Constructs for Programming with Graph Rewrites

Peter Rodgers

Computing Laboratory, University of Kent, UK
P.J.Rodgers@ukc.ac.uk

Abstract. Graph rewriting is becoming increasingly popular as a method for programming with graph based data structures. We present several modifications to a basic serial graph rewriting paradigm and discuss how they improve coding programs in the Grrr graph rewriting programming language. The constructs we present are once only nodes, attractor nodes and single match rewrites. We illustrate the operation of the constructs by example. The advantages of adding these new rewrite modifiers is to reduce the size of programs, improve the efficiency of execution and simplify the host graph undergoing rewriting.

1 Introduction

Graph rewriting is now a common visual paradigm. It is used in the specification [2] and parsing [3] of visual languages. It is also used, as here, as an alternative to regular text programming languages, particularly for applications where graphs are the dominant data structure. This paper concentrates on modifications to the Grrr graph rewriting programming language [6,7,8] in the light of our experience in programming with it. We expand and extend the research presented as work in progress in [9].

We introduce three rewrite modifiers: **once only** nodes, **attractor** nodes and **single match** rewrites. The overall advantage of these constructs is to make programming in Grrr easier, to shorten execution times and to reduce the number of housekeeping nodes in the host graph. The disadvantage is in increasing the complexity of both the paradigm and the language implementation. The constructs are additions to the language, rather than alterations, so there is backwards compatibility, and previous code will work with the new semantics. We emphasise that our approach is pragmatic, so the motivation for the modifications discussed here is based on experience in programming with the system.

The basic Grrr paradigm is a rewriting method that alters arbitrary directed graphs. It is serial both in the calling of trigger nodes that initiate transformations and in the graph rewriting process itself, where only one of several candidate subgraphs is rewritten on a call of a trigger node. The choice of subgraph to rewrite is deterministic. The rewriting process, without modifiers, is performed by a constant embedding with dangling edges deleted. Transformations are lists of graph rewrite pairs, with each rewrite having a LHS (Left Hand Side) graph and a RHS (Right Hand Side) graph. The difference in the LHS and RHS graphs indicates how nodes and edges in the host graph are to be added, deleted or retained. Only one rewrite in a transformation is used on a trigger call. Grrr tests the rewrites from the top down, using the first rewrite with a LHS that matches in the host graph. On the next application of the transformation the process is repeated, with the topmost rewrite being tested first again. This varies from other systems, such as AGG which makes all possible applications of a rewrites before passing down to the next rewrite.

This paradigm is simple in the sense it uses a basic rewrite mechanism and relates to current logic and functional languages in its LHS/RHS rewrite specification and its top down matching method. However, in graph rewriting there is no notion of functional transparency. That is, it is not possible to assume that a given execution of rewrite and subsequent rewrites that result from it will affect only a restricted part of a graph. This means that the Grrr system has to deal with marking the parts of the graph that are of interest. Other problems occur when deleting nodes, as dangling edges are deleted, so potentially useful information is then lost from the graph. Finally, the top down matching strategy can be overly simplistic, as often a rewrite will only be required to execute at most a single time in the host graph.

To address these problems we have introduced three new constructs. The first two, **once only** and **attractor**, are extra possible features of nodes in rewrites. **Once only** nodes appear in the LHS, and when specified by a programmer as **once only**, a node in a LHS of a rewrite can match with each node in the host graph at most once, making it easy to execute an operation on each node of the graph. **Attractor** nodes appear in the RHS of a rewrite and attract the dangling ends of any edges that result from deleting nodes. The third construct, **Single match,** is a feature of rewrites. Once the LHS of a **single match** rewrite has been found in the host graph, it will not be tested again.

Transformations are initiated by trigger nodes in the host graph. The scope for **once only** and **single match** is the trigger node in the host graph that initiated the transformation. This means that restrictions on the rewriting caused by these features only apply to further applications of the initiating trigger node. Hence, if there are two trigger nodes with the same name in the host graph, then a node in the host graph that matches a **once only** node as a consequence of a first trigger node cannot match again when the first trigger initiates the same transformation, but the node may match again as a consequence of the transformation being initiated by the second trigger node.

The constructs alter the rewriting process and reduce the number of steps performed during execution. **Once only** and **single match** do this by considering the history of matching in the host graph, but differ in that **once only** is node based and **single match** is rewrite based. **Attractor** nodes change a rewriting step, as the rewriting process is no longer a constant embedding of a subgraph. The interaction between these three modifiers is minimal, as a **once only** node and an **attractor** node may appear in **single match** rewrites without any effect on their operation. Indeed, a node may be a **once only** node in a LHS, and an **attractor** node in the corresponding RHS.

In Section 2 we discuss the semantics of the new rewrite modifiers, and give examples of use. In Section 3 we present our conclusions, and relate the modifiers to other graph rewriting based systems such as Progres [11], GOOD [5], Δ-grammar programming [4], MONSTR [1] and AGG [10].

2 The rewrite modifiers

The new constructs, **once only** nodes, **attractor** nodes and **single match** rewrites, are illustrated by example. An associational network, shown in Fig. 1, is the example host graph. These networks are used frequently in applications such as police investigations and social network research. Here there are six people in the network, with additional information about their ages. This is a small example, but this type of network can get very large, and would usually include much more information about the members.

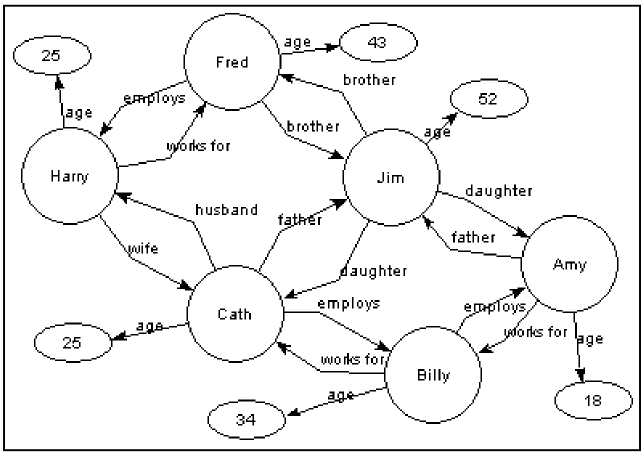

Fig. 1. The associational network

In order to execute programs on this host graph, trigger nodes will be added to it. These can be singleton triggers, triggers attached to some part of the host graph, or triggers with extra information associated with them, in the form of attached nodes. Trigger nodes are not shown in Fig. 1, but would appear with a rectangular border. The difference in node shapes indicates the different sort of data held by the nodes and can vary from graph to graph. In this case round nodes indicate people and oval nodes indicate information about people.
When matching a subgraph, round nodes may only match with round nodes and oval nodes may only match with oval nodes. This differs from the node typing found in systems such as Progres and AGG, in that types are visually specified rather than named, and Grrr types have no concept of hierarchy.

2.1 Once only nodes

The addition of **once only** nodes is designed to reduce the use of tags in transformations. The use of tags was common when programming in Grrr as they were used to indicate which nodes had been visited. Rewrites which iterated through a graph would test for the negative presence of a tag attached to a node, and if not present the rewrite will perform the required action on the node and create a tag attached to it. The next application of the transformation will then not match with that tagged node, because of the negative in the LHS of the rewrite. After all the relevant nodes have been matched the tags were removed by a garbage collection transformation.

This sort of operation is made easier with **once only** nodes. A node in the host graph that matches with a **once only** node in any LHS of a transformation will match with no other **once only** node in the transformation in future applications of the trigger node. This allows a graph to be iterated through, one node at a time, confident that after a node is visited it will not be visited again. This results in transformations that are clearer and easier to specify, during execution programming steps are avoided (testing for negatives, creating tags and destroying tags) and whilst being rewritten the host graph is clearer because it no longer contains tags.

There is no need for the **once only** nodes to have the same name, and it is possible for the **once only** node to be a variable in one LHS and a constant in another LHS graph. A **once only** node is treated as any other node in the rewrite, and can be deleted, or reconnected to various edges as desired. At present the restrictions are that a only one **once only** node is allowed in each LHS graph and a negative cannot be a **once only** node. These restrictions are enforced by the graph editor simply disallowing the option when LHS graphs are constructed. This is the standard technique, used throughout Grrr in order to maintain a correct syntax for rewrites.

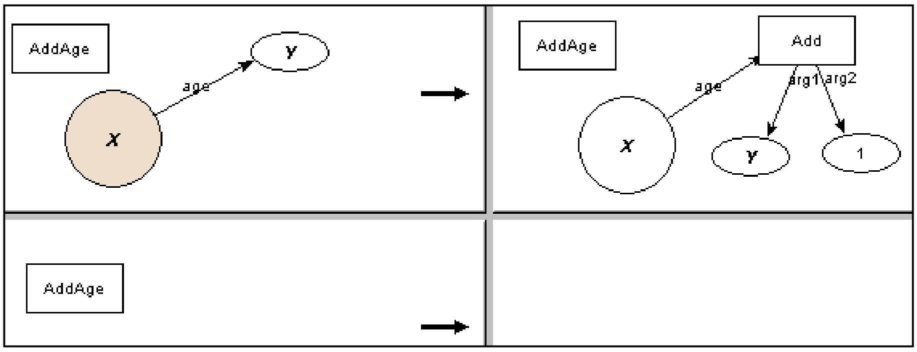

Fig. 2. The transformation ‘AddAge'. The shaded node labeled 'X' in the first LHS is a once only node

The use of **once only** nodes is shown in Fig. 2, where the node labeled ’X’ in the first LHS graph is a **once only**, indicated by shading. This transformation adds one to the age of all people in the network. On execution of the AddAge transformation by a trigger node of the same name in the host graph the first LHS will match in the host graph, with the ’X’ node matching with a person. The new nodes in the RHS are added to the host graph, resulting in an increment of the persons age. Then AddAge is triggered for the second time, and ’X’ can no longer match with the person matched first time, so another will be chosen. This iterative process continues until there are no new people to be matched, and so the first LHS cannot match. The second LHS is then tested, and will match. This rewrite deletes the trigger and so terminates the execution of the transformation.

2.2 Attractor nodes

Attractor nodes are less recent additions than **once only** and **single match**. It was realised at an early stage in the development of this programming system that the simple rewriting mechanism presented difficulties when dealing with node and sub-graph replacement. Replacing a node or defining mathematical expressions are tricky problems when dangling edges are deleted. **Attractor** nodes can be used in this type of case. A programmer specifies that (at most) one node in the RHS of a rewrite is an **attractor**. When a node deletion occurs, dangling edges then get attached to the **attractor** node. This allows replacement of a node by simply deleting it, and making its replacement an **attractor** node. This technique is also useful when dealing with tree replacement as new roots of subtrees can attract the connections to children of deleted roots. The effect is difficult to reproduce without **attractor** nodes because the edges attaching to a node cannot be iterated through easily.

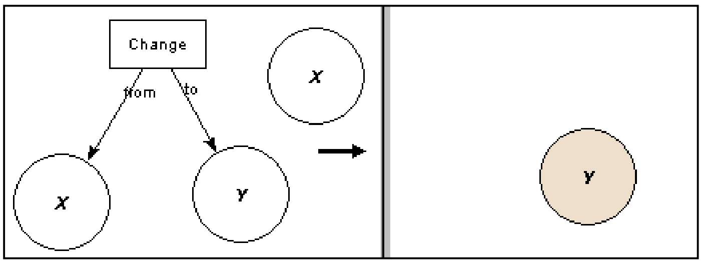

Fig. 3. The transformation ‘Change’. The shaded node labeled 'Y' in the RHS is an attractor node

An example of the use of **attractor** nodes is shown in Fig. 3 where the transformation Change replaces the node that matches with ’X’ with the node that matches with ’Y’. The only node, ’Y’, in the RHS is shaded and so is an **attractor**. The user adds to the host graph the Change node and two nodes attached to it by a ’from’ edge and a ’to’ edge, where the nodes are a duplicate of the node to be replaced and the replacement, respectively. These will match with ’X’ and ’Y’. The effect of the transformation is to delete the Change trigger node and both ’X’ nodes (i.e. the node added by the user and the node already in the host graph). The ’Y’ node is an **attractor** in the LHS, so any edges dangling because of the original ’X’ node deletion are attached to it, so performing a simple replacement. Note that the edge ’to’ is not attracted to the **attractor** node, despite it being left dangling by the deletion of the Change node because it is explicitly deleted in the rewrite (i.e. it appears in the LHS but not the RHS).

The use of **attractor** nodes can be seen in a different context in Fig. 2, that of arithmetic expressions. The built in transformation Add called by the first RHS adds its two arguments together, creating a result node and deleting the Add trigger node and the two argument nodes. The result node is an **attractor**, hence the ’age’ edge that is left dangling is attached to it. In general, there may be many such arithmetic triggers in a tree structure. The basic rewriting process ensures that the leaves of the tree are executed first, and the **attractor** nodes ensure that the result remains attached to the tree.

2.3 Single match rewrites

Single match allows the control of execution of rewrites to be specified more precisely than a simple top down matching strategy allows. The technique previously was to introduce flags. Negatives in LHS graphs would prevent the subsequent matching of rewrites which created the flags. Now rewrites specified as **single match** will match a single time in the host graph, and will be ignored during subsequent applications of the trigger node. This differs from the notion of prioritising rewrites, which forces some rewrites to be considered before others, but still allows rewrites to be executed a number of times. As with **once only** the overall effect is to streamline transformations, lessening the number of negatives and reducing the clutter in the host graph. There are no restrictions on which rewrites are **single match**, and on what they may contain.

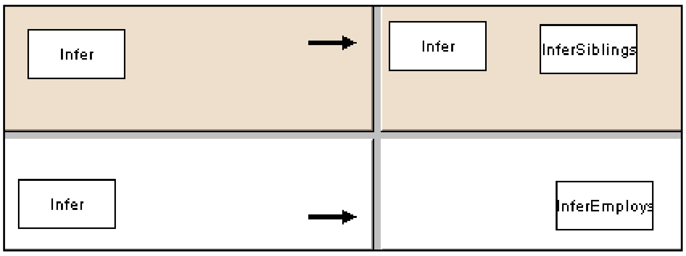

Fig. 4. The transformation ‘Infer'. The first rewrite, shown with a shaded background is a single match rewrite

Fig. 4 shows an example of **single match**. Infer calls two other triggers, InferSiblings and InferEmploys. The desired effect is to derive relationships between people where there is no relationship already, so that InferSiblings finds two people who are not currently directly connected and connects them if they share the same parent. It can be seen in Fig. 1 that ’Amy’ and ’Cath’ are both daughters of ’Jim’, hence this transformation will add a relationship ’sister’ between the two daughters. InferEmploys derives relationships between people where there is no current connection based on the notion that where there is an employee of an employee then it can be inferred the most junior employee works for the most senior. An example of this again in ’Amy’ and ’Cath’, where ’Cath’ employs ’Billy’ who employs ’Amy’. Hence ’Cath’ employs ’Amy’. We wish to infer connections between people, but only one connection is allowed between two people and sibling relations are more significant than employment relations. Hence, Infer must ensure that InferSiblings is called before InferEmploys.

This is achieved by using **single match** rewrites. The first rewrite of Infer is **single match**, and has a shaded background. It is executed once and only once and its effect is to call InferSiblings. This results in the sibling relationships being formed. The second call of Infer with the same trigger node does not test the first **single match** rewrite, so the second rewrite is executed. This deletes Infer and calls InferEmploys, which derives the employs relationships, and then the program terminates. The net effect on the host graph shown in Fig. 1 is to add a sister relationship between ’Amy’ and ’Cath’, rather than the employs relationship.

3 Conclusions

We have described three constructs for modifying graph rewriting. The **once only** nodes allow programmers to more easily iterate through the nodes in a host graph, the **attractor** nodes allow dangling edges to be retained in the graph, whilst the **single match** rewrites allow closer control of execution order. The overall effect is to make programming in Grrr simpler and more effective. These constructs seem to be unique, as similar modifiers do not appear in other programming languages or systems.

The effect on theoretical descriptions of graph rewriting has to be fully integrated if a formal model of Grrr using common graph grammar techniques is to be developed. Methods of describing graph rewriting, such as the common techniques based on pushouts, fail due to **attractor** nodes. These break the universal property of pushouts because dangling edges cannot be reattached without being considered new edges, these new edges then have not appeared in either the LHS or host graph. Also, a history of rewriting needs to be added to current graph rewriting descriptions to express the notion of **once only** and **single match** as these new constructs cannot be fully specified by looking only at a set of transformations and a host graph.

Further experimentation with the modifiers discussed here is a possible future area of work. The **once only** notion could be extended to edges, multiple nodes, or to a subgraph of the LHS graph. Single **once only** edges are easy to interpret, however there is a difficulty in interpreting the desired behaviour of multiple **once only** primitives. For instance, where there are two **once only** nodes in a LHS, can each node in the host graph match with both nodes or just one? Moreover, the mapping between **once only** nodes across several LHS graphs in a transformation cannot easily be specified, except by node label. This reduces the flexibility of the feature as currently variables and constants can map to the same **once only** node across different LHS graphs.

Grrr is serial and deterministic, however the modifiers discussed here could be used in parallel, non-deterministic systems. The **single match** rewrites, in particular are potentially of use, as similar execution order difficulties arise in systems such as Δ-grammar programming, where there are frequent examples of two transformations at the same conceptual level, being executed one after the other by initiating one directly from the other. **Attractor** nodes could also be a feature of other graph rewriting systems, although parallel rewriting means unexpected dangling edges appear which could cause difficulties.

There are other ways of producing the effects of the modifiers suggested here. **Attractor** nodes as a mechanism for node replacement could be replaced with a more general embedding system. It is not clear which variety of embedding is suitable (e.g. path redirection, as seen in Progres) typically such systems are textual in nature, which would break the overall visual nature of Grrr. **Single match** rewrites could be generalised by ensuring that each rewrite in a transformation had some textual application condition, where the number of matches could be included as a factor. This would also allow many other useful restrictions to added to rewrites. Again, this has disadvantages when attempting to maintain a fully visual system. An alternative to **once only** nodes is to modify the system to parallel rewriting. The motivations for serial over

parallel rewriting are discussed in [6,7], but allowing certain transformations to operate on the host graph in parallel is a possibility.
Other notions for addition to Grrr include the ‘fold’ and ‘Kleene *’ operators, seen in Δ-grammar programming, which modify the rewriting process so that nodes and subgraphs must no longer match in a one-to-one manner. This would prevent the requirement for literals to be repeated in the host graph. For example, in Fig. 1 'Harry' and 'Cath' could share the age node '25' and their ages could be changed independently.

Acknowledgements

This work was partially supported by a grant from the UK Engineering and Physical Sciences Research Council (EPSRC), grant reference GR/M23564.

References

1. R. Banach. MONSTR I -- Fundamental Issues and the Design of MONSTR. Journal of Universal Computer Science 2,4 (1996) 164-216.
2. R. Bardohl and G. Taentzer. Defining Visual Languages by Algebraic Specification Techniques and Graph Grammars. IEEE Workshop on Theory of Visual Languages. 1997.
3. H. Göttler. Graph Grammars and Diagram Editing. Proceedings 3rd International Workshop on Graph Grammars and Their Application to Computer Science. LNCS 291. Springer-Verlag. pp. 216-231.1987.
4. S.M. Kaplan, S.K. Goering and R.H. Cambell. Specifying Concurrent Systems with Δ-Grammars. Fifth International Workshop on Software Specification and Design. pp. 20-27. 1989.
5. J. Paredaens, J. Van den Bussche, M. Andries, M. Gyssens and I. Thyssens. An Overview of GOOD. ACM SIGMOD Record, 21,1. pp. 25-31. March 1992.
6. P.J. Rodgers. A Graph Rewriting Programming Language for Graph Drawing. Proceedings of the 14th IEEE Symposium on Visual Languages (VL’98). pp. 32-39. 1998.
7. P.J. Rodgers and P.J.H. King. A Graph Rewriting Visual Language for Database Programming. The Journal of Visual Languages and Computing 8(6). pp. 641-674. 1997.
8. P.J. Rodgers and N. Vidal. Graph Algorithm Animation with Grrr. Agtive99: Applications of Graph Transformations with Industrial Relevance, LNCS. Springer-Verlag. 2000.
9. P.J. Rodgers and N. Vidal. Pragmatic Graph Rewriting Modifications. Proceedings of the 15th IEEE Symposium on Visual Languages (VL’99). pp. 206-207. 1999.
10. M. Rudolf and G. Taentzer. Introduction to the Language Concepts of AGG. Available from http://tfs.cs.tu-berlin.de/agg/. 1998.
11. A. Schürr. Rapid Programming with Graph Rewrite Rules. Proceedings USENIX Symposium on Very High Level Languages (VHLL), Santa Fe. pp. 83-100. October 1994.