Graph Programs

Detlef Plump
The University of York, UK

1 Introduction

This paper gives a brief introduction to GP (for Graph Programs), an experimental nondeterministic programming language for high-level problem solving in the domain of graphs. The language is based on conditional rule schemata for graph transformation and thereby frees programmers from handling low-level data structures for graphs. The prototype implementation of GP compiles graph programs into bytecode for the York abstract machine, and comes with a graphical editor for programs and graphs [8].

Graph programs have a simple syntax the core of which contains only four commands: single-step application of a set of rule schemata, sequential composition, branching and as-long-as-possible iteration. Despite their simplicity, graph programs are computationally complete in that every computable function on graphs can be programmed [6]. A major goal of the GP project is to develop a practical graph-transformation language that comes with a concise formal semantics, to facilitate program verification and other formal reasoning on programs.

A special feature of GP's formal semantics is the use of failing computations to define powerful branching and iteration constructs. (Failure occurs when a set of rule schemata to be executed is not applicable to the current graph.) The evaluation of a condition C of a branching command succeeds if there exists an execution of C on the current graph that produces a graph. If all executions of C result in failure, we say that C *finitely fails* on the current graph. Finite failure is also used to define as-long-as-possible iteration of arbitrary programs: a loop terminates if its body finitely fails on the current graph. The concept of finite failure stems from logic programming where it is used to define negation [3].

Control constructs which allow programmers to write strategies for applying rewrite rules have long been present in term-rewriting languages such as Elan [1] and Stratego [2]. These languages allow recursive definitions of strategies whereas GP is based on a small set of built-in, non-recursive constructs. (See [15] for an extension of GP with recursive procedures.) The distinguishing features of GP can be summarised as follows:

- A graph-transformation language with *simple* syntax and semantics, facilitating understanding by programmers and formal reasoning on programs. Our experience so far is that very often short and easy to understand programs can be written to solve problems on graphs.

- The first formal operational semantics for a graph-transformation language (to the best of our knowledge). Well-known languages such as AGG [4], Fujaba [9] and GrGen [5] have no formal semantics. The only graph-transformation language with a complete formal semantics that we are aware of is PROGRES [14]. Its semantics, given by Schürr in his dissertation, translates programs into control-flow diagrams and consists of more than 300 rules (including the definition of the static semantics) .

- A powerful branching construct based on the concept of finite failure, allowing to conveniently express complex destructive tests on input graphs. In addition, finite failure enables an elegant definition of as-long-as-possible iteration. These definitions do not depend on the structure of graphs and can be used for string- or term-based rewriting languages, too.

This introductory paper is based on [12, 13].

2 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs: a program is essentially a list of declarations of conditional rule schemata together with a command sequence for controlling the application of the schemata. Rule schemata generalise graph-transformation rules in the double-pushout approach with relabelling [7], in that labels can contain expressions over parameters of type integer or string. Figure 1 shows an example for the declaration of a conditional rule schema. It consists of the identifier `bridge` followed by the declaration of formal parameters, the left and right graphs of the schema which are labelled with arithmetic expressions over the parameters, the node identifiers `1`, `2`, `3` specifying which nodes are preserved, and the keyword `where` followed by the condition.

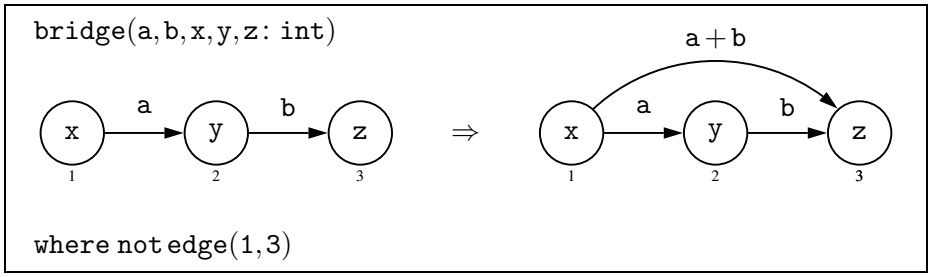


Figure 1: A conditional rule schema

In the GP programming system [8], rule schemata are constructed with a graphical editor. Labels in the left graph must be variables or constants because their values at execution time are determined by graph matching. The condition of a rule schema is a boolean expression built from arithmetic expressions and the special predicate `edge`, where all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1,3)` in the condition of Figure 1 forbids an edge from node 1 to node 3 when the left graph is matched.

Conditional rule schemata represent possibly infinite sets of conditional graph-transformation rules which are obtained by instantiating variables with integers and evaluating expressions. The resulting rules are applied according to the double-pushout approach with relabelling [7].

3 Example Programs

We discuss two example programs to familiarize the reader with GP's features.

Example 1 (Colouring). A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each edge have different colours. The program `colouring` in Figure 1 produces a colouring for every integer-labelled input graph, recording colours as so-called tags. In general, a tagged label is a sequence of expressions separated by underscores.

The program initially colours each node with 1 by applying the rule schema `init` as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is highly nondeterministic: Figure 1 shows two different colourings produced for the same input graph, where one is optimal in that it uses only two colours while the other uses three colours.

It is easy to see that whenever `colouring` terminates, the resulting graph is a correctly coloured version of the input graph. For, the output cannot contain an edge with the same colour at both nodes as

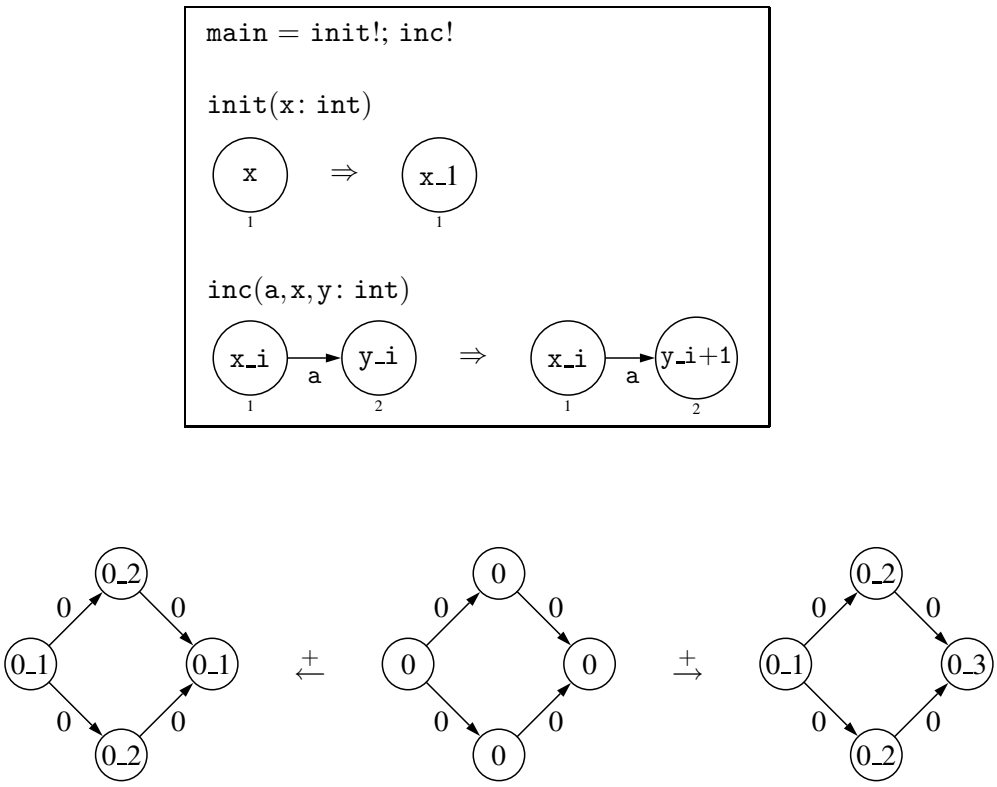


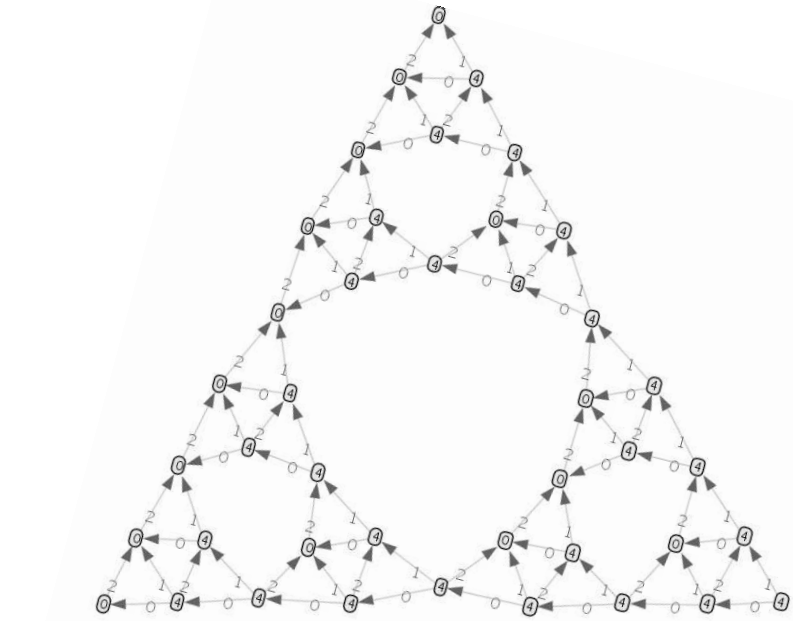
Figure 2: The program `colouring` and two of its executions

then `inc` would have been applied at least one more time. Also, it can be shown that every execution of the program terminates after at most a quadratic number of rule schema applications [12].

Example 2 (Sierpinski triangles). A *Sierpinski triangle* is a self-similar geometric structure which can be recursively defined [10]. Figure 3 shows a Sierpinski triangle of generation three, composed of three second-generation triangles, each of which consists of three triangles of generation one. The triangle and its geometric layout have been generated with the GP programming system [16].

The program in Figure 3 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 Sierpinski triangle of generation 0 and turns the input node into a unique "control node" with the tagged label x_0 in order to hold the required generation number x together with the current generation number.

After initialisation, the nested loop (`inc`; `expand!`)! is executed. In each iteration of the outer loop, `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 top node is labelled with the current generation number: the triangle is replaced by four triangles such that the top nodes 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, causing the outer loop to terminate and return the current graph which is the Sierpinski triangle of the requested generation.



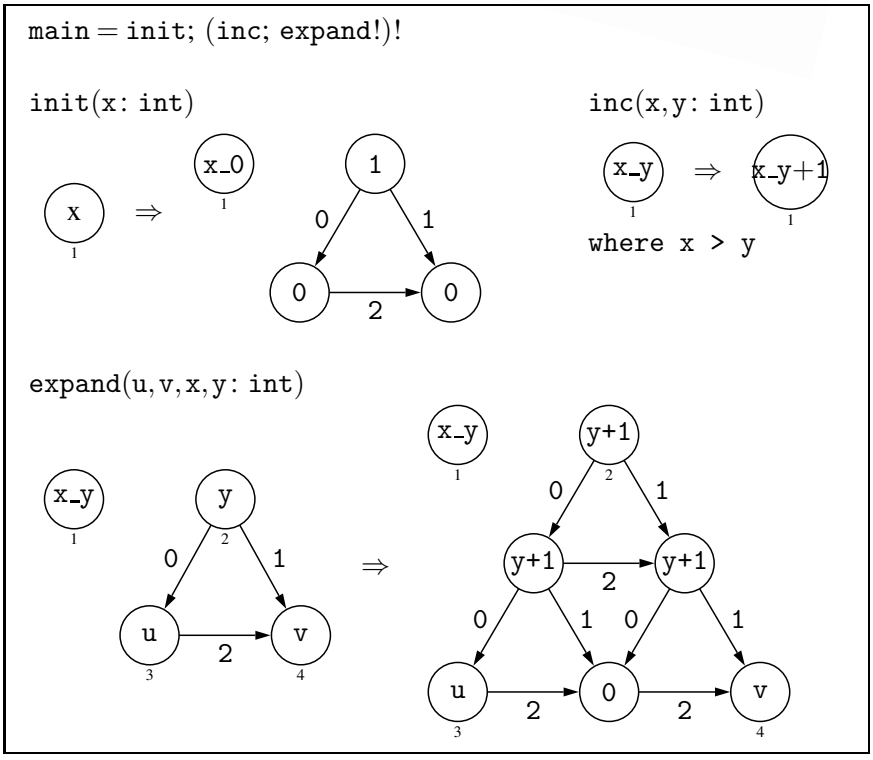


Figure 3: A Sierpinski triangle (third generation) and the program `sierpinski`

4 Operational Semantics

GP's formal semantics is given in the style of Plotkin's structural operational semantics [11]. As usual for this approach, inference rules inductively define a small-step transition relation $\rightarrow$ on configurations. In our setting, a configuration is either a program together with a graph, just a graph or the special element fail. Configurations that are pairs represent unfinished computations, given by a rest program and a state in the form of a graph, while graphs are proper results of computations. In addition, the element fail represents a failure state.

In the inference rules in Figure 4, R stands for a set of conditional rule schemata, C, P, P', Q stand for arbitrary programs and G, H stand for graphs. The single-step derivation relation of R is denoted by $\Rightarrow_R$. Its domain $\mathrm{Dom}(\Rightarrow_R)$ is the set of all graphs G such that $G \Rightarrow_R H$ for some graph H.

Figure 4 shows the inference rules for the core constructs of GP. We write $\rightarrow^+$ and $\rightarrow^*$ for the transitive and reflexive-transitive closures of $\rightarrow$. A command sequence C *finitely fails* on a graph G if (1) there does not exist an infinite sequence $\langle C, G\rangle \rightarrow \langle C_1, G_1\rangle \rightarrow \ldots$ and (2) for each terminal configuration γ such that $\langle C, G\rangle \rightarrow^* \gamma$, $\gamma = \text{fail}$. (A configuration γ is terminal if there is no configuration δ such that $\gamma \rightarrow \delta$.) In other words, C finitely fails on G if all computations starting from (C, G) eventually end in the configuration fail.

$$[\mathrm{Call}_1]\ \frac{G \Rightarrow_R H}{\langle R, G\rangle \rightarrow H} \qquad [\mathrm{Call}_2]\ \frac{G \notin \mathrm{Dom}(\Rightarrow_R)}{\langle R, G\rangle \rightarrow \text{fail}}$$

$$[\mathrm{Seq}_1]\ \frac{\langle P, G\rangle \rightarrow \langle P', H\rangle}{\langle P;Q, G\rangle \rightarrow \langle P';Q, H\rangle} \qquad [\mathrm{Seq}_2]\ \frac{\langle P, G\rangle \rightarrow H}{\langle P;Q, G\rangle \rightarrow \langle Q, H\rangle}$$

$$[\mathrm{Seq}_3]\ \frac{\langle P, G\rangle \rightarrow \text{fail}}{\langle P;Q, G\rangle \rightarrow \text{fail}}$$

$$[\mathrm{If}_1]\ \frac{\langle C, G\rangle \rightarrow^+ H}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q, G\rangle \rightarrow \langle P, G\rangle} \qquad [\mathrm{If}_2]\ \frac{C \text{ finitely fails on } G}{\langle \texttt{if } C \texttt{ then } P \texttt{ else } Q, G\rangle \rightarrow \langle Q, G\rangle}$$

$$[\mathrm{Alap}_1]\ \frac{\langle P, G\rangle \rightarrow^+ H}{\langle P!, G\rangle \rightarrow \langle P!, H\rangle} \qquad [\mathrm{Alap}_2]\ \frac{P \text{ finitely fails on } G}{\langle P!, G\rangle \rightarrow G}$$

Figure 4: Inference rules for core commands

The meaning of graph programs is summarised by a semantic function $[\![\_]\!]$ which assigns to each program P the function $[\![P]\!]$ mapping an input graph G to the set of all results of running P on G. The set may contain, besides proper results in the form of graphs, the special value $\perp$ which indicates a nonterminating or stuck computation. The function $[\![\_]\!]: \mathrm{ComSeq} \rightarrow (\mathscr{G} \rightarrow 2^{\mathscr{G} \cup \{\perp\}})$ is defined by

$$[\![P]\!]G = \{H \in \mathscr{G} \mid \langle P, G\rangle \xrightarrow{+} H\} \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}^1$$

where P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \rightarrow \langle P_1, G_1\rangle \rightarrow \langle P_2, G_2\rangle \rightarrow \ldots$, and P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \rightarrow^* \langle Q, H\rangle$.

Note that $[\![P]\!]G = \emptyset$ if and only if P finitely fails on G. A program can get stuck only in two situations: either it contains a subprogram `if` C `then` P `else` Q where C both can diverge from some graph and cannot produce a proper result from that graph, or it contains a subprogram $B!$ where the loop's body B possesses the said property of C. The evaluation of these subprograms will get stuck because the inference rules for branching and iteration are not applicable.

[1] $[\![P]\!]G$ is the application of $[\![P]\!]$ to a graph G, and $\mathscr{G}$ is the set of all graphs.

References

[1] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[2] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178, 2006.

[3] Keith L. Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[4] Claudia Ermel, Michael Rudolf, and Gabi Taentzer. The AGG approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 14, pages 551–603. World Scientific, 1999.

[5] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer-Verlag, 2006.

[6] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[7] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.

[8] Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[9] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

[10] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals*. Springer-Verlag, second edition, 2004.

[11] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[12] Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.

[13] Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proc. Rule-Based Programming (RULE 2009)*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–38, 2010.

[14] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.

[15] Sandra Steinert. *The Graph Programming Language GP*. PhD thesis, The University of York, 2007.

[16] Gabriele Taentzer, Enrico Biermann, Dénes Bisztray, Bernd Bohnet, Iovka Boneva, Artur Boronat, Leif Geiger, Rubino Geiß, Ákos Horvath, Ole Kniemeyer, Tom Mens, Benjamin Ness, Detlef Plump, and Tamás 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*, volume 5088 of *Lecture Notes in Computer Science*, pages 514–539. Springer-Verlag, 2008.