# Graph Rewriting

## Applications of Graph Rewriting and the Graph Rewriting Calculus

Jan Lehmann
University of Technology Dresden
Department of Computer Science
Jan.Lehmann@inf.tu-dresden.de

## ABSTRACT

This paper will give an overview over different approaches for graph rewriting, namely term graph rewriting, the graph rewriting calculus and an algebraic approach presented in [3]. Whereas term graph rewriting is little more than a slightly improved version of term rewriting, the graph rewriting calculus is a powerful framework for manipulating graphs.

## 1. INTRODUCTION

Rewriting in general is widely used. Not only in computer science but also in everydays life. Assume you want to buy two pieces of chocolate and each piece costs $0,70$ €. In order to calculate the amount of money to pay, you write down (or think of) $2 * 0,70$ €. The result of this, not really difficult, computation is $1,40$ €. How this is correlated with rewriting? Formally your computation is the application of the following rewrite rule:

$$2 * 0,70 \rightarrow 1,40 \tag{1}$$

As presented in [4] term rewriting is a quite powerful tool to rewrite terms and can be widely used, for example in functional languages or logical languages. So why do we need an additonal theory for graph rewriting? The reason is, that terms can be represented as trees and trees have some severe limitations like the lack of sharing and cycles. Especially sharing can be useful when optimizing programming languages. Sharing means, that a node in the graph is referenced more than once. This is not possible in trees, because in a tree every node has only one incoming edge. However in a graph a node can have several incomming edges, allowing to point to this node from several other nodes in the graph. For example, when building the abstract syntax graph of a programming language, one only needs to reference a variable once, regardless of the number of occurences in the source code.

The possiblility to use cycles allows you to define rewrite rules like this:

$$x \rightarrow f(x) \tag{2}$$

As one can see here, the rule produces a right hand side which regenerates the left hand side again. The application of this rule would result in an endless cycle of function $f$ calling itself:

$$f(f(f(\ldots))) \tag{3}$$

Another advantage of the graph rewriting approaches is, that they are higher order, what means that rewrite rules can be generated with other rewrite rules. This also means that one can match a variable to a function symbol. Imagine the rewrite rule from above is held more general:

$$2 * x \rightarrow double(x) \tag{4}$$

Lets also assume that the price of our piece of chocolate is given without VAT (value added tax). The VAT will have to be added by a function $vat()$. Due to the ability to assign function symbols to variables, one can match the left hand side of 4 with

$$2 * vat(0,59) \tag{5}$$

In order to be able to buy more than one piece of chocolate you need the higher order capabilities of term graph rewriting or the graph rewriting calculus, both presented in this paper. A nice side effect of being able to match variables with functions is, that you can share whole subgraphs in graph rewriting. In addition to saving memory, you need to evaluate the shared subgraph only once.

## 2. TERM GRAPH REWRITING

This section will provide an operational approach on graph rewriting, defining a graph as set of nodes, a successor function and a labeling function.

### 2.1 Notation

Term graphs are graph representations of (algebraic) terms. They can be represented as tuple $G = (N, s, l)$ where $N$ is a set of nodes in the graph. $s$ is defined as function returning the direct successor of a given node: $s : N \rightarrow N^*$ and $l$ provides a label for each node in $N$. There is a special label for empty nodes: $\perp$. Typically empty nodes are nodes representing variables.

Term graphs may contain roots and garbage. If a term graph has a root, it will be denoted as $G = (r, N, s, l)$ where $r \in N$ is the root node of the graph. Nodes are called garbage if

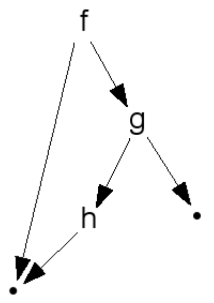

**Figure 1:** $f(x, g(h(x), y))$

there is no path between the root and these nodes. Garbage can be removed via garbage collection.

## 2.2 Representation

This leads to the question, how one can represent term graphs graphically. Lets take algebraic expressions as example. Every part of the expression becomes a node in the graph. Constant values or algebraic variables like $x$ in $f(x)$ are seen as empty nodes and drawn as $\cdot$.

*E*xample 1: The term $f(x, g(h(x), y))$ will lead to the following term graph: $G_1 = (n_1, N, s, l)$ where:

- $N = \{n_1, n_2, n_3, n_4, n_5\}$
- $s : s(n_1) = n_2n_3, s(n_3) = n_4n_5, s(n_4) = n_2, s(n_2) = s(n_5) = e$
- $l : l(n_1) = f, l(n_3) = g, l(n_4) = h, l(n_2) = l(n_5) = \perp$

The rules above define a graph with 5 nodes. The nodes $n_2$ and $n_5$, representing the variables $x$ and $y$ are leafes of the graph. The resulting graph looks like in figure 1.

## 2.3 Rewriting

To be able to rewrite term graphs, we need some kind of formalism to define that two term graphs are equal. This formalism is called graph homomorphism and means a function that maps the nodes of one graph $G_1$ to the nodes of a graph $G_2$. A variable in one graph may be mapped to any kind of subgraph of the other graph. That means that the following homomorphism is possible:

$$G1 : add(x, x) \rightarrow G2 : add(s(y), s(y)) \tag{6}$$

Figure 2 shows which parts of $G_1$ are matched against which parts of $G_2$.

The actual rewriting is defined by rewrite rules. A rewrite rule is a triple of form $(R, l, r)$. $R$ is a term graph and $l, r$ are the roots of the left and right hand sides respectively. Therefore $gc(l, R)$, which means the term graph $R$ with $l$ as root node after garbage collection, is the state of the term graph to rewrite (or a subgraph of it) before the rewriting and $gc(r, R)$ after the rewrite step.

The rewriting itself is done as follows:

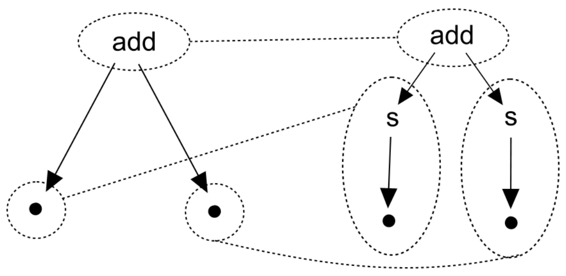

**Figure 2:** $G_1 \ll G_2$

- Find a subgraph that matches to the left hand side.
- Copy the parts of the right hand side to the graph to rewrite when there are no occurences yet.
- Redirect all pointers to the root of the left hand side to the root of the right hand side.
- Run garbage collection to remove parts of the left hand side, that are not reachable anymore.

Example 2 ([2]) shows the application for rewriting a graph defining the addition of two natural numbers. The term to rewrite is $add(s(0), s(0))$ where $add$ is the addition of two natural numbers and $s$ denotes the successor of a number. In fact this term simply formalizes the addition 1+1. This term as graph will look like the left graph in figure 3. To rewrite this term we use the rule $add(s(x), s(y)) \rightarrow s(add(x, s(y)))$. This rewrite rule is drawn as graph 2 in figure 3. This notation is a little bit confusing because both, the left hand side and the right hand side are drawn into one graph. To see the left hand side one has to set the left upper node as root and run garbage collection, which removes all nodes that are not reachable from this root. The same could be done for the right hand side. Now we can match the left hand side of the rule to the graph to rewrite. the left $\bullet$ will be mached to $x$ and the right $\bullet$ will be mached to $s(y)$. Now we copy all nodes of the right hand side of our rewrite rule to graph 1 and reset the root to the root of this right hand side. After doing this, we remove all nodes, which are not reachable anymore. This are the left nodes labeled $add$ and $s$. As result of this operation we get a graph looking like graph 3 in figure 3 representing the term $s(add(0, s(0)))$. In order to finish the addition one would define a second rewrite rule like $add(0, x) \rightarrow x$ to declare that the addition of 0 and a number equals to the number.

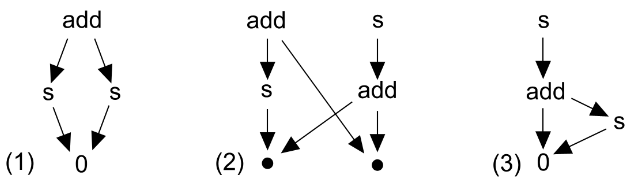

**Figure 3: Example 2**

## 3. THE GRAPH REWRITING CALCULUS

The approach of term graph rewriting is rather simple and understandable, but also rather weak. Weak means in this case, that one has no control about the application of the rewriting rules. Rules are always applied where possible. This is because the application of rewrite rules is done at meta level, not at object level. There exist more sophisticated methods to deal with graph rewriting. One is the extension of the $\rho$ calculus, the so called $\rho_g$ calculus or Graph Rewriting Calculus as presented in [2].

### 3.1 Overview

The $\rho_g$ calculus is a higher order calculus, which main feature is to provide an abstraction operator, that is capable of handling more complex terms on the left hand side, than just simple variables. An example of an abstraction is $f(x) \rightarrow x$. As one can see, an abstraction is denoted by "$\rightarrow$". Such an abstraction can be seen as rewrite rule. The application of an abstraction is written as follows:

$$(f(x,x) \rightarrow 2x)f(a,a) \tag{7}$$

This means that the abstraction inside the first paranthesis is applied to the term $f(a,a)$. As in $\rho$ calculus the application of such an abstraction is done via a constraint. The application in (7) would result in

$$2x[f(x,x) \ll f(a,a)] \tag{8}$$

Every application of a rewrite rule to another term is done like this. First the right hand side of the rewrite rule is written as term and than the matching of the left hand side of the rewrite rule and the term, the rule was applied to, is written as constraint to this right hand term. The operator $\_\,[\,\_]$ is called a constraint application. These constraint applications are useful for matching terms, that will be explained in the subsection about reducing terms. It also allows us to describe sharing of constants, as mentioned in the section about graph rewriting. There is not nessessarily only one constraint in $\rho_g$ calculus but a list of constraints, which are separated by comma. Such a constraint list can be generated when a constraint is applied to a term inside another constraint or if a function of arity ¿ 1 is matched. In the last case every parameter of the left hand side of the matching is matched against the corresponding parameter of the right hand side of the matching.

Another operator I did not consider so far is $\wr$. It is the so called structure operator which by default has no fixed semantics. This fact shows, that the graph rewriting calculus is more framework than a ready to use tool.

### 3.2 Further formalisms

Before actually showing how the example above is rewriten, I have to give some additional formalisms. One referes the operator $\_ \ll \_$. This operator is called matching operator, which means it tries to match the left and the right hand side. A special form of matching operator ist $\_ = \_$, that can be seen as association, for example $x[x = a]$ means that variable $x$ is associated with value $a$.

Another important thing is the theory of bound and free variables. A bound variable is a variable that occurs on a left hand side of a matching constraint. The table below shows how the free ($\mathcal{FV}$) and bound variables ($\mathcal{BV}$) are determined.

| $G$ | $\mathcal{BV}$(G) | $\mathcal{FV}$(G) |
|---|---|---|
| $x$ (var) | $\emptyset$ | $\{x\}$ |
| $k$ (const) | $\emptyset$ | $\emptyset$ |
| $G_1G_2$ | $\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$ | $\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$ |
| $G_1 \wr G_2$ | $\mathcal{BV}(G_1) \cup \mathcal{BV}(G_2)$ | $\mathcal{FV}(G_1) \cup \mathcal{FV}(G_2)$ |
| $G_1 \rightarrow G_2$ | $\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1)$ $\cup\, \mathcal{BV}(G_2)$ | $\mathcal{FV}(G_2) \setminus \mathcal{FV}(G_1)$ |
| $G_0\,[E]$ | $\mathcal{BV}(G_0) \cup \mathcal{BV}(E)$ | $(\mathcal{FV}(G_0) \cup \mathcal{FV}(E))$ $\setminus\, \mathcal{DV}(E)$ |

In a constraint $E$ there is an additional type of variables, the defined variables $\mathcal{DV}$.

| $E$ | $\mathcal{BV}(E)$ | $\mathcal{FV}(E)$ | $\mathcal{DV}(E)$ |
|---|---|---|---|
| $\epsilon$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $x = G_0$ | $x \cup \mathcal{BV}(G_0)$ | $\mathcal{FV}(G_0)$ | $\{x\}$ |
| $G_1 \ll G_2$ | $\mathcal{FV}(G_1) \cup \mathcal{BV}(G_1)$ $\cup\, \mathcal{BV}(G_2)$ | $\mathcal{FV}(G_2)$ | $\mathcal{FV}(G_1)$ |
| $E_1, E_2$ | $\mathcal{BV}(E_1) \cup \mathcal{BV}(E_2)$ | $\mathcal{FV}(E_1)$ $\cup\, \mathcal{FV}(E_2)$ | $\mathcal{DV}(E_1)$ $\cup\, \mathcal{DV}(E_2)$ |

The distinction of free and bound variables is important for the so called $\alpha$-conversion. This means renaming of variables when evaluating a term. Renaming might become nessessary when applying a rewrite rule to a term which has equaly named variables as these rewrite rule. In order to prevent free variables to become bound accidently they are renamed before the actual matching. Lets assume you have a $\rho_g$ term of form $G \rightarrow H$ where $G$ and $H$ are $\rho_g$ terms themselfes. In case of $\alpha$-conversion every free variable in $G$ gets another, not yet used, name in order to prevent it from becoming bound accidently. This is similar for constrainted terms. There the term where the constraint is applied to will get it's free variables renamed if there is a equaly named variable in one of the left hand sides in the constraint list.

### 3.3 Graphical Representation

As long as there are no constraints in the graphical representation of $\rho_g$ -terms is just as defined in the section about term graph rewriting above. Recursion can be represented as (self-)loops and sharing as multiple edges. A little bit problematic is the representation of matching constraints. These constraints are terms which can be drawn as graphs themselfes but they do not really belong to the main graph. [2] suggests drawing of these matching constraints as separate boxes. The boxes can be nested, due to the fact, that matching constraints can be nested, too. The roots of the boxes and sub-boxes are marked with $\Downarrow$. Figure 4 visualizes the rewrite rule $mult(2, s(x)) \rightarrow add(y, y)[y = s(x)]$ An interesting question on this example is: Why isn't the $s(x)$ on the left hand side shared, too? This was not done in order to keep the term wellformed. The condition of wellformedness demands that there is no sharing between the left and the right hand side of a $\rho_g$-term.

### 3.4 Rewriting

So how to do the rewriting in $\rho_g$ calculus? To demonstrate this lets take our example from the term graph rewriting section. The rewrite rule we want to apply is

$$add(s(x), y) \rightarrow s(add(x, y)) \tag{9}$$

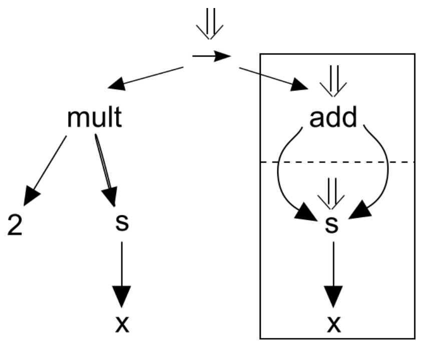

**Figure 4:** $mult(2, s(x)) \to add(y, y)[y = s(x)]$

and it is applied to the term

$$add(s(0), s(s(0))) \tag{10}$$

which is in fact the very complicated computation 1+2. The rewriting goes as follows. I will describe the steps after the example.

$(add(s(x), y) \to s(add(x, y)))add(s(0), s(s(0)))$
$s(add(x, y))[add(s(x), y) \ll add(s(0), s(s(0))]$
$s(add(x, y))[s(x) \ll s(0), y \ll s(s(0))]$
$s(add(x, y))[x \ll 0, y = s(s(0))]$
$s(add(x, y))[x = 0, y = s(s(0))]$
$s(add(0, s(s(0))))$

The first two lines follow the claims I made in the Overview part of this section. It is the application of the rewrite rule in (9) to the term in (10). In line three the function *add* was matched and splits up into two constraints which are applied as list to the term before [_]. In the fourth step this is done a second time with the successorfunction. The matching operator between $y$ and $s(s(0))$ is replaced by the application operator. This can be done because $y$ is a variable in the term and there is no need or possiblity to further match it in any way. The same is done in line 5 where $x$ equals to 0. The last line is finally the result of the rewriting. Unlike the pure $\rho$-calculus the $\rho_g$-calculus does not require the last step, which may be an advantage if one wants to preserve the sharing of equal parts.

## 3.5 Confluence and Termination

As presented in [4] there is no guarantee that the application of rewriting steps in term rewritings terminates. One can easily see that this also holds for the $\rho_g$ calculus. First of all, graphs may contain cycles. If a rewrite rule matches one of these cycles but does not eliminate it, it will be applied over and over again. A second example to show that a rewriting may not terminate is the following set of rules:

$f(x) \to x$
$x \to y$
$y \to f(x)$

The reduction always comes to it's starting point and restarts again.

In general there is also no clue that (graph-)rewriting is confluent. However the graph rewriting calculus restricts the left hand sides of it's rewrite rules in order to achieve this property. The restriction is that the left hand sides have to appliy to linear patters. A linear pattern is formally defined as follows:

$$\mathcal{L} := \mathcal{X} \mid \mathcal{K} \mid (((\mathcal{K}\ \mathcal{L}_0)\mathcal{L}_1)\ldots)\mathcal{L}_n \mid \mathcal{L}_0[\mathcal{X}_1 = \mathcal{L}_1, \ldots, \mathcal{X}_n = \mathcal{L}_n] \tag{11}$$

where 2 patterns $\mathcal{L}_i$ and $\mathcal{L}_j$ are not allowed to share free variables if $i \neq j$. Furthermore a constraint of form $[L_1 \lll G_1, \ldots, L_n \lll G_n]$, with $\lll$ is either $\ll$ or $=$, is called linear if all patterns $L_i$ are linear. The complete proof of confluence in the linear $\rho_g$ calculus can be found in [2].

## 4. AN ALGEBRAIC APPROACH

Another approach for rewriting of graphs was presented in [3]. The paper describes graphs as logical structures and their properties with help of logical languages. Graphs are described as classes $D(A)$, where $D$ is the class and $A$ is an alphabet containing the labels for the edges of graphs in $D$. Such a graph can be represented as follows:

$|G|_1 := \langle V_G, E_G, (edg_{aG})_{a\in A}\rangle$
$|G|_2 := \langle V_G, E_G, (lab_{aG})_{a\in A}, edg_G\rangle$

Where $V_G$ and $E_G$ are the sets of vertices and edges respectively, $lab_{aG}(x)$ means an edge $x$ labeled with $a$, $edg_G(x, y, z)$ is an edge $x$ from vertex $y$ to vertex $z$ and $edg_{aG}(x, y, z)$ describes the combination of $lab_{aG}(x)$ and $edg_G(x, y, z)$.

### 4.1 Properties of graphs

In order to match graphs for rewriting, one has to define isomorphisms between graphs. [3] defines two graphs as isomorphic if there exist bijections from $V_G$ to $V_{G'}$ and from $E_G$ to $E_{G'}$. A property of a graph is a predicate over a class of graphs. Such predicates may be expressed by logical languages like first order logic, second order logic and so on. The more powerful a language is, the better is it's expessiveness. A discussion about the expressiveness of several logical languages can be found in [3]. The following example expresses the property that there are no isolated vertices in a graph, that means that every vertex is linked somehow with the other parts of the graph.

$$\forall x \exists y \exists z \left[\bigvee edg_a(z, x, y) \vee edg_a(z, y, x) | a \in A \wedge \neg(x = y)\right] \tag{12}$$

### 4.2 Graph manipulations

In this approach graphs are not manipulated directly but overlayed with so called hypergraphs. Each hypergraph consists of hyperedges which are defined through a label and zero to many edges of the graph. There are three operations defined over these hypergraphs. The first is the addition of two hypergraphs denoted by $\oplus$. This operation merges the edges and vertices of two hypergraphs into one. The operation $\theta_{\delta,n}$ describes the merge of two vertices, i.e $\theta_{1,3}$ means that vertices 1 and 3 are merged into one vertex. Finally the operation $\sigma_{\alpha,p,n}$ describes renaming of vertices. The expression $\sigma_{1,4}$ means that vertex 1 is renamed to 1 and vertex 4 is renamed to 2, according to their positions in the subscript.

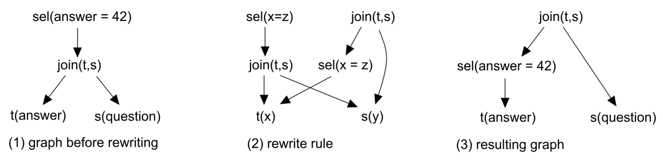

**Figure 5: Rewrite of a SQL execution graph**

With these basic operations one can define derived operations like the series-composition of hypergraphs:

$$G \cdot G' = \sigma_{1,4}(\theta_{2,3}(G \oplus G')) \tag{13}$$

First the hypergraphs $G$ and $G'$ are added together. Then the $\theta$-operation merges the second vertex of $G$ (2) with the first vertex of $G'$ (3) and finally the first vertex and the last vertex of the resulting hypergraph get new labels. The vertex in the middle is not labeled at all.

## 5. APPLICATIONS

Now we know what possiblities we have to rewrite graphs, but where is this useful? Graph rewriting is especially useful for optimization purposes. Everything that can be represented as graph can be optimized with help of graph rewriting, where the optimizations are encoded as rewrite rules (see also [1]). An example for this may be an SQL statement. Each statement can be transformed into a query execution plan, which essentially is a tree or graph. There exist several rules of thumb, like the rule, that a selection should be executed as early as possible in order to keep the amount of tuples as small as possible. This kind of optimization is called static because it does not use any knowledge of the data to transform. Lets assume we have a simple SQL statement like

```
SELECT *
FROM table1 t, table2 s
WHERE t.id = s.id;
AND t.answer = 42;
```

A query execution plan may look like in figure 5.1. The application of the rule in figure 5.2 would result in the graph of figure 5.3.

Another example, similar to the one above, is the execution graph of functional languages. One can imagine that there are similar rules as in the SQL example. The representation as graphs will save a severe amount of space because variables (in the second case) or whole subqueries (in the first example) can be shared.

The third application of graph rewriting, i would like to mention, is optimization and analysis of syntax graphs in languages like C♯ or Java. Rewriting can assist you in refactorings, too. The desired refactoring can be encoded as graph rewrite rule and will be applied to the syntax graph.

## 6. CONCLUSION

As we have seen, there exist several approaches for graph rewriting. All of them have some properties, like sharing and the possibility to handle cycles, in common, which make them to powerful tools for changing the structure of graphs. However, these approaches use different ways for defining rules and graphs and therefor differ in power and flexibility. The most intuitive way is term graph rewriting, but it's also a bit limited. The graph rewriting calculus is a powerful framework, which allows to build an own calculus on top of it. Common to all approaches is, that you can finally buy more than one piece of chocolate.

## 7. REFERENCES

[1] U. Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.*, 22(4):583–637, 2000.

[2] C. Bertolissi. The graph rewriting calculus : confluence and expressiveness. In G. M. P. Mario Coppo, Elena Lodi, editor, *9th Italian conference on Italian Conference on Theoretical Computer Science - ICTCS 2005, Siena, Italy*, volume 3701 of *Lecture Notes in Computer Science*, pages 113–127. Springer Verlag, Oct 2005.

[3] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.

[4] A. Rümpel. Rewriting. 2007.