

Compiling AGG into the network linear graph reduction system

Charles Stewart

Theory and Formal Specifications, Technische Universität Berlin

This talk describes a course of research whose aim is to develop an optimising compiler from the visual graph transformation language AGG developed at TU Berlin to the *network linear graph reduction system* (NLGR system) developed by Alan Bawden. The NLGR system is a distributed run-time environment for a cluster of workstations that achieves dynamic load balancing by graph migration. The principal benefits of the proposed compiler from the AGG language developed at TU Berlin, are firstly to make maximally efficient use of the distributed resources available in the NLGR system without requiring explicit guidance from the programmer, and secondly to give a distributed framework which allows us to exploit work on distributed specifications using graph transformation systems in the AGG language.

The NLGR system

The NLGR system is a distributed graph reduction engine which allows a single graph to be divided among many *agents*; these agents are the distinct processor nodes running a standard run-time environment, coded in C, and communicate with each other through a TCP layer. The representation used in the graph is a *linear graph grammar*, a form of graph grammar which anticipated and generalises the interaction nets introduced by Yves Lafont (this kind of graph grammar is closely related to hyperedge replacement grammars, where vertices have ports which edges are linked to, and where edges are unattributed and undirected).

The patterns of the whole graph consist of two vertices connected by an edge. We shall need two ways of handling rewrites, depending upon whether the edge is contained in the local graph of an agent.

1. Rewrites of patterns within the local graph of an agent are computationally inexpensive. The patterns can be recognised by a simple hash table lookup; deletion of patterns and layout of replacement graphs is performed by C code generated from the specification of the linear graph grammar.
2. Rewrites not so contained are handled by graph migration (where a subgraph is detached from the local subgraph at one agent and glued onto the subgraph of another

agent). The NLGR system is distinguished by the low communication costs associated with graph migration.

3. The communication protocols of the NLGR system are entirely decentralised: there needs to be no central registry of any information in the system. This ensures that the system scales smoothly and reduces the incidence of bottlenecks.

The NLGR system is intended as the target of a compiler. The PhD thesis of Alan Bawden describes the implementation of a compiler from a dialect of Scheme (essentially R4RS without tests for storage equality). Compiling from such a language has a serious defect, namely that since the language is sequential, there are rarely opportunities to exploit much of the parallelism available in the runtime. To remedy this an explicit concurrency construct, namely the `future` construct, was added.

Rather than expecting the programmer to explicitly direct the compiler, it would be desirable to use a source language that naturally offers more opportunities for parallelism to be exploited by the compiler. At Boston University there is an ongoing effort to make use of the increased opportunities for parallelism available in a lazy language. However graph transformation languages offer even greater opportunities for concurrent execution, and so it is natural to use this language as our source language.

The proposed compiler architecture

The proposed compiler shall take the familiar staged design. The programmer's source code is given in the AGG language, which is an attributed graph transformation language based upon the single pushout approach, and which allows the specification of negative application conditions (NACs) to constrain the applicability of patterns and the use of Java objects as attributes and Java methods in the calculation of replacement graphs.

Patterns in the AGG language may be disconnected, that is either the pattern graph may be disconnected or the NAC graph may not share any gluing points with the main pattern. Also, rules in an AGG program may be stratified, that is one may specify an order in which rules are applicable. Both of these features of the AGG language are extremely bad when we try to exploit concurrency in the NLGR system, since to ensure that a rule is applicable requires inspecting the whole graph. So before we can transform our AGG program into code executable on the NLGR system we must eliminate these features. We achieve this by a transformation of our AGG program into a restricted form, which we might call *connected AGG*.

The heart of the compiler is the transformation of the connected AGG program into an attributed linear graph grammar. The mechanism by which this transformation will be achieved has not yet been defined; however the proposed solution strategy has been shown to be effective for a simpler problem, the reducibility of static-port graph grammars to linear graph grammars.

With these two transformations defined, we can now give the full stages for the compiler. We introduce optimisation stages by applying partial evaluation at each of the three representation forms.

1. Partial evaluation of AGG source to obtain simplified AGG program.
2. Transformation of AGG program to connected AGG form.
3. Partial evaluation of connected AGG program.
4. Transformation of connected AGG form to attributed LGG form.
5. Partial evaluation of attributed LGG form.
6. Generation of NLGR runtime.

References

- [1] Uwe Assman. Graph rewrite systems for program optimisation. In *ACM Trans. Programming Languages and Systems*, 22(4):583–637, 2000.
- [2] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation II: single pushout approach and comparison to double pushout approach. In *Volume I: Foundations of the Handbook of Graph Grammars and Computing by Graph Transformations*, edited G. Rozenberg. World Scientific, 1997.
- [3] Alan Bawden. Connection Graphs. In *Proc. Conf. on LISP and Functional Programming*, pages 258–265. ACM Press, 1986.
- [4] Alan Bawden. *Implementing Distributed Systems Using Linear Naming..* PhD thesis, MIT, 1993.
- [5] Yves Lafont. Interaction nets. In *Proc. Annual 17th Symp. Principles of Programming Languages*, pages 95–108. ACM Press, 1990.
- [6] C. A. Stewart. Reducibility between classes of static-port graph grammar. Submitted to *Journal of System and Computer Science*, 2001.
- [7] G. Taentzer, I. Fischer, M. Koch, V. Volle. Visual design of distributed systems by graph transformation. In *Volume III:Concurrency, Parallelism and Distribution of the Handbook of Graph Grammars and Computing by Graph Transformations*, edited by G. Rozenberg. World Scientific, 1997.