b$.

Usually the equality is the ordinary reduction-based equality of lambda-terms. Two morphisms $f, g\colon G \rightarrow H$ are considered as equal if all components on all levels are equal.

Fig. 4 shows an example of the DPoPb formalism representing attributed graph and attributed graph morphisms. In this example, we have a forest R composed of three trees T_1, T_2 and T_3 in which T_1 = ({7,8}, 15, $\lambda x\ y\ .\ x + y$), T_2= ({"Good"}, 4, $\lambda s\ .\ length\ s$) and T_3 = ({"Good","Luck"}, "GoodLuck", $\lambda s1\ s2\ .\ s1 ++ s2$).

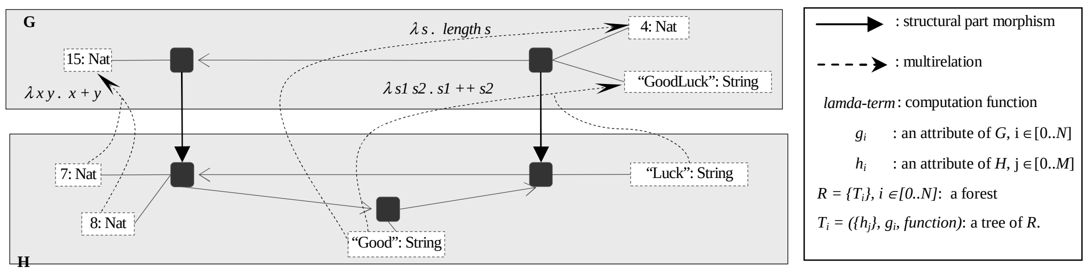

G
H
15: Nat
4: Nat
"GoodLuck": String
7: Nat
8: Nat
"Good": String
"Luck": String
λ x y . x + y
λ s . length s
λ s1 s2 . s1 ++ s2
: structural part morphism
: multirelation
lamda-term : computation function
g_i : an attribute of G, i ∈ [0..N]
h_i : an attribute of H, j ∈ [0..M]
R = {T_i}, i ∈ [0..N]: a forest
T_i = ({h_j}, g_i, function): a tree of R.

Fig. 4 An attributed morphism having a computation forest composed of three trees

The above definition of morphisms in DPoPb requires some comments on the reverse direction for the attribute relations and the role of partitions and associated trees in the attribute part of the morphism.

In our framework which uses the double pushout approach to rewrite graphs, the arrows for attribute parts are reversed with respect to the arrows of the structural parts. This reversal permits us to have a pseudo-pullback (pushout in dual category) to organize the computations with attributes. The main idea of changing the orientation of the arrows for the functions is to allow that the attributes in graph *L* can be stored in the graph *K* and then the attributes in graph *R* can easily "go and pick up" any value of attributes in the graph *K*. Because graph *K* is the intersection of graphs *L* and *R*, it contains only the common attributes of *L* and *R*. To preserve information, we may need several computation functions converging to the same target. The use of multirelations follows naturally from the same assumptions. We assume that all lambda-terms (containing, probably, free variables) are defined in the same context that remains fixed (in principle, the context may be infinite). Thanks to this mechanism, several attributes in graph *L* can be computed into one attribute in graph *K*, and several attributes in graph *R* can share the same value of attributes in *K*.

2.2.2 Attributes computations in DPoPb

Now we take the same example presented in Section 2 on the computation of *n!* to illustrate the attribute computation in DPoPb. InDPoPb to calculate *n!* we need only one rule shown in Fig.5.

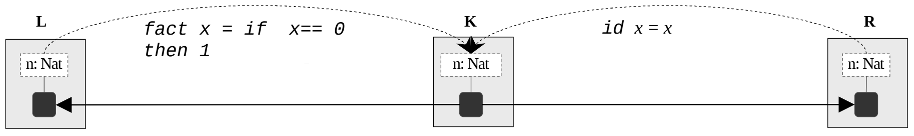


Fig. 5. Calculating *n!* by graph rewriting and attribute computation in DPoPb

The number *n* for which *n!* will be computed is specified as an attribute of type *Nat* associated to the unique structural node of the graph *L*. The computation of *n!* is realized by a graph rewrite which preserves the structure graph. So we just discuss here the computation to perform on attributes. The attribute part of the morphism *l: K → L* specified at two levels: a relation connecting the attribute *Nat* in *L* to the attribute *Nat* in *K* and a lambda-term associated to this relation to define the computation function realized during the graph rewriting. To simplify programs of Fig. 5, we replaced the lambda-term by the definition of a function *fact* which computes the factorial value for its parameter *x*. On the right-hand, the attribute part of the morphism *r: K → R* specified with a relation connecting the attribute *Nat* in *R* to the attribute *Nat* in *K* and an *id* function that allow copy the value of *Nat* in *K* to *Nat* in *R*. Given an initial graph *G* with a concrete value of *n*, the lambda-term of *l* will be applied to *n* to produce the result *n!*. This computation is performed by the β-reduction mechanism which substitutes the effective value of *n* for the formal variable *x* in the term. So the computation of *n!* in this example with DPoPb system requires only one transformation rule and one realizing step.

As seen in the illustrating example of this section, in certain cases, the systems based on ∑-algebras cannot represent directly complex computations on attributes if the operators used in the computation are not defined in the supporting ∑-algebras. In such cases, users have to decompose the computation into several rules (e.g. as illustrated in Fig. 2).

In DPoPb, computations are based on lambda-calculus [25], a formal model for computations. Let us recall that in this system, we can express computations with lambda-terms which allow representing the terms (*s*), the function abstractions (*λs . t*) and the function applications (*t s*). Lambda-terms then can be evaluated by the simple and powerful β-reduction mechanism based on substitutions. This reduction mechanism is semantically defined and can be easily implemented by a computer program. Thanks to this generic model of computation, the DPoPb approach can allow a more natural and straightforward way to represent complex computations defined only by abstraction and application of functions (e.g. as shown in Fig. 5).

3 Implementation of HLR and DPoPb

In this section, we compare the potential implementations of HLR and DPoPb with respect to their computational models. First we analyse the requested effort to implement exactly the foundation models of each approach. Then we discuss the solutions used by some significant systems to implement the HLR framework, as well as our solution for implementing the DPoPb framework. Discussions show the distance between the implementation and the formal model in each approach and some important side-effects raised in the tools resting on HLR.

3.1 Underlying mechanism for the transformation engine

In the HLR approach [3], attribute values are defined by separate data nodes which are elements of some algebras. When attributed graphs and graph morphisms are considered over Σ-algebras, operations and constants defining the algebra must be always present and thus previously defined. Of course, this is practically impossible because it is very cumbersome to implement large graphs and unattainable for infinite graphs. In a tool implementation this problem could be solved by including the attribute values of the algebra graph that are directly reachable from the structural part of the graph. Consequently, most of the systems based on the approach HLR use an object-oriented programming language to implement attribute computations. Concretely, pre-conditions, post-conditions and actions are mainly expressed in an object-oriented programming language: Java for AGG [1] and Fujaba [21], C++ for GReAT [20], and Python for AToM[3] [10]. Besides this popular solution, actions changing the models are sometimes coupled to the rule selection process as in VIATRA [17] which supports ASMs or in VMTS [18][19] where UML-like models are manipulated owing to stereotyped activity diagrams, XSLT and (Imperative) OCL.

Let’s consider a system using an external language to express actions as well as conditions on attributes, for instance AGG tool. In AGG, graphs are attributed by Java objects which can be instances of Java classes from libraries like JDK or from user-defined classes. The main difference with the formal system is the use of Java classes and expressions instead of algebraic specifications and terms. Thanks to some interoperability with the host language, the obtained system is a general purpose graph transformation tool covering a large variety of applications including graph transformation. However, classes of the underlying object-oriented language whose semantics is not covered by the formal foundation belong to applications as well.

A guiding principle of DPoPb is to propose a close relationship between the formal ground and its underlying engine. The lambda-calculus which formalizes the algorithmic notion of a function is proposed as a model of attribute computations. Existing graph nodes describing attributes thus can be reused and updated thanks to lambda-terms implementing the attribute computations. Lambda-terms can be easily expressed in a functional programming which is based also on lambda-calculus. Such an implementation preserves the semantics of the formal model, and provides static strong typing, polymorphism, higher-order functions and lazy evaluation for the graph rewriting system. For implementing the DPoPb prototype, we chose the Haskell language and benefit all of these advantages of the lambda-calculus paradigm.

3.2 Types declarations of attributed graphs

In HLR graph transformation tools, an attribute is often declared as a variable in a conventional programming language. For instance, in AGG, an attribute is implemented by a Java variable which can be assigned to any value conforming to its type. Because users can use any Java acceptable expressions to compute attributes' value, the Java type system defines the type system of the graph rewriting. This issue is not specific to AGG; it exists also in other known graph transformation systems such as Fujaba, GReAT, AToM^3... A strongly typed language such as Java is considered useful to reinforce the security of programs by preventing programmers from making freely mistakes. In fact, this statement is not true in certain cases. For example, a class in Java is perhaps a wrong subtype of its superclass. In order to be a subtype, the methods of the subclass must satisfy the superclass' specifications. This relation cannot be checked at compile-time, so it is possible to create a subclass that is not a subtype [22]. Hence the type system used by the graph transformation scheme where each attribute has a name, a type and a value can be unsecure.

For instance, let us consider the two following classes *Integer* and *MyInteger* in Java. *MyInteger* looks like an *Integer* by adding an attribute which specifies a name *s* for the value *v*:

```
public class Integer {
        private int v ;
        public Integer (int v) {…}
        public boolean equals (Integer i) {...}
}

public class MyInteger {
        private int v ;
        private String s ;
        public MyInteger (int v, String s) {…}
        public boolean equals (MyInteger i) {…}
}
```

MyInteger is not a subtype of *Integer*. To insure subtyping, we need that *MyInteger* must have a stronger specification than *Integer*. This is not the case because the type of the parameter of the *equals* method of *MyInteger* should be at most as strict as in the supertype. Using the Java's *extends* relation between an *Integer* and *MyInteger* is also not appropriate with respect to subtyping. This is mainly because a C++ or Java class defines at the same time attributes (state) and methods (behaviour). Subclasses are not subtypes. Consequently, an *Integer* object cannot be dynamically substituted by a *MyInteger* one during the rewriting process.

In contrast to these systems, the DPoPb can avoid such kinds of problem on the type system. In Haskell, a safe polymorphic type system is supported by a powerful type inference algorithm. A type specification is separated from its methods (functions). A class specifies the operations that the types must support. It's a template for types. A type is said to be an instance of a class if it supports these operations. For instance, here is the (incomplete) *Eq* class from the Standard Prelude defining the == (equals) and /= (different) functions.

```
class Eq a where                        -- a is an instance of Eq
(==), (/=) :: a -> a -> Bool            -- if a implements == and /=
x /= y   =   not (x == y)

data MyInteger = MyInteger {v :: Integer, s :: String}

instance Eq MyInteger where
   (MyInteger v1 s1) == (MyInteger v2 s2)    =   (v1 == v2) && (s1 == s2)
```

This code defines *MyInteger* as a data type which wraps an *Integer* presenting the value *v* and a *String s* representing the name of the value. This type is then considered as an instance of *Eq*. The three definitions (class, data type and instance) are completely separated and there is no rule about how they are grouped.

We think that this separation is more secure than actual attribute declarations in an object-oriented host language because well-typed lambda-terms are always well-behaved with respect to reduction. In addition, all the types associated with a function definition can be checked at compile-time, and inferred automatically. To take a full advantage of the typed lambda-calculus, an attractive perspective of DPoPb is to define type checking rules between the types of the computation functions and the types of the attributes of the attributed graph.

3.3 Attributes computations

3.3.1 Loading compiled codes

As previously stated, several transformation systems rely upon an underlying language for the specification of textual constraints and attribute updates. These definitions have to be compiled and provided to the graph transformation machinery in the form of a dynamic library, which is loaded at runtime. Within the transformation environment, it is quite easy to propose a special attribute editor that pops up when a graph object is selected for attribution. However, the user cannot directly access to the code of the function dealing with these attributes. As the function is considered as a black box, round trips between the host language and the graph rewriting tool are necessary to finalize the computation code. Each round trip is translated by a compilation process in the host language.

In DPoPb, the use of a unified formalism based on type theory for manipulating attributes enables a reliable environment so that both structural and attribute manipulations are handled in the same framework. The β-reduction mechanism used to evaluate lambda-terms can be easily implemented. Implementation which allows compiling and dynamically evaluating attribute computations is then possible. In Section 4, we will show how this capability is instantiated for the DPoPb's implementation.

3.3.2 Lazy evaluation

Another relevant feature for attribute computations is about lazy evaluation. Using this technique, no expression is evaluated until its value is needed and no shared expression is evaluated more than once. Lazy functions, also called non-strict, only evaluate their arguments when needed. On the opposite, C functions and Java methods are strict and evaluate their arguments in an eager mode. Lazy evaluation makes it possible for functions to manipule infinite data structures. This interesting feature enables us to describe an object without being tied to one particular application of that object.

For comparison purposes, let us consider an infinite list of integers starting from a given value. Such a lazy list can be represented in Java as a process [23] which returns objects either forever, or until no more are left[1]:

```
public interface Process {
        public Object nextElement () throws NoSuchElement;
}
```

[1] Java codes are extracted from [23]

```
public class NumFromProcess  implements Process {
        private int upto;
        public NumFromProcess (int n) {
                this.upto = n;
        }


        public Object nextElement () {
                return new Integer (this.upto ++);
        }
}
```

In Haskell language, for the same construction, we simply define:

```
numFrom n = n : numFrom (n + 1)
```

Extracting a finite list from *NumFromProcess* implies to manage exception handlings because we don't have a method to explicitly test for the presence of the next element. This test is assumed by throwing a *NoSuchElement* exception when the *nextElement* method is invoked. For instance, the Java following class *SingleProcess* computes a *Process* producing only one object:

```
public class SingleProcess  implements Process {
        private Object item;
        public SingleProcess (Object item) {
                this.item = item;
        }
        public Object nextElement () throws NoSuchElement {
                if (item == null) throw  new NoSuchElement ();
                else {
                        Object temp = item;
                        item = null;
                        return temp;
                }
        }
}
```

In addition, the *Process* interface defines a lazy list that is consumed as fast as it is produced and a shared expression is evaluated more than once. If previous elements need to be saved, then the programmer must add classes to store computed values in a structured data type.

In comparison, this mechanism is intrinsically supported by Haskell thanks to lazy evaluation. The following Haskell function *f* extract with *take* a list containing the successives values of the factorial computations, starting from 1! until *n*!. This is done by first building the infinite list *nats* and then applying the *fact* function to the obtained *nats* list. With the same technique, we build the infinite list *facts* of factorials. In this code, the *map* function is a higher-order function which goes through every element of a list and applies a function given by its first argument: + in the case of *nats* and *fact* for the list of factorials.

```
f n = take n facts                    facts = 1 : map (fact) nats
nats = 1 : map (+1) nats              fact 0 = 1
                                      fact n = n * fact (n -1)
```

With respect to functional programming languages, Java lacks some conciseness. Some libraries have been proposed to implement the lazy-evaluation mechanism for object-oriented environments. For instance Lambda4J [24] provides lazy lists and associated operations. More recently, LazyJ [23] extends Java's type system with lazy types. Besides expressiveness, a major challenge with lazy evaluation concerns sharing computation results. In all relevant functional language implementations, terms are represented as a graph. In the future, we would like to establish mappings between rewriting such terms and rewriting terms in an attributed graph.

4 The DPoPb prototype

To validate the theoretical model DPoPb, we have developed a prototype in Haskell language. In the first time, the goal of this prototype is to construct the basic DPoPb categorical concepts for attributed graphs rewriting when focusing on the implementation of attribute computation. Fig. 6 displays the architecture of our prototype.

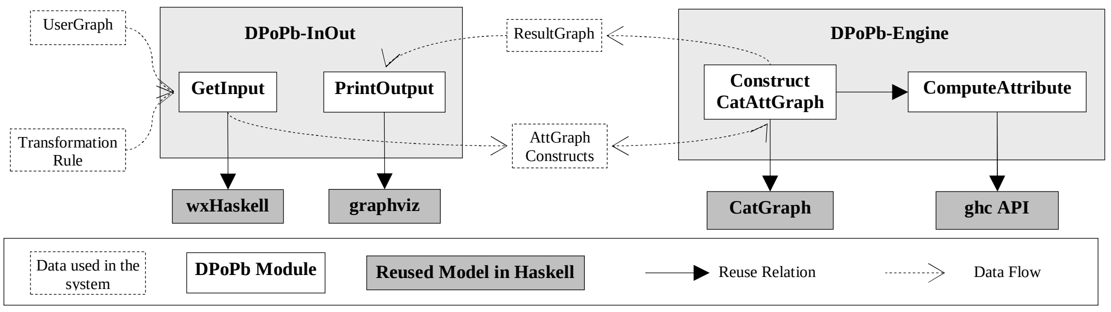


Fig. 6. Architecture of the DPoPb prototype

DPoPb prototype is a general purpose graph rewriting system composed of two modules *DPoPb-InOut* and *DPoPb-Engine*. The module *DPoPb-InOut* provides an interface for the prototype. It allows users to specify transformation rules and initial graphs (via the sub-module *GetInput*) as well as visualize the result of the transformation (by the sub-module *PrintOutput*). At the current stage of development, we base on the Haskell predefined modules *wxHaskell* and *graphviz* (defined in the Hackage Database [16]) for the graphical user interface and the graph visualization respectively.

The *DPoPb-Engine* module is the kernel of our prototype. It contains two sub-modules: *ConstructCatAttGraph* and *ComputeAttribute*. The sub-module *ConstructCatAttGraph* implements the main concepts of DPoPb including the colimits of the category of DPoPb attributed graphs (*CatAttGraph*) as well as the graph rewriting based on the approach double pushout. The sub-module *ComputeAttribute* supplies the utilities functions concerning attribute computations during the rewriting (*e.g.* the composition the attribute part of attributed graph morphims which is needed in the construction of *CatAttGraph* pushout; the dynamic evaluation of lamda-expressions representing attribute computations). In *ConstructCatAttGraph*, when constructing the structural part of the colimits we reused *CatGraph*, the implementation of Schneider [7] defining the colimits of the category of graphs. We also reuse some functions in the API of Glasslow Haskell Compiler to support the dynamic specification and evaluation of attribute computations.

Our main contributions in the development of the prototype concern solutions for the following theoretical and technical questions: how to implement the theoretical concepts of DPoPb, how to evaluate users-defined attribute computations at runtime and how to update graphs during the rewriting process.

4.1 Implementing the theoretical concepts of DPoPb

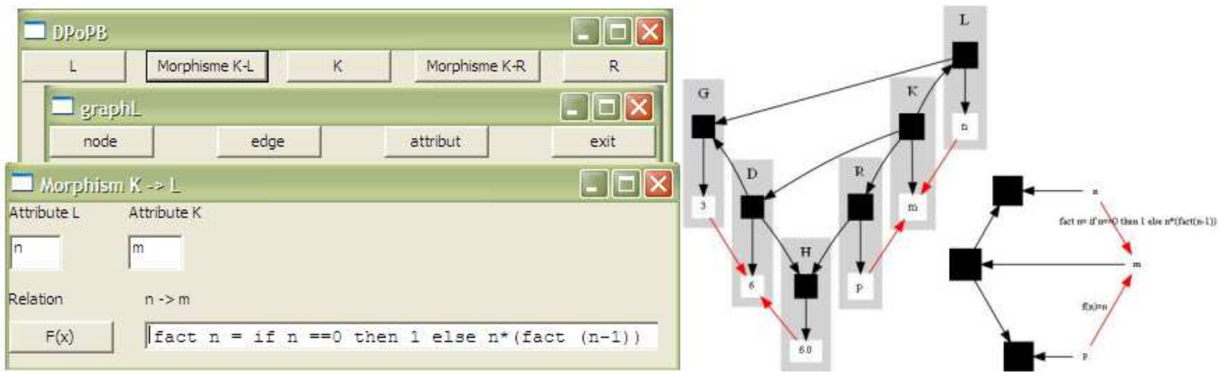
The mathematical model in [4] provided a formal framework for the category of attributed graphs *CatAttGraph* but it is not straightforward to map those categorical concepts into computational constructs. Thus, we had to define the constructive data structures and algorithms for storing graphs and graphs morphisms; for constructing the coproduct, coequalizer, pushout complement and pushout of the category *CatAttGraph*. The difficulty of this task resides in defining the attribute part of the graphs and graphs morphisms in the construction of each colimit such that its categorical properties are satisfied.

4.2 Evaluating users-defined attribute computations at runtime

We want to allow users to define their graphs together with attribute computations as non-compiled functions (written in Haskell for example). The challenge here is that at runtime the rewrite engine must enable the generation of Haskell codes of user-defined functions and integrate these codes into the engine in order to evaluate them in the rewrite process. Thus we need to support the meta-programming at runtime. To support this flexibility, we relied on the Glasgow Haskell Compiler (GHC) which proposes the necessary API functions to compile and evaluate Haskell functions. We used Haskell module hint 0.3.2 [16] wrapping those GHC functions to invoke the GHC compiler at runtime.

4.3 Updating graphs during the rewriting

The double pushout rewriting process necessitates an update of the transformation rules when the content of an initial graph is given. Using an imperative language, such an update is not difficult. However, the update implemented in an imperative language is undefined semantically and then out of control. Haskell is a pure functional language that does not allow side-effects. Hence we must ensure that computations with side-effects for the update of attributes during the rewrite process will be encapsulated to respect the functional style of the program. For this purpose, we base on monads [14]. We defined a monad transformer (*State transformer*) that enables hiding underlying machinery for updating graphs during the DPoPb process. The main interest here is that we can allow the update operations during the rewrite term process without losing the advantages of the functional paradigm and the Haskell type system.
We now show in Fig. 7 some screenshots of our prototype.



Attribute L: n
Attribute K: m
Relation: n -> m
F(x): fact n = if n ==0 then 1 else n*(fact (n-1))

fact n= if n==0 then 1 else n*(fact(n-1))

Fig. 7. DPoPb prototype's screenshots

In the left-hand are the widgets used to receive inputs including the transformation rules and the initial graph. Actually we do not implement an algorithm to find a match, so the initial graph *G* is took as an instance of graph *L* defined with concrete attributes' values. The frame *Morphism* $K \rightarrow L$ shows how the attribute part of the morphism is specified with the attribute relations and the computation functions. The input information will be encoded in the internal data structure and manipulated by the DPoPb engine module to construct the pushout complement graph *D* and the pushout graph *H*. To visualize DPoPb graphs, currently we rely on the GraphViz system

[28]. The DPoPb internal data structures are thus translated to the graph descriptions in the DOT language (by using the Haskell module *graphviz* [16]) which can be displayed with GraphViz as shown in the right-hand of the figure. On the right side of this part, we display the graph representing the rules $L \leftarrow K \rightarrow R$. The attributes n of L and p of R are connected to the attribute m of K by the function factorial and the function identity respectively. The graph on the left side shows the pushout complement D and the pushout H constructed by applying the rule $L \leftarrow K \rightarrow R$ on the concrete graph G. Since the value of G's attribute is 3, the value of D's attribute is 6 - factorial of 3. The value of H's attribute is the copy of D's attribute, thus it is also 6.

5 Conclusion

In the HLR framework, attributed graph structures are given by algebras over a specific signature where the structure part and the attribute part are separated from each other. If this solution is theoretically acceptable, it is not very efficient and cannot be easily implemented for a general purpose graph transformation system. Consequently, users have to program and compile computation functions separately within a companion programming tool before integrating their functions into transformation rules.

Contrary to HLR approach, our approach proposes a single formalism that integrates the rewrite of structural parts of graphs with attribute computations. This solution rests on category theory and type theory thus doesn't entail a semantic gap between the theoretical model and its implementation. This advantage has been validated by an ongoing–developed prototype of the DPoPb system implemented in the Haskell language.

We have identified two directions for future researches. The first one concerns proving properties of transformations. As we want keeping trace of evaluation or verification of the correctness of attributes' computations during the transformation, we plan to import transformations supported by our DPoPb tool into the Isabelle/HOL proof assistant via Haskabelle [26] in order to specify and prove relevant properties the transformations have to satisfy. Another direction deals with reasoning on programs transformations. It relies on the use of functional programming languages for programming applications based on rewriting attributed graphs. As these languages promote a more abstract style of programming and support higher-level constructions, we have in mind simplification of programs transformation and to cope with them as functional programs.

References

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

[2] Haskell Homepage, http://www.haskell.org/

[3] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, ICGT, volume 3256 of LNCS, pp. 161-177, Springer, 2004.

[4] Maxime Rebout. Une approche catégorique unifée pour la réecriture de graphes attribuées. PhD thesis, Université Paul Sabatier, 2008.

[5] Maxime Rebout, Louis Féraud, and Sergei Soloviev. A unifed categorical approach for attributed graph rewriting. In E.Hirsch, A.Razborov, A.Semenov, and A.Slissenko, editors, CSR, volume 5010 of LNCS, pp. 398-409, Springer, 2008.

[6] Maxime Rebout, Louis Féraud, Lionel Marie-Magdeleine, Sergei Soloviev. Computations in Graph Rewriting: Inductive types and Pullbacks in DPO Approach. In IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2009), Krakow, Pologne, 2009.

[7] Hans Jurgen Schneider. Implementing the Categorical Approach to Graph Transformations with Haskell. In An Introduction to the Categorical Approach, Draft March 7, 2007.

[8] Harmen Kastenberg. Towards Attributed Graphs in Groove: Work in Progress. Electr. Notes Theor. Comput. Sci. 154(2), pp. 47-54, 2006.

[9] Schurr, A. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. Proc. WG89. LNCS 411, pp. 151-165, Springer, 1990.

[10] de Lara, J. and Vangheluwe, H. AToM3: A Tool for Multi-Formalism Modeling and Meta-Modelling. Proc. FASE'02, LNCS 2306, pp. 174-188, Springer, 2002.

[11] Lowe, M., Korff, M., Wagner, A. An Algebraic Framework for the Transformation of Attributed Graphs. In Term Graph Rewriting: Theory and Practice, John Wiley and Sons Ltd. (1993), pp. 181-1993.

[12] Heckel, R., Küster, J., Taentzer, G. Confluence of Typed Attributed Graph Transformation with Constraints. In Proc. ICGT 2002, Volume 2505 of LNCS, Springer (2002), pp. 161-176.

[13] Berthold, M., Fischer, I., Koch, M. Attributed Graph Transformation with Partial Attribution, Technical Report 2000-2, 2000.

[14] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.

[15] Bézivin, J. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171-188, 2005.

[16] Hackage Database http://hackage.haskell.org/packages/hackage.html

[17] Varro, D., Pataricza, A. Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds., Proc. UML 2004, 7th International Conference on the Unified Modeling Language, Lisbon, Portugal, Springer (2004), pp. 290-304.

[18] VMTS Web Site, http://avalon.aut.bme.hu/_tihamer/research/vmts

[19] Levendovszky T., Lengyel L., Mezei G., Charaf H. A Systematic Approach to Metamodeling Environments and Model Transformation Systems. In VMTS, 2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT.

[20] Daniel Balasubramanian, Anantha Narayanan, Chris vanBuskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006); workshop at ICGT.

[21] Robert Wagner. Developing Model Transformations with Fujaba. In Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany (Holger Giese and Bernhard Westfechtel, eds.), vol. tr-ri-06-275 of Technical Report, pp. 79-82, University of Paderborn, September 2006.

[22] Sophia Drossopoulou, Susan Eisenbach. Java is Type Safe – Probably. European Conference on Object Oriented Programming, 1997.

[23] Dekker, Anthony H. Lazy functional programming in Java. SIGPLAN Not. Volume 41, Number 3, pp. 30-39, 2006.

[24] Lambda4J Web Site. http://www.nongnu.org/lambda4j/

[25] Henk Barendregt, Erik Barendsen. Introduction to Lambda Calculus, 1994.

[26] Plump, D. and S. Steinert. Towards Graph Programs for Graph Algorithms. In ICGT, LNCS 3256, pp. 128-143, Springer, 2004.

[27] Haskabelle website : http://www.cl.cam.ac.uk/research/hvg/isabelle/haskabelle.html

[28] GraphViz website : http://www.graphviz.org