The GP Programming System

Greg Manning and Detlef Plump

The University of York

Abstract: We describe the programming system for the graph-transformation language GP, focusing on the implementation of its compiler and abstract machine. We also compare the system's performance with other graph-transformation systems. The GP language is based on conditional rule schemata and comes with a simple formal semantics which maps input graphs to sets of output graphs. The implementation faithfully matches the semantics by using backtracking and allowing to compute all possible results for a given input.

Keywords: GP, programming system, graph transformation, non-determinism

1 Introduction

GP is a non-deterministic graph programming language based on conditional rule schemata in the double-pushout approach [PS04]. The core of GP consists of just four constructs: single-step application of a set of rule schemata, sequential composition, branching and iteration. The language is computationally complete [HP01] and comes with a formal semantics [PS08]. The current implementation of GP consists of a graphical editor for programs and graphs, a compiler and the York Abstract Machine (YAM). These components communicate as shown in Figure 1 (where YAMG is an internal graph format of the abstract machine).

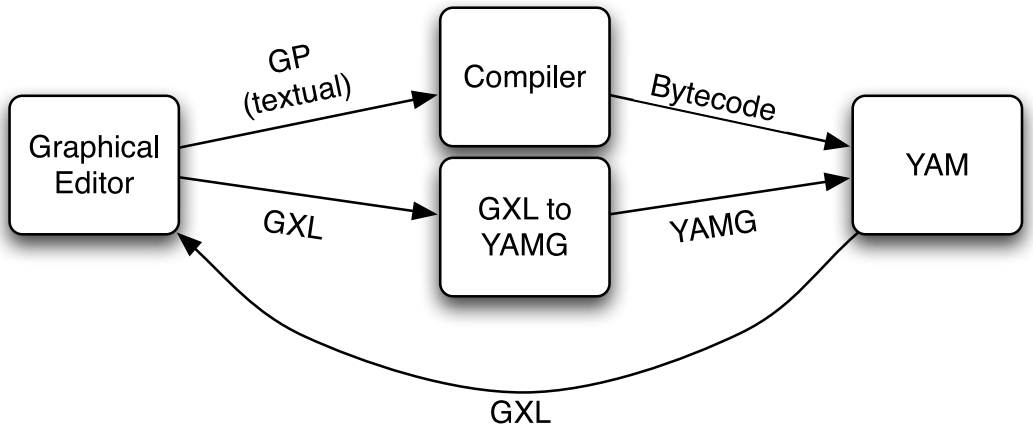

Figure 1: An overview of the GP system

We describe GP by means of an example. Consider the program `minimum_spanning_tree`

in Figure 2. This program calculates a minimum spanning tree for its input graph.[1] The program

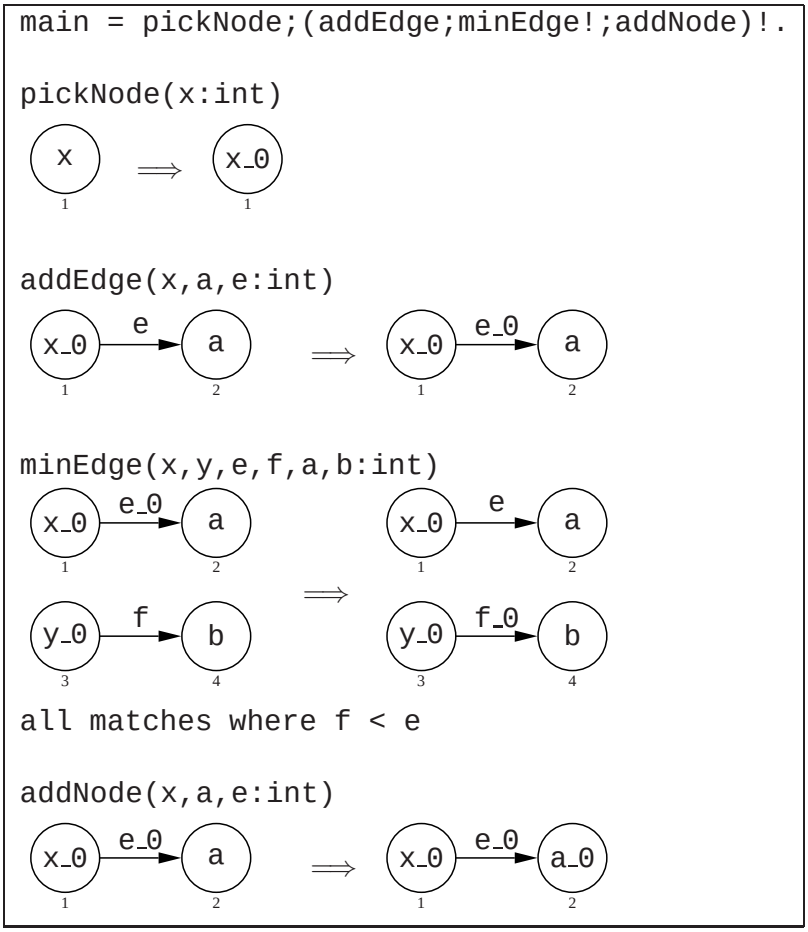

Figure 2: GP program `minimum_spanning_tree`

consists of four rule-schema declarations and the main command sequence following the key word `main`. Given an input graph whose nodes and edges are labelled with integers, the program first uses the rule schema `pickNode` to choose any node and replace its label x with x_0. The underscore operator allows to add a *tag* to a label, where in general a tagged label consists of a sequence of expressions joined by underscores. (Sequences of expressions are just ordinary labels, allowing GP's underlying theory to be based on a standard variant of the double-pushout approach rather than on some complicated model of attributed graph transformation.) This program uses the tag 0 to mark the nodes of a spanning tree. After the initial node has been marked, the iteration operator '!' executes the subprogram `(addEdge;minEdge!;addNode)` as long as possible. The subprogram first picks any edge between a marked node and an unmarked node. Then the loop `minEdge!` repeatedly swaps this edge with an edge having a smaller label, where

[1] A spanning tree for a directed graph G is a subgraph S of G such that the undirected graph underlying S is a spanning tree for the undirected graph underlying G.

the latter is checked by the condition `where f < e`. The flag `all matches` allows this rule schema to be matched non-injectively whereas the default in GP is injective matching. After the minimum edge between the current tree nodes and any unmarked node has been determined, the unmarked node of this edge is added to the spanning tree by the rule schema `addNode`. It is not difficult to see that upon termination of the outer loop, the marked nodes and edges constitute a minimum spanning tree of the input graph. (The rule schemata `addEdge`, `minEdge` and `addNode` are actually sets of rule schemata which are obtained from those depicted by reversing edges in all possible ways: `addEdge` and `addNode` consist of two rule schemata while `minEdge` contains four schemata. For readability, we have omitted these rule schemata in Figure 2.) In general, a graph can have several minimum spanning trees and the program in Figure 2 allows to compute all of them.

A leitmotiv for GP's design has been syntactic and semantic simplicity, see also [PS08]. There is only one other core construct besides those occurring in the minimum spanning tree program: a conditional statement of the form `if` *C* `then` *P* `else` *Q* (where *C*, *P* and *Q* are programs). Our programming experience so far suggests that these few constructs are sufficient and allow succinct solutions to problems. It is possible though to simulate more elaborate control mechanisms. Consider, for example, a conditional loop of the form `while` *C* `do` *P* which executes its body *P* as long as the program *C* succeeds. An equivalent GP program is (`if` *C* `then` *P* `else fail`)!; `if` *C* `then fail` (where `fail` is an always failing program such as the empty set of rules). As another example, the choice to apply a rule *r* either once or not at all can be simulated by the rule set $\{r, \emptyset \Rightarrow \emptyset\}$ (where $\emptyset \Rightarrow \emptyset$ has the empty graph on both sides).

The rest of this paper is organized as follows. The next section briefly adresses the graphical user interface of the GP system, Section 3 introduces the York abstract machine and Section 4 discusses the GP compiler. Section 5 compares the performance of the GP system with other graph-transformation environments. In Section 6, we conclude and give some topics for future work.

2 Graphical Editor

The GP graphical editing environment is a Java application which allows graph and program creation, loading, editing and saving, and program execution on a given graph. The outputs of executions are then available as inputs to other programs. Figure 3 shows a screenshot of the graphical editor with the rule `minEdge` of Figure 2 being edited. The editor visualises graphs using the prefuse data visualisation library [HCL05], which permits graph layout and editing. The main graph drawing algorithm used is a force-directed layout. Figure 5 shows a graph drawn by this algorithm.

3 The York Abstract Machine

The York abstract machine (YAM) is more fully described in [MP06]. Here, we give an overview highlighting the areas which have changed in the meantime.

The YAM is a backtracking graph-transformation machine which executes bytecode for low-level graph operations. It can handle nondeterministic programs and is in parts similar in de-

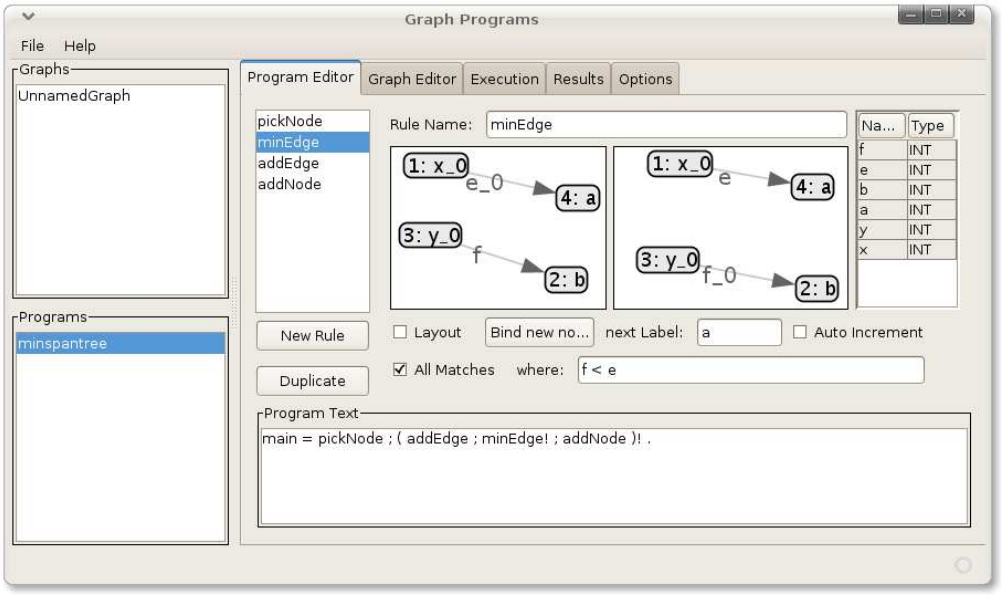

Figure 3: A screenshot of the graphical editor

sign to Warren's abstract machine for Prolog [AK91]: it manages GP's nondeterminism using a mixed stack of choice points and environment frames. The implementation of backtracking in PROGRES [Zün92] takes a WAM-like approach too, although it uses the host language's call stack rather than explicit data structures. The YAM also manages the current host graph and a (typically) small data stack.

Figure 4 shows an example state of the choice point and environment frame stack. Choice points consist of a record of the number of graph changes at their creation time, a program position to jump to if failure occurs when the choice point is the highest on the stack, and pointers to the previous choice and containing environment. The number of graph changes is recorded so that, when backtracking, the graph changes can be undone: using the stack of graph changes, the graph as it was at the choice point is recreated. Environment frames have a set of registers to store label elements or graph element identities, and an associated function and program position in the bytecode. They also show which environment and program position to return to. The number of registers each frame has is determined by the bytecode — it is fixed at compile time.

The current host graph is stored in a complex data structure, making use of the heavily optimised Judy data structures [Siv02]. The structure is designed in such a way that the graph can be interrogated easily and very quickly, at the cost of slightly slower graph updates. Typical queries to the graph structure are "edges whose target node is node *n*" or "nodes whose label has the value 1 in position 1". Each element (node or edge) in the graph is labelled with a list of values, each of which is of type integer or string. The YAM bytecode allows any query over the length of the list or the type or value of the list elements, such as "all nodes with a label of size two", or "all edges with an integer in the second position."

The machine as presented in [MP06] handled nondeterminism internally. At the bytecode

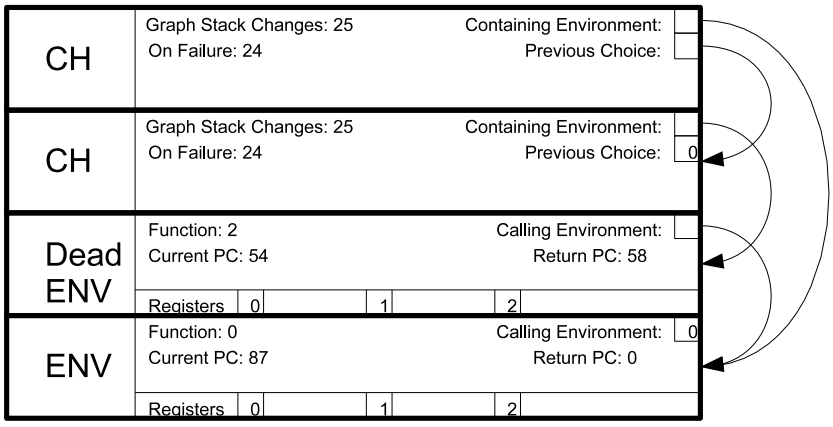

Figure 4: An example choice/environment stack

level, the instructions effectively returned a correct result: if a choice led to a failure (and such was a wrong choice), then the machine would trigger backtracking and retry the choice until a correct result was obtained. Now, however, the machine simply provides explicit instructions for handling nondeterminism such as `OnFail`[2], `UpdateFail`[3] and `Assert`[4]. This change was made to unite the failing and non-failing versions of the graph queries, and to allow more expressiveness at the bytecode level.

Using these instructions, the compiler constructs helper functions to implement backtracking. Nondeterministic choice between a set of graph rules is handled by trying them in textual order until one succeeds. Before each is tried, the failure behaviour is configured to try the next. Nondeterministic choice between graph-element candidates for a match is handled by choosing and saving the first element, and on failure, using the saved previous answer to return (and save) the next element.

Nodes and edges are identified in the structure by integers, and the graph structure contains many ordered lists of such integers. This allows complex conjunctive queries to be performed by intersecting ordered lists of integers. For example, in finding the left-hand side of the rule `addEdge` in the program of Figure 2, having found the `x_0` node, a list of potential edges is created by intersecting the list of all edges leaving this node, the list of all edges which have `e` as their first label, and the list of all edges having a label sequence of length one. The code then creates an environment to store the previous answer returned, and uses the `Next` instruction to give the first answer in the intersection which is numerically greater than the previous answer. It saves this answer to the stack and returns it. If a failure occurs whilst this choice point is on top of the stack, the code will return the next answer. When there are no more answers it will propagate the failure to a previous choice point.

[2] `OnFail` pops a code location and creates a new choice point which will jump to that code location on failure.
[3] `UpdateFail` pops a choice point pointer and a code location and changes the choice point so that it now goes to the new location on failure.
[4] `Assert` pops the top of stack and fails if it is zero.

Because the underlying data structure stores the node and edge references in the lists as ordered lists of integers, finding the next element in the intersection is very fast. An intersection can be done in time $O(ln)$, where l is the length of the shortest list in the intersection and n is the number of lists being intersected. Note that the entire intersection is not generated in one go, the elements of the intersection are found one at a time, as needed.

4 Compiler

The GP compiler converts textual GP programs into YAM bytecode. It does this by translating each individual rule or macro into a sequence of instructions, and composing these sequences using the YAM function calls.

4.1 Generating a searchplan for graph matching

Searchplan generation is a common technique for implementing graph matching [GBG+06, HVV07, Zün96]. The GP compiler decomposes graph rules into a static searchplan of node lookups, edge lookups (find an edge whose source and target have not been found yet) and extensions (find an edge whose source or target has been found). The choice and order of these search operations is determined using the following priorities, always preferring elements with value labels over those with variable labels:

1. Check parts of `where` clauses whose variables have all been bound, that is, all label variables have been instantiated, and all nodes or edges referred to have been found.
2. Find nodes on the ends of edges which have been found.
3. Find edges where both the start and the end node have been found.
4. Find edges where either the start or the end node has been found.
5. Find nodes where there is a negative edge condition of the form `not edge(`v`,`w`)` at the top level of the `where` clause[5] and either v or w has been found.
6. Find nodes.

The nondeterminism in this list of priorities increases: where clause checks and finding nodes of known edges are deterministic operations, finding unrestricted nodes is highly nondeterministic. There will be many different plans which satisfy these priorities, however since the compiler generates static plans (that is, the host graph is not interrogated), there is no more information to use in the generation. The choice between possible plans is made using an ordering taken from the programmer: elements mentioned first in the textual input program will be found first. For example, the first step in a searchplan is to find the first (labelled) node mentioned.

Both GrGen [GBG+06] and Fujaba [NNZ00] also make extensive use of searchplans. GrGen.NET [BG07] uses online searchplan generation, so that the searchplans can be recalculated during execution. This improves the quality of the generated searchplan significantly.

[5] That is, the negative edge condition is one of the conjuncts $C_1,\ldots,C_n$ in `where` C_1 `and` $\ldots$ `and` C_n.

Once a match has been found, and the `where` clause has passed, the remainder of the code for the graph rule handles the changes to be made to the graph. The compiler determines the changes and orders them as follows: deleted edges, deleted nodes, relabelled nodes or edges, added nodes, added edges. This order ensures that no nodes are deleted before their incident edges, and no edges are created before their incident nodes.

4.2 Compiling GP commands

With the individual graph rules compiled, they can be composed into a complete YAM program. There are several ways of joining subprograms in GP: sequential composition, macro calling, if-then-else branching, and as-long-as-possible iteration. The compilation of a sequential composition $P;Q$ is trivial: the bytecodes for P and Q are concatenated. Macro calling is achieved by using the `Call` or `TailCall`[6] bytecode instructions.

As-long-as-possible iteration, $P!$, is implemented as follows:

1. Create new failure behaviour to succeed (continue) on failure.
2. Execute P once.
3. Change failure from instruction 1 from succeed to fail. Having a failure behaviour which simply fails is the same as having no failure behaviour at all; however, failure behaviours cannot be removed since they may be referenced elsewhere in the stack.
4. Go to instruction 1.

The failure behaviour must be altered in step 3 to maintain the semantics and not an any-number-of-times semantics.

As rule sets (apply one rule from a set) are compiled, an ordering is imposed upon them. The compiled bytecode tries the first rule, and on failure will try the next rule until the end of the set is reached. If none of the rules successfully applied, then the whole rule set fails. The ordering imposed is the order in which the rules appear in the program text.

GP's branching construct `if` C `then` P `else` Q first executes the subprogram C on the input graph. If this yields a result, program P is executed *on the input graph*. Otherwise, if all executions of C end in failure, program Q is executed on the input graph. The construct is compiled in the following way:

1. Create failure behaviour to go to step 5 on failure.
2. Execute the condition C.
3. `ClearFail` the failure point created in step 1, that is, undo the graph changes, forget the choices back to that point and remove that failure frame, but leave the program pointer unchanged.
4. Execute the then-part P and succeed.
5. Execute the else-part Q and succeed.

[6] `TailCall` is equivalent to call-then-return, but is actually implemented as return-then-call because this saves space on the call stack.

5 Performance

In this section we compare the performance of the GP system with the performance of similar environments. We focus on a simple problem which has been implemented in different graph programming systems in the context of the AGTIVE 2007 tool contest [TBB+08]. The task is to generate a graph of the *n*th generation of the Sierpinski triangle, producing one generation at a time. A Sierpinski triangle is a triangle split into 4 subtriangles (made by joining the midpoints of the 3 edges), where the 3 subtriangles containing one of the original vertices are themselves Sierpinski triangles. Sierpinski triangles are represented as graphs using nodes as vertices of triangles and edges as edges of triangles. As a true Sierpinski triangle has infinite detail, we must generate an approximation. We say that the *n*th generation of a Sierpinski triangle is one which has a depth of *n*. The 0th generation Sierpinski triangle is a simple triangle. Figure 5 shows the 4th generation Sierpinski triangle.

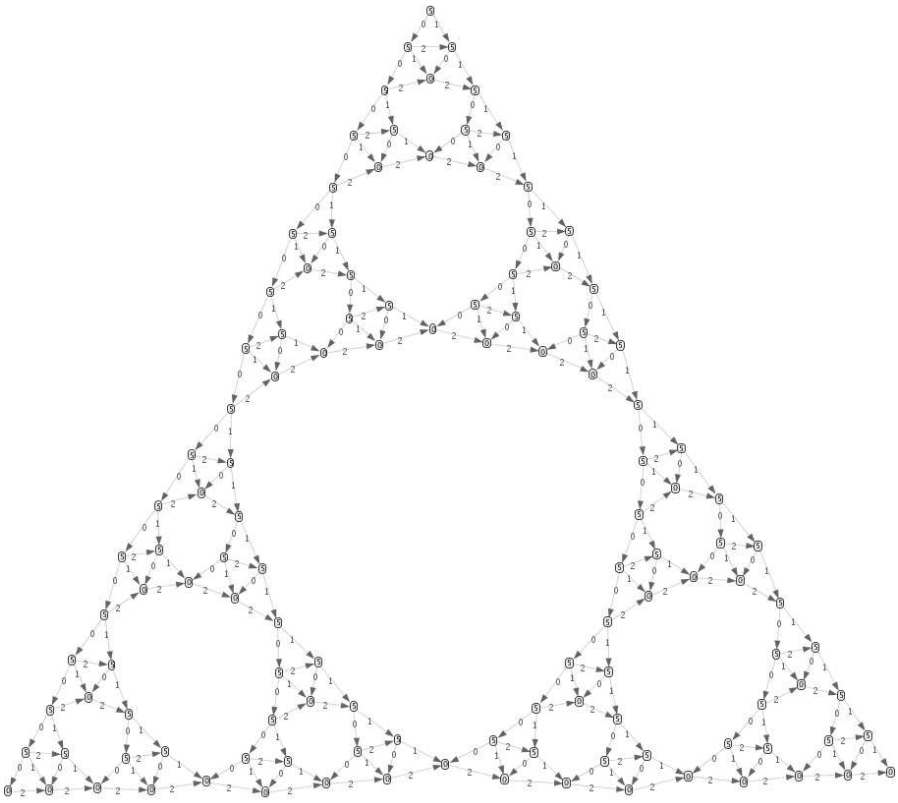

Figure 5: A 4th generation Sierpinski triangle

5.1 Generating Sierpinski triangles with GP

The GP program `sierpinski` is presented in Figure 6. It expects as input a graph consisting of a single node labelled with the generation number of the Sierpinski triangle to be produced. The rule schema `init` creates the initial Sierpinski triangle (generation 0) and turns the input node into a unique "control node" whose label is of the form `x_y`. The underscore operator is used here to hold the required generation number `x` and the current generation number `y` in a single node.

After `init` has been applied, the nested loop `(inc; expand!)!` is executed. In each iteration of the outer loop, the rule schema `inc` increases the current generation number if it is smaller than the required number. The latter is checked by the condition `where x > y`. If the test is successful, the inner loop `expand!` performs a Sierpinski step on each triangle whose root[7] is labelled with the current generation number: the triangle is replaced by four triangles such that the roots of the three outer triangles are labelled with the next higher generation number. The test `x > y` fails when the required generation number has been reached. In this case the application of `inc` fails and, as a consequence, the outer loop terminates and returns the current graph. It is not difficult to see that the resulting graph is indeed the Sierpinski triangle of the required generation.

5.2 Comparison with other systems

In Figure 7 we present the execution times for the GP system and some other graph transformation systems that participated in the Sierpinski tool contest. The times for GP were obtained on a PC with an Intel Pentium 4 processor with a clock rate of 2.8GHz and 512MB of main memory. The times for the other systems were obtained on comparable machines. Our figure includes only a subset of the tools described in [TBB+08]. We have omitted tools that are tailored for parallel rule applications in specialised areas but cannot be considered as general-purpose graph transformation tools.

As Figure 7 demonstrates, GP is faster than five other systems and is beaten only by GrGen.NET and Fujaba. GrGen.NET requires the programmer to specify types of node and edges (often hierarchical types with multiple inheritance). The information gained from these types gives more information to the graph matching algorithm and also allows better compilations. GP has very little typing, freeing the programmer from specifying these overarching types. This allows shorter, more succinct programs at the cost of some speed. However, as demonstrated by this benchmark, the speed lost is not too great.

5.3 Non-deterministic programs

Other graph programming systems do not fully exploit the non-deterministic nature of graph transformation rules. The semantics of GP programs on input graphs are *all* possible output graphs, and this is taken seriously by the implementation in that it provides users with the option to generate several or even all possible results. This mechanism is complete for terminating

[7] The *root* of a triangle is the unique node (if it exists) from which a 0-edge and a 1-edge is outgoing. Note that the inner triangle on the right-hand side of `expand` does not have a root, hence it will never be expanded.

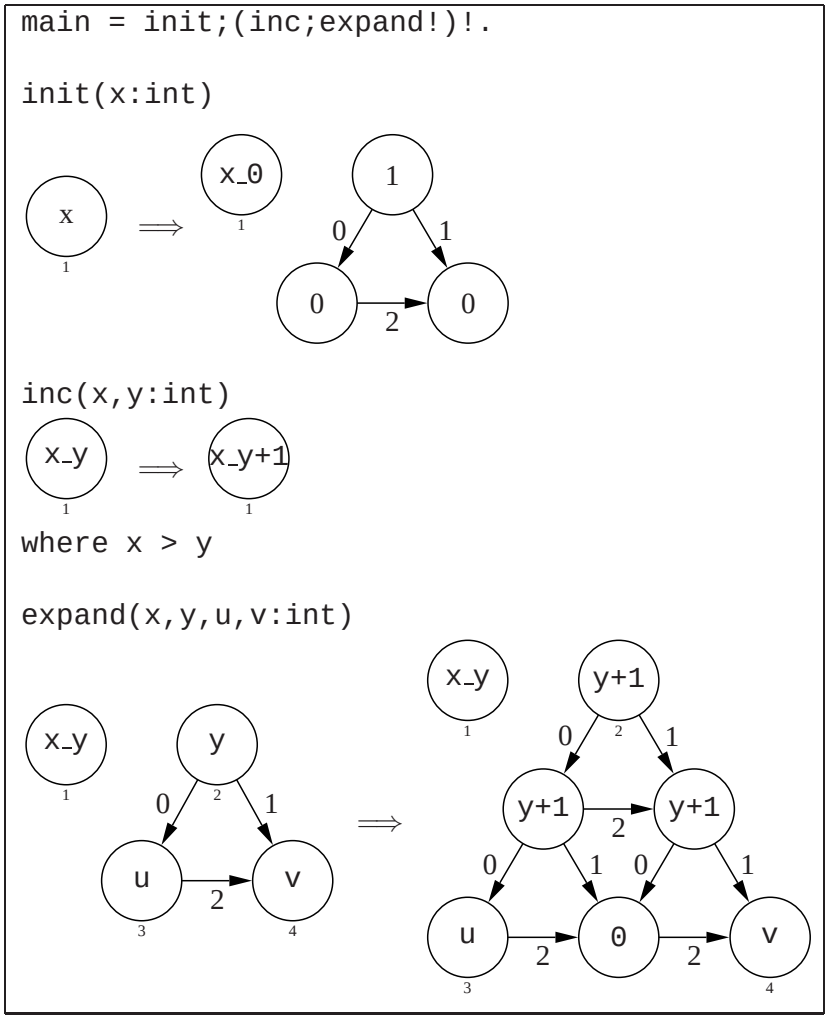

Figure 6: The Program `sierpinski`

programs. In contrast, AGG [ERT99] makes its nondeterministic choices randomly, with no backtracking. Similarly, Fujaba has no backtracking. It seems that PROGRES [SWZ99] is the only other graph transformation language in use that provides backtracking.

The Sierpinski example presented above is a deterministic problem. That is, the program `sierpinski` computes a function where each input graph produces a single output graph. Although in the GP implementation there is a choice of which order to convert subtriangles to the next generation, since they will all get done eventually this is a *confluent* program in that all output graphs are isomorphic. This is not always the case. For example, the program `minimum_spanning_tree` presented in the Introduction is non-confluent: for an input graph, there is not necessarily a unique minimum spanning tree. The implementation of GP respects the semantics, and allows computation of all possible minimum spanning trees. The use of

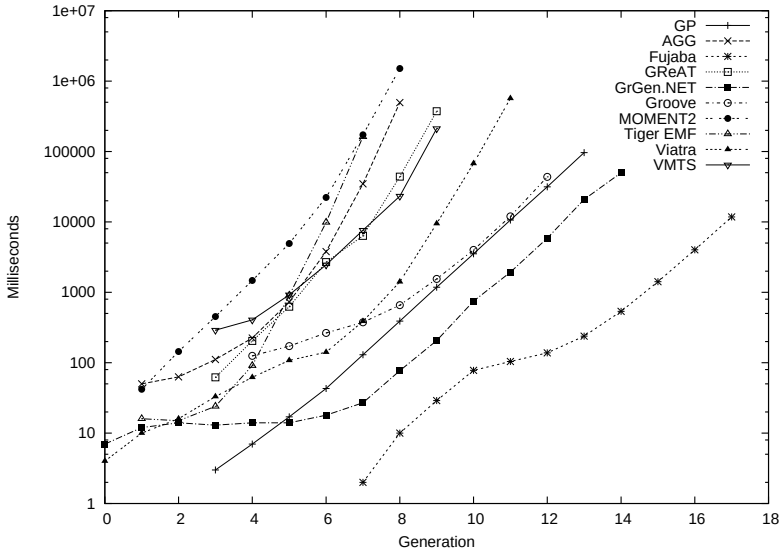

Figure 7: Execution times for the Sierpinski benchmark

this non-determinism is not limited to finding multiple answers. It is possible, and common, to write programs which rely on the backtracking behaviour to filter results, or make correct non-deterministic guesses. Most other graph transformation systems do not allow such programs. Neither AGG nor Fujaba allow backtracking over graph rules. The GrGen.NET API supplies tools necessary to perform backtracking (such as presenting all possible matches of a rule), but the GrShell example environment [BG07] does not allow backtracking in this manner.

Using nondeterminism to this extent is sometimes problematic. In the Sierpinski example, the order in which the matches of the `expand` rule are applied makes no difference, yet if later in the program there was a failure, the backtracking mechanism would try all possible different orders of the matches. It is an item of future work to develop analysis techniques to detect and disable the backtracking in such cases (see also the remarks in the next section). Since no backtracking is required in the Sierpinski example, our solution had the backtracking mechanism of the YAM disabled.

6 Conclusion and Future Work

The GP implementation matches faithfully GP's semantics and allows to compute all results of a (terminating) program. GP is a small clean language, with enough structure so that it is usable, but little enough that the semantics is understandable and useable for arguments and proofs [PS08]. The system is reasonably fast; slow execution is usually caused by a vast nondeterministic search space which can often be avoided by programming carefully.

The YAM can give more than one answer, or all answers. It provides a clearly defined separation between runtime and compile time actions. Whilst the YAM has been designed as part

of the GP system, it is by no means restricted to it; other graph systems and semantics could be realised using the bytecode provided by the YAM.

In the current implementation of GP, only one match of one rule is executed at a time. Other graph programming systems can execute multiple rules or multiple matches in parallel, which can give large speed gains in certain situations. The GP system does not currently do this because it involves a considerable amount of checking that all the matches can be successfully executed without interfering with each other, and would make the bookkeeping for backtracking very complex.

The GP system generates static searchplans at compile time, so no host-graph interrogation is possible. With runtime searchplan generation (as in GrGen.NET [GBG+06]), it is possible to always match the rarest elements first, which reduces the search space to find a match.

As GP programs get larger, it may be useful to include optional conservative static type checking. This may be implemented as graph metamodels or more complex typing systems such as the GRS types of [BPR04]. By analysing programs, it will sometimes be possible to guarantee that certain graph structures do or do not occur.

In many cases, nondeterministic (sub)programs are confluent: they cannot possibly fail, and all solutions are isomorphic. Using static analysis techniques such as critical-pair analysis [Plu05], it will sometimes be possible to detect these situations. This is useful information in itself, but can also be used to speed up the implementation, since backtracking would not be required through a confluent section of a program.

Bibliography

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[BG07] J. Blomer, R. Geiß. The GrGen.NET User Manual. Technical report 2007-5, Universität Karlsruhe, IPD Goos, July 2007. http://www.info.uni-karlsruhe.de/papers/TR_2007_5.pdf

[BPR04] A. Bakewell, D. Plump, C. Runciman. Specifying Pointer Structures by Graph Reduction. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*. Lecture Notes in Computer Science 3062, pp. 30–44. Springer-Verlag, 2004.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. The AGG Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2, chapter 14, pp. 551–603. World Scientific, 1999.

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. Graph Transformations (ICGT 2006)*. Lecture Notes in Computer Science 4178, pp. 383–397. Springer-Verlag, 2006.

[HCL05] J. Heer, S. K. Card, J. A. Landay. Prefuse: A Toolkit for Interactive Information Visualization. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI 2005)*. Pp. 421–430. ACM Press, 2005.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*. Lecture Notes in Computer Science 2030, pp. 230–245. Springer-Verlag, 2001.

[HVV07] Á. Hórvath, G. Varró, D. Varró. Generic Search Plans for Matching Advanced Graph Patterns. In *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Electronic Communications of the EASST 6. 2007.

[MP06] G. Manning, D. Plump. The York Abstract Machine. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006)*. Electronic Notes in Theoretical Computer Science. Elsevier, 2006. To appear.

[NNZ00] U. Nickel, J. Niere, A. Zündorf. The FUJABA Environment. In *Proc. Software Engineering (ICSE 2000)*. Pp. 742–745. ACM Press, 2000.

[Plu05] D. Plump. Confluence of Graph Transformation Revisited. In Middeldorp et al. (eds.), *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science 3838, pp. 280–308. Springer-Verlag, 2005.

[PS04] D. Plump, S. Steinert. Towards Graph Programs for Graph Algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*. Lecture Notes in Computer Science 3256, pp. 128–143. Springer-Verlag, 2004.

[PS08] D. Plump, S. Steinert. A Structural Operational Semantics for GP. 2008. In preparation.

[Siv02] A. Siverstein. Judy IV Shop Manual. 2002. http://judy.sourceforge.net.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2, chapter 13, pp. 487–550. World Scientific, 1999.

[TBB+08] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, Á. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, T. Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007), Revised Selected and Invited Papers*. Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[Zün92] A. Zündorf. Implementation of the imperative/rule-based language PROGRES. Technical report 92-38, Fachgruppe Informatik, RWTH Aachen, 1992. http://citeseer.ist.psu.edu/albert92implementation.html

[Zün96] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 1073, pp. 454–468. Springer-Verlag, 1996.