GP 2: Efficient Implementation of a Graph Programming Language

Christopher Bak

Doctor of Philosophy

University of York
Computer Science

September 2015

Abstract

The graph programming language GP (Graph Programs) 2 and its implementation is the subject of this thesis. The language allows programmers to write visual graph programs at a high level of abstraction, bringing the task of solving graph-based problems to an environment in which the user feels comfortable and secure. Implementing graph programs presents two main challenges. The first challenge is translating programs from a high-level source code representation to executable code, which involves bridging the gap from a non-deterministic program to deterministic machine code. The second challenge is overcoming the theoretically impractical complexity of applying graph transformation rules, the basic computation step of a graph program.

The work presented in this thesis addresses both of these challenges. We tackle the first challenge by implementing a compiler that translates GP 2 graph programs directly to C code. Implementation strategies concerning the storage and access of internal data structures are empirically compared to determine the most efficient approach for executing practical graph programs. The second challenge is met by extending the double-pushout approach to graph transformation with root nodes to support fast execution of graph transformation rules by restricting the search to the local neighbourhood of the root nodes in the host graph. We add this theoretical construct to the GP 2 language in order to support rooted graph transformation rules, and we identify a class of rooted rules that are applicable in constant time on certain classes of graphs. Finally, we combine theory and practice by writing rooted graph programs to solve two common graph algorithms, and demonstrate that their execution times are capable of matching the execution times of tailored C solutions.

Contents

Abstract 3

Table of Contents 4

List of Figures 7

List of Tables 10

Acknowledgements 11

Author's Declaration 13

1. Introduction 14

1.1. Motivation . . . 14
1.2. Thesis Contributions . . . 15
1.2.1. Extending GP 2 . . . 16
1.2.2. Implementing GP 2 . . . 16
1.2.3. Rooted Graph Transformation . . . 17
1.2.4. Rooted Graph Programs . . . 17
1.3. Thesis Structure . . . 17

2. Graph Transformation: Theory and Practice 19

2.1. The Theory: Double-Pushout Approach . . . 19
2.1.1. Fundamentals: Graphs, Rules and Graph Morphisms . . . 19
2.1.2. Double-Pushout Approach with Relabelling . . . 22
2.2. The Practice: Programmed Graph Transformation . . . 26
2.2.1. Attributed Programmed Graph Grammars . . . 26
2.2.2. PROGRES . . . 27
2.2.3. AGG . . . 28
2.2.4. GROOVE . . . 29
2.2.5. GrGEN.NET . . . 30
2.2.6. PORGY . . . 31
2.3. Summary . . . 31

3. The GP 2 Programming Language 33

3.1. Introduction . . . 33
3.2. Conditional Rule Schemata . . . 33
3.3. Semantics of Rule Schemata . . . 37
3.4. Graph Programs . . . 40
3.5. Operational Semantics . . . 42

3.6. Summary and Discussion . . . 45

4. Extension to Rooted Graph Programs 47

4.1. Introduction . . . 47
4.2. Rooted Graph Transformation . . . 47
4.3. A Matching Algorithm for Rooted Rules . . . 50
4.4. Extension to Rooted Rule Schemata . . . 54
4.5. Complexity of Rooted Rule Schemata . . . 59
4.5.1. Fast Rule Schemata . . . 59
4.5.2. Unbounded Node Degree . . . 62
4.6. Summary and Discussion . . . 63

5. Implementing GP 2 65

5.1. Introduction . . . 65
5.2. GP 2 Textual Format . . . 68
5.3. The GP 2 Reference Interpreter . . . 68
5.3.1. Reference Interpreters: Uses and Requirements . . . 68
5.3.2. Implementation . . . 70
5.3.3. Performance Evaluation and Conclusions . . . 74
5.4. Experimental Environment . . . 75
5.5. GP 2 System Architecture . . . 76
5.6. Data Structures . . . 77
5.6.1. Host Graphs . . . 77
5.6.2. Host Graph Labels . . . 78
5.6.3. Morphisms . . . 78
5.7. Host List Storage . . . 79
5.7.1. Case Study: Generation of Sierpinski triangles . . . 80
5.7.2. Case Study: Computing Euler Cycles . . . 83
5.7.3. Analysis . . . 87
5.8. Host Graph Backtracking . . . 88
5.8.1. Static Analysis . . . 89
5.8.2. Implementation . . . 91
5.8.3. Case Study: Cycle Checking . . . 92
5.8.4. Case Study: Recognition of Series-Parallel Graphs . . . 94
5.8.5. Analysis . . . 96
5.9. Code Generation . . . 97
5.9.1. Searchplans . . . 99
5.9.2. Conditions . . . 104
5.9.3. Rule Application . . . 106
5.9.4. The Main Routine . . . 109
5.10. Comparison with Existing Implementations . . . 112
5.11. Summary and Discussion . . . 115

6. Case Studies in Rooted Graph Programs 116

6.1. Introduction . . . 116
6.2. Graph Traversing . . . 116

6.2.1. Depth-first Search 117
6.2.2. Breadth-first Search 122
6.3. Case Study: 2-Colouring 126
6.3.1. Non-rooted 2-colouring 127
6.3.2. Rooted 2-colouring 128
6.3.3. Complexity Comparison 133
6.3.4. Experimental Results 134
6.4. Case Study: Topological Sorting 136
6.4.1. Standard Sorting 137
6.4.2. Depth-first Sorting 140
6.4.3. Experimental Results 144
6.5. Comparison with C Programs 146
6.6. Summary and Discussion 151

7. Conclusion **154**
7.1. Evaluation 154
7.2. Future Work 155
7.2.1. Dynamic Rule Matching 155
7.2.2. Optimising the Current Implementation 156
7.2.3. Extending the GP 2 Tool Suite 157
7.2.4. Development of Larger Graph Programs 158

Appendices **159**

Appendix A. GP 2 Concrete Syntax **160**
A.1. Identifiers 160
A.2. Programs and Declarations 161
A.3. Rule Syntax 162
A.4. Host Graph Syntax 163
A.5. Type Grammar and Simple Lists 164
A.6. Context Conditions 165
A.7. Keywords and Operators 167

Bibliography **170**

List of Figures

Figure 2.1. A pushout diagram and an example pushout 22
Figure 2.2. A non-natural pushout and a natural pushout 23
Figure 2.3. A double-pushout diagram illustrating rule application . . . 24
Figure 2.4. A concrete rule application using the DPO with relabelling construction 25

Figure 3.1. A conditional rule schema 34
Figure 3.2. GP 2's type hierarchy 35
Figure 3.3. Abstract syntax of rule schema labels 35
Figure 3.4. Abstract syntax of rule schemata conditions 37
Figure 3.5. Definition of $e^{g,\alpha}$ 38
Figure 3.6. Definition of $c^{g,\alpha}$ 39
Figure 3.7. Declaration and application of a conditional rule schemata . 40
Figure 3.8. Abstract syntax of GP 2 programs 41
Figure 3.9. Inference rules for core commands 43
Figure 3.10. Inference rules for derived commands 44

Figure 4.1. A non-natural pushout and a natural pushout 48
Figure 4.2. A rooted rule schema and two double-pushouts 50
Figure 4.3. The Rooted Graph Matching Algorithm 51
Figure 4.4. The Rooted Graph Matching Algorithm 54
Figure 4.5. The Update Assignment procedure 56
Figure 4.6. Executions of Update Assignment 57
Figure 4.7. The Check procedure 58
Figure 4.8. A left-hand side L and a host graph G 62

Figure 5.1. A GP 2 program in graphical and textual format 68
Figure 5.2. Data flow of the reference interpreter 70
Figure 5.3. Module dependencies in the reference interpreter 71
Figure 5.4. A GP 2 program for transitive closure 74
Figure 5.5. GP 2 system architecture 77
Figure 5.6. C data structure for host graph labels. 78
Figure 5.7. C data structure for host graph labels and atoms. 78
Figure 5.8. GP 2's list hashing function 79
Figure 5.9. Initial triangle and first generation of the Sierpinski triangle 81
Figure 5.10. The program `sierpinski` 82
Figure 5.11. Third generation Sierpinski triangle 82
Figure 5.12. Plots of the `sierpinski` experimental results 83

Figure 5.13. The GP 2 program euler 84
Figure 5.14. Example run of euler 85
Figure 5.15. Alternate output graphs of euler 85
Figure 5.16. The euler cycle computed by euler on the star cycle $SC_{4,4}$. 86
Figure 5.17. Plot of euler's memory use 87
Figure 5.18. Rules for acyclicity testing 92
Figure 5.19. Plot of the memory use of the acyclicity checking programs. 93
Figure 5.20. Rules for series-parallel testing 94
Figure 5.21. Plot of the memory use of the series-parallel checking programs. 95
Figure 5.22. Modified rules for series-parallel testing 96
Figure 5.23. Plot of the memory use of the modified series-parallel checking programs. 97
Figure 5.24. GP 2 compiler architecture 98
Figure 5.25. The GP 2 rule schema rule. 99
Figure 5.26. The structure of rule and its associated searchplan. 101
Figure 5.27. Skeleton of the rule matching code. 102
Figure 5.28. Generated code for label matching. 104
Figure 5.29. Generated C code for a condition 105
Figure 5.30. Generated code for matching a node that participates in the condition 107
Figure 5.31. Generated C code for rule calls 110
Figure 5.32. C code for GP 2 control constructs 111
Figure 5.33. Generated C code for failure. 112
Figure 5.34. Features of graph transformation tools 113

Figure 6.1. Illustration of a depth-first search 117
Figure 6.2. The GP2 program dfs 118
Figure 6.3. Example execution of dfs 120
Figure 6.4. Illustration of a breadth-first search. 122
Figure 6.5. The program bfs 123
Figure 6.6. Example execution of bfs 123
Figure 6.7. The program 2colouring 127
Figure 6.8. The program dfs-2colouring 129
Figure 6.9. Example run of dfs-2colouring 129
Figure 6.10. The GP2 program bfs-2colouring 131
Figure 6.11. Example run of bfs-2colouring 132
Figure 6.12. An example grid graph 134
Figure 6.13. Plots of the runtimes of the rooted 2-colouring programs on grids 135
Figure 6.14. Plot of the runtimes of the rooted 2-colouring programs on stars 136
Figure 6.15. The function topsort 137
Figure 6.16. The program topsort 138
Figure 6.17. Example run of topsort 138

Figure 6.18. A function to compute topological sorting using DFS 141
Figure 6.19. The program `dfs-topsort` 142
Figure 6.20. Example run of program `dfs-topsort` 143
Figure 6.21. Two outgrowing forests of 15 nodes generated by GP 2 . . . 145
Figure 6.22. Topological sortings of a forest 145
Figure 6.23. Plots of the runtimes of the topological sorting programs on stars 146
Figure 6.24. Adjacency-list graph representation in C 148
Figure 6.25. Functions to build the graph. 148
Figure 6.26. DFS 2-colouring in C 150
Figure 6.27. DFS topological sorting in C 150
Figure 6.28. Plots of the runtimes of GP 2 and C 2-colouring programs . 151
Figure 6.29. Plots of the runtimes of GP 2 and C topological sorting programs 152

Figure A.1. Identifier syntax 160
Figure A.2. Program syntax 161
Figure A.3. Rule declaration syntax 162
Figure A.4. Rule label syntax 162
Figure A.5. Condition syntax 163
Figure A.6. Host graph syntax 163
Figure A.7. Syntax of well-typed expressions 164
Figure A.8. Syntax of simple lists 164
Figure A.9. Context conditions (1) 165
Figure A.10.Context conditions (2) 166
Figure A.11.GP 2 keywords (1) 167
Figure A.12.GP 2 keywords (2) 168
Figure A.13.GP 2 operators 169

List of Tables

Table 5.1. Reference interpreter results for the transitive closure program 74
Table 5.2. Experimental results of `sierpinski` using two list storing implementations . . . 83
Table 5.3. Experimental results of `euler` using two list storing implementations . . . 87
Table 5.4. Experimental results of three versions of the acyclicity program 93
Table 5.5. Experimental results of three versions of the series-parallel checking program . . . 95
Table 5.6. Experimental results of three versions of the modified relabelling series-parallel program . . . 97

Table 6.1. Experimental results of three 2-colouring programs on grids 135
Table 6.2. Experimental results of three 2-colouring programs on stars 136
Table 6.3. Experimental results of two topological sorting programs on forests . . . 146
Table 6.4. Comparison of `dfs-2colouring` with a C 2-colouring program 151
Table 6.5. Comparison of `dfs-topsort` with a C topological sorting program . . . 152

Acknowledgements

First and foremost, I thank my family, who demonstrated great patience and understanding in my four years as a PhD student. Time and time again I promised that I would be finishing at a certain point in time, as a means to motivate myself more than anything else, only for my estimate to be grossly unrealistic. These broken promises came without rebuke or questioning, for which I am extremely appreciative. I imagine my life as a research student is something of a mystery to them, and I am very grateful to them for never doubting that I would see it through to the end.

I thank my supervisor, Detlef Plump, whose expertise and guidance was essential to the completion of this thesis. Detlef's enthusiasm and confidence in my work meant that I came out of every supervisor meeting feeling very positive about my research. His attention to detail, technical understanding and open-mindedness greatly benefited my work. I also thank my examiners, Alan Wood and Maribel Fernández, for their interest in my research, for their constructive feedback, and for providing a very enjoyable viva!

I thank the Engineering and Physical Sciences Research Council who provided me with financial support during my doctoral studies. I am grateful to the Department of Computer Science, its staff and students for providing a fantastic, friendly, and well-organised environment to conduct my studies. I thank the Programming Languages and Systems (PLASMA) group for providing a stimulating research environment. I owe particular thanks to the following people: Michael Banks, for going out of his way to welcome me to PLASMA and acclimatising me to the life of a research student. José Calderón for being a great friend for the duration of my PhD and a constant companion during the long final days before the deadline. Glyn Faulkner for his unrivalled enthusiasm and cheerfulness, for our very enjoyable pair programming sessions, and for his invaluable advice in programming and compiler implementation. Chris Poskitt, for his frequent support and whose writing is a great influence on my own. And Colin Runciman, for spearheading an interesting and highly-stimulating collaborative project, and for working his Haskell magic on the GP 2 Reference Interpreter.

Finally, I would like to thank all of the friends whom I met at the University of York for making my life very pleasant during my eight years as a student.

Author's Declaration

I, Christopher Bak, declare that this thesis is a presentation of original work and I am the sole author, except where explicit attribution has been given. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as references. Parts of this thesis have been published previously within the following publications:

- Christopher Bak and Detlef Plump. Rooted Graph Programs. In: *Proc. International Workshop on Graph-Based Tools (GraBaTs 2012)*. Vol. 54. Electronic Communications of the EASST. 2012. *Chapter 4* of this thesis is an extended and restructured version of this paper.
- Christopher Bak, Glyn Faulkner, Detlef Plump, and Colin Runciman. A Reference Interpreter for the Graph Programming Language GP 2. In: *Proc. Graphs as Models (GaM 2015)*. Vol. 181. Electronic Proceedings in Theoretical Computer Science. 2015, pp. 48–64. *Section 5.2* of this thesis contains extracts of this paper, parts of which were written by the other named authors.

1. Introduction

1.1. Motivation

Graphs are an abstract structure used to model the relationships among a collection of objects that provide a direct, intuitive and mathematically precise way of describing complex structures. Graphs are used to model various structures within computer science including software architectures, Petri nets, control flow diagrams, pointer structures and internal representations of programs. Graph transformation, the study of the algorithmic manipulation of graphs, adds another dimension by modelling the dynamic evolution of graph-based systems in which graphs represent static states and graph transformation rules represent a small computation steps on states. Graph transformation has been the subject of heavy research for several decades resulting in the development and study of various theoretical models of graph transformation. We are concerned with *algebraic graph transformation*, which has been one of the most prevalent formalisms since its inception in the 1970s. Broadly speaking, this approach formalises graph transformation by specifying the behaviour of transformation rules as mathematical constructions over the category of graphs and total graph morphisms. The algebraic approach forms the foundation of many implementations of graph transformation systems.

The study of graph transformation began with graph grammars, a generalisation of string rewriting and of tree-based term rewriting. This was motivated theoretically by the study of classifying graph grammars and graph languages analogous to formal language theory, and practically by pattern recognition and compiler construction. As computer science grew and the complexity of both programming languages and software systems increased, standard graph grammars became insufficient to solve the problems for which they had been created from both a practical and a theoretical point of view. Graph grammars were extended with constructs to control the application of productions in a more fine-grained manner. This led to a rich theory of programmable graph grammars and accompanying implementations. Therefore it was desirable to use the well-researched field of graph transformation as a non-deterministic programming paradigm in which a graph is the global environment that is manipulated by controlled application of graph transformation rules.

A graph transformation-based programming language is not only desirable for the specification and manipulation of graph-based systems, but also as a very high-level programming environment accessible to users outside of computer science. A visual environment for programming with graphs allows programmers to define graphs and graph transformation rules without having to concern them-

selves with relatively low-level data structures and pointer manipulation, a frequent source of bugs in production code. Existing tools that offer a form of programmed graph transformation are typically targeted towards applications within computer science, particularly software engineering. However, graphical structures occur in other disciplines such as molecular structures in chemistry and cell structures in biology. Graph transformation systems tailored for computer scientists may be unsuitable for users in less conventional application areas.

The general purpose graph programming language GP (Graph Programs) originated as a core set of control constructs necessary for a computationally complete graph transformation language with double-pushout rules. Although the language has since been extended, this minimalist approach remains the driving force in the design of the language. To this day, it remains very small in comparison to related languages and tools. There are multiple benefits to this design philosophy. First, the simpler the language, the easier it is to learn and use. Writing GP programs only requires a basic understanding of graphs and programming language concepts, and the function of a graph transformation rule can be determined by examining its graphical representation, freeing the user from poring over programming language syntax. Second, the language can be described completely by a formal semantics, one small enough to be used practically for formal verification. Third, the smaller the language, the easier and faster implementation becomes, allowing more focus to be placed on performance and maintainability. However, we do not get all of these advantages for free. The principal drawback is a potential lack of expressiveness: a language providing a limited set of tools for a programmer may not be amenable for writing large scale programs, and may require the programmer to jump through hoops to write a program that could be expressed more concisely in a more complicated language. Striking the balance between simplicity and practicality is a challenge, especially if more weight is placed on the former from the start.

We present GP 2, the second version of the GP language, and its implementation. Implementing a programming language based on graph transformation presents some interesting challenges. The most significant of these challenges is the high complexity in finding a *match*, a subgraph of a host graph at which a graph transformation rule is to be applied. For a graph program with fixed rules, this is polynomial-time in the size of the host graph, making it the bottleneck of an executable graph transformation system. Another challenge is compiling high-level, abstract and non-deterministic graph programs to executable code that preserves the semantics of the language.

1.2. Thesis Contributions

One of the main differences between GP 2 and its predecessor is that GP 2's semantics does not enforce general backtracking. This choice was made with the intention of admitting an efficient implementation. The goal of this implementation is to see how efficiently we can execute graph programs written at a very high level of abstraction. Specifically, we ask the question: *How close can a high-*

level graph programming language come to the performance of graph programs written at a much lower level of abstraction? It is clear that the high-level graph programming language is GP 2. For the latter part of the question, we use the C programming language as a basis of comparison: C code is generated from a GP 2 program specification which we run against a tailored C program performing the same graph-based computation.

1.2.1. Extending GP 2

The first published definition of GP 2 is several years old [Plu12]. Since then, the language has been extended in a number of ways, motivated both by programming in GP 2 and by the theoretical developments and the implementation described in this thesis. GP 2 has been augmented by:

- allowing users to create root nodes. Rooted graph transformation is described in further detail below.
- extending GP 2's macros to *procedures* by providing the ability to create local rules and subprocedures.
- introducing the break control construct, for exiting loops, and its associated semantic rules.
- replacing the Boolean mark system with a fixed set of marks.
- introducing the character type.
- adding the bidirectional edges and wildcard mark, syntactic sugar for frequently occurring program patterns.

1.2.2. Implementing GP 2

GP 2 was implemented in two different ways. Initially, we implemented a reference interpreter in Haskell, motivated by the desire for a quick implementation to facilitate verification of future implementations. To support this aim, the reference interpreter generates all possible outputs of a graph program. The output of future implementations are verified by testing membership in the set returned by the reference interpreter. We succeeded in this goal with a concise and readable Haskell program that can interpret GP 2 programs and examine all nondeterminstic branches of the computation. Although performance was a not a design goal, the interpreter is sufficiently fast to generate output graphs for sample programs to test against a more sophisticated implementation.

The second implementation is the GP 2 compiler, which translates GP 2 graph programs directly to C. C acts as a suitable intermediate language between GP 2 programs and machine code: it is sufficiently high level so that the translation step is not too great, and low level enough to hard-code memory management and optimisations. In addition, we receive all the benefits that come with a highly-established language such as portability, tool support, and optimising compilers.

We detail each step of the translation phase, paying particular attention to critical design choices that could have a large impact on performance in terms of both runtime and memory consumption. The translation steps should satisfy the reader that the generated C code is faithful to the semantics.

1.2.3. Rooted Graph Transformation

One method that graph transformation researchers use to overcome the complexity of searching for an instance of a rule graph in a host graph is to uniquely identify a node in the rule and a node in the host graph. With this "hook", these items are matched in constant time, and the rest of the search takes place in the local neighbourhood of the unique host node if we assume a connected rule graph. The unique identification could arise from a special node label or an explicit user directive. We generalise this concept to *rooted graph transformation*, a novel extension to the double-pushout approach. The basic idea is to equip graphs and morphisms with *root nodes*, forming a new category of rooted graphs and rooted morphisms. Unlike previous approaches, the property of rootedness is independent of a node's label or type, and multiple root nodes are allowed in host graphs and rules. We show that the standard double-pushout construction is preserved in this generalised category. We also present a matching algorithm for rooted graph transformation rules and prove that, under reasonable conditions, a match can be found in constant time.

1.2.4. Rooted Graph Programs

The theory of rooted graph transformation can seamlessly be lifted into GP 2's programming environment by extending GP 2's rules and host graphs with root nodes. We write graph programs for common graph algorithms featuring these rooted rules and demonstrate that they perform in accordance with the theoretical complexity. For some programs, the runtime is in the same order of magnitude as tailored imperative implementations, which we show by comparing compiled GP 2 programs against C implementations of the same graph algorithms. The cost of faster graph programs is increased program complexity in comparison to their unrooted counterparts. However, the case studies illustrate that rooted graph programs are not so complicated as to be impractical.

1.3. Thesis Structure

The thesis is structured as follows.

Chapter 2 sets the scene by presenting the double-pushout approach with relabelling, the mathematical framework for graph transformation used as a base for GP 2. It examines related work in the field, namely the existing languages and tools that support programmable graph transformation, and discusses how GP 2 differs from these approaches.

Chapter 3 presents the GP 2 language. Conditional rule schemata are defined, which are graph transformation rules equipped with variables, expressions and application conditions. Application of a conditional rule schema is formalised and demonstrated with an example. It introduces GP 2's control constructs for controlling rule application and we show GP 2's abstract syntax for labels, conditions and programs. Finally, GP 2's structured operational semantics are presented.

Chapter 4 defines rooted graph transformation as both an extension to the double-pushout approach and an extension to conditional rule schemata. It shows a matching algorithm that can match rooted graph transformation rules and extends it to GP 2 by giving supplementary procedures for matching GP 2 labels. The complexity of matching rooted rule schemata is discussed, including the identification of a class of fast conditional rule schemata that match in constant time under certain conditions.

Chapter 5 describes the implementations of GP 2, starting with the Haskell reference interpreter. Then the GP 2 compiler, which translates GP 2 programs to equivalent C programs, is thoroughly detailed with its supporting runtime library. Two important design choices are addressed, namely the internal storage of GP 2 lists and the management of host graph backtracking, supported by experiments. The code generation phase is described in depth to show how conditional rule schemata and GP 2 control constructs are translated to C code adhering to the semantics. The chapter concludes by comparing the presented implementation with existing implementations of graph transformation systems.

Chapter 6 confirms the theoretical results concerning rooted graph transformation with a number of case studies. It shows rooted GP 2 programs for depth-first search and breadth-first search which are used as the core of GP 2 programs for two graph algorithms: 2-colouring and topological sorting. It analyses these programs and provides experimental evidence demonstrating that the programs perform as efficiently as the theory states, and in some cases matching the performance of tailored C programs.

Chapter 7 summarises the thesis and evaluates its contributions. It also discuss several areas of further research.

Appendix A shows GP 2's concrete syntax, context conditions, keywords and operators.

2. Graph Transformation: Theory and Practice

2.1. The Theory: Double-Pushout Approach

Graph transformation is the study of the modification of graphs by rules. A rule in this context is a rewriting step $L \to R$ where an occurrence of L is located in a *host graph* G and replaced by a copy of R. The formalisation of such a rewriting step is broken down into three fundamental questions:

1. What kind of graphs do we use?
2. How can we identify the left-hand side of a rule with a subgraph of the host graph?
3. How can we replace a subgraph of a host graph with the right-hand side of a rule?

Many formalisms have been defined to answer these questions, the main distinction being how the third question is answered. The Handbook of Graph Grammars and Computing by Graph Transformation [Roz97] gives a thorough coverage of the most enduring frameworks. We focus on the *double-pushout* approach (DPO), introduced in the 1970s by Ehrig, Pfender and Schneider [EPS73]. This approach belongs to the class of *algebraic graph transformation*, named because the graph transformation step is characterised by an algebraic construction, contrasting *set-theoretic* or *algorithmic graph transformation* where the transformation is described algorithmically. Applications of DPO graph transformation rules are free from side effects, so the left-hand and right-hand graphs give a precise description of the behaviour of the rule when applied to a host graph. The results and techniques taken from category theory provide a strong theoretical foundation for proving properties about classes of graph transformation rules in the double-pushout approach. Furthermore, this allows the generalisation of algebraic graph transformation results to other structures because the proofs are often independent of the specific objects and morphisms. The interested reader is referred to [Cor+97] and [Ehr+06].

2.1.1. Fundamentals: Graphs, Rules and Graph Morphisms

We answer the first two fundamental questions by defining graphs and graph morphisms. This is intended to be the theoretical base for a graph programming language. Hence we desire a formalisation that is precise enough to define a

formal semantics for rule application, as well as being practical and usable for programmers.

The GP language operates on directed, labelled graphs [Plu09; Plu12]. In practice, edges are commonly directed because they model asymmetric relations between objects. In addition, it is straightforward to simulate undirected graphs by representing an undirected edge as two edges of opposite direction. Labels are used to encode information and to discriminate between different nodes and edges. We formally define GP 2's graphs below.

Definition 1. A *graph* G over a label alphabet $\mathcal{L}$ is a system $G = (V_G, E_G, s_G, t_G, l_G, m_G)$ where V_G is the finite set of nodes, E_G is the finite set of edges, $s_G\colon E_G \to V_G$ and $t_G\colon E_G \to V_G$ are total functions that map edges to their source and target nodes respectively, $l_G\colon V_G \to \mathcal{L}$ is the partial node-labelling function and $m_G\colon E_G \to \mathcal{L}$ is the total edge-labelling function. We write $l_G(v) = \perp$ if $l_G(v)$ is undefined. If l_G is a total function then G is said to be *totally labelled*.

It is more common to distinguish between node and edge labels by defining two label alphabets. One alphabet is sufficient for our purposes because GP 2 has a universal label set. The other unusual feature of this definition is the partial node-labelling function. This contrasts the traditional graph definition in which all nodes are required to be labelled. As we shall see, the given definition allows us to write double-pushout rules that relabel nodes, something that would otherwise not be possible in general. We note that unlabelled nodes are only used in rules; host graphs are totally-labelled.

Remark 1. We say a node v is *incident* to an edge e, or vice versa, if either $v = s(e)$ or $v = t(e)$. The notation $v \to w$ is used to refer to an edge whose source is v and whose target is w. We use the word *item* to collectively refer to nodes and edges.

Definition 2. A *directed path* is an alternating sequence of nodes and edges $v_0, e_1, \ldots, e_n, v_n$ such that for all $i \in \{1, \ldots, n-1\}$, $s(e_k) = v_{k-1}$ and $t(e_k) = v_k$. An *undirected path* is an alternating sequence of nodes and edges $v_0, e_1, \ldots, e_n, v_n$ such that for all $i \in \{1, \ldots, n-1\}$, e_k is incident to v_{k-1} and v_k. The *length* of a path is the number of edges in the path. A *directed (undirected) cycle* is a directed (undirected) path as above with $v_1 = v_n$.

Remark 2. We use *path* to refer to either an undirected path or a directed path when it is clear from the context. A node w is *directly (undirectly) reachable* from a node v if there exists a directed (undirected) path containing the nodes v and w. An edge e is *undirectly (directly) reachable* from a node v if there exists a directed (undirected) path from v containing e. Again, we say *reachable* when appropriate. A graph is *connected* if every node is undirectly reachable from every other node. A graph is *cyclic* if it contains a cycle, and *acyclic* otherwise. Note that this includes looping edges as they are paths of length 1.

Definition 3. Given graphs G and H, a *graph premorphism* $g\colon G \to H$ is a pair of functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ that preserves sources and targets.

More precisely, for all edges $e \in G$, $s_H(g_E(e)) = g_V(s_G(e))$ and $t_H(g_E(e)) = g_V(t_G(e))$. If g also preserves labels, that is $m_H(g_E(e)) = m_G(e)$ and $l_H(g_V(v)) = l_G(v)$ for all edges $e \in E_G$ and for all nodes $v \in V_G$ with $l_G(v) \neq \bot$, then g is a *graph morphism*. A graph morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective). A bijective graph morphism is called a *graph isomorphism* if g_v preserves undefined labels. A graph morphism g is an *inclusion* if $g(x) = x$ for all nodes and edges $x \in G$. Given two graph morphisms $f\colon G \to H$ and $g\colon H \to J$, the composition is $g \circ f\colon G \to J = (g_V \circ f_V, g_E \circ f_E)$, where $\circ$ is the standard function composition operator.

Remark 3. The graph morphism is a formal description of a structural relationship between two graphs. We omit the prefix "graph" when talking about graph morphisms as we do not use any other type of morphism. In addition, we may specify morphisms by their domain and codomain. For example, the definition of morphism composition could have been written as $(H \to J) \circ (G \to H) = G \to H \to J$.

Definition 4. A *rule* $r = (L \leftarrow K \to R)$ is a pair of inclusions $K \to L$ and $K \to R$ where L and R are totally labelled graphs. L and R are referred to as the *left-hand side* and *right-hand side* respectively. K is called the *interface*.

The purpose of the interface is to precisely specify the rule's behaviour: $L - K$ is the set of items to be deleted, and $R - K$ is the set of items to be added. We are ready to answer the second fundamental question of the start of this section: an instance of L in G is defined by an injective morphism $g\colon L \to G$ called the *match*. Injectivity is not essential, but desirable. From a theoretical point of view, requiring an injective match gives us some nice properties for free. From a practical point of view, it clarifies the purpose of a rule. For example, with non-injective morphisms, a left-hand side containing two nodes and a connecting edge could match a single host graph node with a looping edge. While not strictly a side effect, this is behaviour the user may not have intended or expected when constructing the rule.

A cause for concern when applying a rule is that a node in the host graph may be deleted while one of its incident edges remains. The edge will be left without one of its incident nodes which is not permitted in the definition of a graph. One way to avoid this problem is to forbid matches that violate the condition defined below.

Definition 5. Given a rule $r = (L \leftarrow K \to R)$ and an injective morphism $g\colon L \to G$, the *dangling condition* states that no edge in $G - g(L)$ is incident to any node in $g(L - K)$.

The items in $g(L - K)$ are those removed from G in the first step of rule application, while the edges in $G - g(L)$ are the edges that remain in G after the first step of rule application. Thus the dangling condition states that no preserved edge can be incident to a node that is removed.

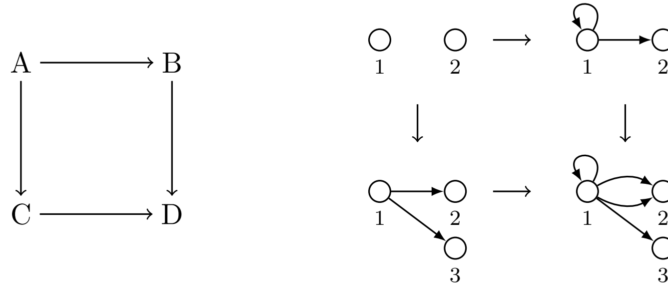


Figure 2.1.: A pushout diagram and an example pushout

2.1.2. Double-Pushout Approach with Relabelling

We are now ready to answer the third fundamental question by defining rule application in terms of a pair of pushouts, a construction from category theory[1] . This thesis uses the definition of double-pushout rules with relabelling as defined in [PS04; Ste07]. The concept originates from Habel and Plump, who modified the original DPO framework to allow straightforward relabelling of nodes by defining graphs to be partially labelled [HP02], although the idea had been investigated as far back as 1987 [PPEM87]. Relabelling is essential in the context of graph programming, since practical computations on graphs cannot be expressed without being able to modify labels. For example, a program computing shortest paths needs to perform arithmetic on labels. In the standard definition of a double-pushout, where graphs are totally labelled, there is no way to specify node relabelling in general because interface nodes need to have the same label in the left-hand side and the right-hand side. Furthermore, applying a rule that relabels a node would force the deletion of the node and the creation of a node with the new label in the same place. Because of this, the dangling condition makes node relabelling impossible in some circumstances. Note that the problem does not occur for edges as they can be arbitrarily deleted and reinserted from a graph without consequence, so edges can be relabelled in this way. Plump and Steinert tailored their approach to GP [PS04] by only allowing nodes to be unlabelled (as in the graph definition of the previous section), and by forcing the left- and right-hand sides of rules to be totally labelled.

Definition 6. Given morphisms $A \to B$ and $A \to C$, a graph D with morphisms $B \to D$ and $C \to D$ is a *pushout* if the following conditions hold:

(i) *Commutativity:* $A \to B \to D = A \to C \to D$

(ii) *Universal Property:* For all pairs of morphisms $(B \to D', C \to D')$ such that $A \to B \to D' = A \to C \to D'$, there exists a unique morphism $D \to D'$ such that $B \to D' = B \to D \to D'$ and $C \to D' = C \to D \to D'$.

An abstract pushout diagram and a concrete example of a pushout are given in Figure 2.1. The diagram on the right contains four graphs which we refer to by their positions in the pushout diagram, e.g. the top left graph is A. The numbers

[1]See Appendix A of [Ehr+06] for an introduction to category theory

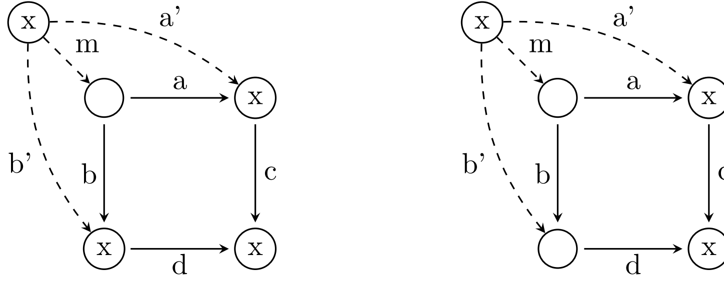


Figure 2.2.: A non-natural pushout and a natural pushout

below the nodes are the node identifiers which are displayed to show how the morphisms map nodes. Edge identifiers and labels are omitted for clarity. The pushout is a formal way of gluing two graphs with respect to a common subgraph. The graph D is the "union" of B and C, where the items that also occur in A are merged. For example, consider the edge $1 \to 2$ and its incident nodes. All these items are present in B and C. The edge is duplicated in D because it does not occur in A. On the other hand, the nodes are not duplicated because they are both in A. The items in A must be in both B and C for commutativity to hold. By the universal property, D is unique up to isomorphism.

Definition 7. Given morphisms $B \to D$ and $C \to D$, a graph A with morphisms $A \to B$ and $A \to C$ is a *pullback* if:

(i) *Commutativity:* $A \to B \to D = A \to C \to D$

(ii) *Universal Property:* For all pairs of morphisms $(A' \to B, A' \to C)$ such that $A' \to B \to D = A' \to C \to D$, there exists a unique morphism $A' \to A$ such that $A' \to B = A' \to A \to B$ and $A' \to C = A' \to A \to C$.

A pushout that is also a pullback is called a natural pushout. We have given the categorical definitions; a construction of pushouts and pullbacks (for totally-labelled graphs) is in the book [Ehr+06]. Pushouts are sufficient to suitably model graph transformation without relabelling, but the use of graphs with partially-labelled nodes causes ambiguity. Consider the two pushout diagrams in Figure 2.2.

In these diagrams, the node outside the square is an instance of the graph A' in the definition of the universal property for pullbacks. The diagram on the left is not a pullback: the universal property is not satisfied. The morphisms a' and b' satisfy $c \circ a' = d \circ b'$, but there is not a unique morphism m that makes the triangles commute because m does not exist. The only possible mapping is to map the node labelled x to an unlabelled node, but that is not a graph morphism since it does not preserve labels. On the other hand, the right-hand diagram is a pullback. Similar to the above, there is no morphism b' because it maps a labelled node to unlabelled node. Therefore the universal property is trivially satisfied because no pair of morphisms $(A' \to B, A' \to C)$ exists.

This example is an illustration of a general result Habel and Plump prove in [HP02] which defines a characterisation of natural pushouts.

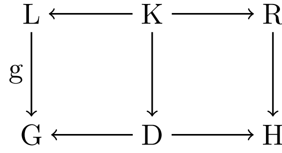


Figure 2.3.: A double-pushout diagram illustrating the application of a rule $r = (L \leftarrow K \rightarrow R)$ with a match g. Both squares are pushouts

Lemma 1. Given two graph morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ such that f is injective, the pushout depicted by the left-hand diagram of Figure 2.1 is *natural* if and only if for all $z \in A, l_A(z) = \bot$ implies $l_B(f(z)) = \bot$ or $l_C(g(z)) = \bot$.

The lemma states that a pushout is a natural pushout if and only if all unlabelled items in A have an unlabelled image in at least one of B and C. Natural pushouts are required for constructing unique double-pushouts with relabelling. If the pullback condition is not enforced, there may be more than one non-isomorphic graph produced from a particular rule application.

Given graphs G and H, a rule $r = (L \leftarrow K \rightarrow R)$, and an injective match, g: $L \rightarrow G$, a *direct derivation* from G to H is a pair of natural pushouts, or a *double-pushout*, depicted in Figure 2.3. If such a derivation exists, we write $G \Rightarrow_{r,g} H$, or more commonly, $G \Rightarrow_r H$. It has been proven that $G \Rightarrow_{r,g} H$ if and only if g satisfies the dangling condition [HP02]. It follows from the definition of a pushout that D and H can be constructed uniquely up to isomorphism. The pushout construction from [Plu09] is an algorithmic description of rule application:

1. To obtain D, remove all nodes and edges in $g(L - K)$ from G. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_D(g_V(v)) = \bot$.

2. Add all nodes and edges, with their labels, from $R - K$ to D. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(E) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.

3. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is H.

In this construction, we see that the interface K represents all nodes and edges that are preserved by the rule. As described before, the items in L that are not in K are removed from G, and the items in R that are not in K are added to G. In addition, any unlabelled nodes in the interface are nodes to be relabelled. Their original labels are removed in the first stage of the rule application, and the new labels from R are added during the third stage.

Figure 2.4 depicts a complete double-pushout rule application according to the above construction. Numbers below nodes are their identifiers, numbers inside nodes are their labels, and characters next to edges are edge identifiers. Edge labels are omitted for clarity. The rule matches an instance of the top left graph. It deletes both edges, creates a new loop on node 1, adds a new edge $2 \rightarrow 1$,

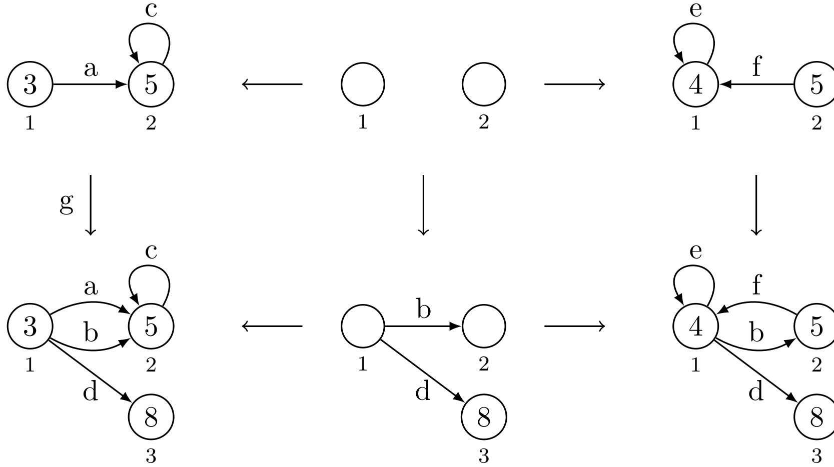


Figure 2.4.: A concrete rule application using the DPO with relabelling construction

and relabels node 1. L is identified in G by the match $g : g_V = (1 \mapsto 1, 2 \mapsto 2); g_E = (a \mapsto a, c \mapsto c)$. This match does not violate the dangling condition because $g(L - K) = \emptyset$. Hence we can apply the rule:

1. $g(L - K) = \{a, c\}$. These edges are removed from G to give D. Neither node in K has a label, so the corresponding nodes in D are also unlabelled.

2. $R - K = \{e, f\}$. These edge are added to D. The source of e is defined to be the image of its source in R: $g_V(s_R(e)) = g_V(1) = 1$. Its target is defined analogously: $g_V(t_R(e)) = g_V(1) = 1$. Similarly the source and target of f are 2 and 1 respectively.

3. Nodes 1 and 2 are unlabelled in K, so they are assigned the corresponding labels in R to obtain the new graph H.

Henceforth, we show only the graphs L and R of rules, and we adopt the convention that the interface contains the nodes specified by numbered identifiers in the left-hand side and the right-hand side. Furthermore, the interface contains no edges and all interface nodes are unlabelled. By this convention, all edges that are matched by the rule are deleted (and reinserted if necessary) according to the pushout construction. This works in theory, but in practice we would prefer to keep the edge instead of deleting and reinserting it to avoid unnecessary computation. For this reason, the language implementation infers preserved edges from the interface nodes [2]. We use this convention to free the user from concerns about edge behaviour: the user only needs to declare the interface nodes.

Given a set of rules $\mathcal{R}$ and two graphs G and H, we say G directly derives H by $\mathcal{R}$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$. Direct derivations can also be applied in sequence. G derives H, or $G \Rightarrow^*_{\mathcal{R}} H$, if either $G \cong H$ or $G \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} H$.

We conclude by remarking that the interface is not essential to formalise graph transformation algebraically. Another prevalent formalism is the single pushout

[2]See Section 5.9 for a detailed description of preserved edge inference.

approach (SPO), introduced in the algebraic graph transformation setting by Löwe and Ehrig [LE91]. As the name suggests, direct derivations only contain one pushout, due to the lack of an interface. A partial morphism $h\colon L \to R$ describes the behaviour of the rule: items not in the preimage of h are removed, while items not in the image of h are created. All other rule items are preserved. Unlike DPO, there are no conditions on the morphism, which introduces some rule application behaviour not visible from the rule graphs. For instance, dangling edges that are created by the rule are deleted, hence SPO rule applications are not completely local. Consequently, single pushout direct derivations are more flexible than their double-pushout counterparts: if a match exists, the rule is always applicable. This is not true for a double-pushout rule because of the dangling condition. For a description of single pushouts and a comparison to double-pushouts, we refer the reader to [Ehr+97].

2.2. The Practice: Programmed Graph Transformation

Graph grammars were originally developed to investigate formal graph languages [PR69; Mon70]. Not only is this interesting from a theoretical point of view, for instance the generation and formal classification of graph languages akin to formal language theory with strings, but it also has various applications in pattern recognition, specification of programming language semantics, and modelling of biological structures. See [Nag78] for a comprehensive bibliographical survey of these early applications. These applications can broadly be categorised as generation of graph languages and recognition of graphs belonging to a particular language. Implementations to support these applications, if necessary at all, did not have to be especially sophisticated in terms of controlling the order of rule application. For some applications, these implementations were impractical because of the complexity of graph matching and the non-determinism of rule application. This motivated the development of more sophisticated graph transformation systems that allowed rules to be partially ordered. In practice, this could be used to limit the number of rules taken into consideration at any particular point in the computation, and also to succintly specify complex graph transformations. We present an early approach called *programmed graph grammars* followed by a survey of existing implementations of graph transformation systems.

2.2.1. Attributed Programmed Graph Grammars

Introduced by Horst Bunke, attributed programmed graph grammars [Bun82] are a precursor to graph programming languages. Bunke gives a two-part extension to conventional graph grammars. First, nodes and edges are attributed. Each production is equipped with an applicability predicate over attributes and a mechanism to modify attribute values. Second, a *control diagram* is added to the grammar to control the application of productions. The control diagram is a state machine where states represent productions. There are two types of transition: a ‘Y’ transition, taken if the production can be applied to the current

graph; and a ‘N’ transition, taken if the production is not applicable. This offers some basic programming of graph rewriting rules.

Bunke’s examples [Bun82] illustrate the key differences between a standard pattern recognition system and his generative approach. In the former, the membership of an input graph is tested by as-long-as-possible applications of the inverse production rules for the language in question which is computationally slow. In the latter, the input graph is transformed into an output graph by the controlled productions. This implicitly recognises the input graph: if it is not a member of the language, the control diagram would take a ‘N’ transition at some point, resulting in no output graph. A transformed output graph signifies that the input graph was valid. In short, the control diagram increases determinism of the system, reducing the computational complexity. Bunke gives two practical examples: the generation of a graphical interpretation of a circuit diagram from an input diagram and generation of graphs representing flowcharts from an input specification. Similar graph grammar-based approaches have been used in diagram recognition to process, for instance musical scores [FB93] and mathematical notation [GB95].

As applications for graph grammars increased in complexity (see the collections [Ehr+99; And+99]), such a method of controlling rule application proved to be too primitive, leading to the invention of implementations that may be collectively referred to as “graph programming languages”. The rest of this section describes the most prevalent of these from a roughly chronological point of view. The aim is to give an overview of these systems and how they specify programmed graph transformation; details on specific implementations and optimisations appear later in the thesis.

2.2.2. PROGRES

Programmed Graph Replacement Systems (PROGRES) [SWZ99] is one of the first high-level graph-based programming languages. The motivation behind PROGRES is to support the development of an integrated programming support environment [ELS87; Eng+92] by providing a high-level specification and programming environment for graph-based structures. Thus the PROGRES language and its integrated tools are used for the specification, generation and validation of graph-based structures [SWZ99]. We present an overview of the key features of the PROGRES programming environment.

The basic object in PROGRES is the DIANE graph, which stands for directed, attributed, node- and edge-labelled graphs. Both nodes and edges are typed: the labels specify the names of these types. Node types exist in a class-based hierarchy, allowing the inheritance of common attribute sets in the usual object-oriented way. In contrast, edges are not attributed because edges are defined as a relation on nodes as opposed to edges being their own objects. Edge types are used only to restrict the types of their incident nodes. PROGRES offers primitive data types for attributes; more complex types can be imported from a C-compatible language. Users define a *graph schema*, a DIANE graph with

attribute type declarations and cardinality constraints on edge types. A graph schema represents a class of permissible graphs, used to formally specify a system from, say, a set of informal requirements. This disambiguates the requirements at the start of software development and facilitates static checking of rules to ensure that they are well-formed.

PROGRES offers complex forms of graph transformation rules due to its powerful querying features. Composable path expressions, restrictions and attribute conditions can be used to query the host graph, either as a predicate or to find a set of nodes matching some particular criteria. PROGRES' graph transformation rules are formulated with a left-hand side and a right-hand side as standard, but they are far more expressive than the rules encountered so far. The left-hand side may contain queries in addition to negative items, optional nodes and node sets. A negative item matches if a suitable host graph item does not exist. An optional node does not cause failure of rule application if the node is not matched. Node sets are matched to as many host graph nodes as possible, including none. Right-hand sides cannot contain any of these constructs. Otherwise, they are arbitrary graphs, so all the standard host graph modifications are expressible, including the modification of attribute values by expressions. Rules are organised with imperative-style control constructs such as sequencing, conditional branching, looping and operators for nondeterministic rule application. Nondeterminism is handled by a Prolog-style depth-first search/backtracking paradigm.

It is clear that, even if we ignore the implementation, PROGRES is a far more sophisticated graph transformation framework than anything we have discussed up to this point. It is a stretch to say that we have only scratched the surface with PROGRES, but we have certainly not covered many important features of the system. A full language description is in [SWZ99] and a terser coverage in [Sch91b]. The existing graph transformation formalisms, including the algebraic approach, were insufficient to capture the entire feature set of PROGRES. Instead, a new logic-based framework was constructed to specify the language and define its formal semantics [Sch97].

2.2.3. AGG

The Attributed Graph Grammar system (AGG) [ERT99; RET11] arose from the desire for a high-level programming environment based on graph transformation rules. Instead of defining a graph transformation formalism for a particular application, the AGG approach bases its programming language on an existing graph transformation framework, specifically the SPO approach, in order to use the established mathematical techniques and results. In this sense it could be considered as a direct implementation of algebraic graph transformation as a programming paradigm.

AGG's graphs contain labelled (typed) nodes and directed edges with attributes. We emphasise that, in contrast to PROGRES, edges are considered objects in their own right, so both nodes and edges can be attributed. AGG is tightly coupled with Java: attribute types are equivalent to valid Java types.

Rules are specified by two graphs and a partial morphism relating their items. These are abstract rules, in the sense that attributes in left-hand side items may contain variables that are instantiated according to the matched host graph item. On the right-hand side, attributes are modified with arbitrary Java methods, allowing powerful computations on attributes. *Negative application conditions* (NACs) are an optional attachment to rules. An NAC is a graph N and a morphism $L \to N$ that acts as an inverse left-hand side. Roughly speaking, the rule is applicable if a match for the NAC does not exist. Prior to the development of AGG, both attributed graphs and NACs were formalised as extensions to SPO [LKW93; HHT95]. Rule application is controlled by a layering system. Rules are grouped into layers by the programmer with the following behaviour: nondeterministically apply rules in the first layer for as long as possible, then move on to the second layer, and continue until the final layer is reached.

AGG's implementation respects its SPO foundation which allows many useful tools and features from the theoretical results to be integrated into the system. For example, a sequence of rules can be combined into a *concurrent rule* whose behaviour is equivalent to the rule sequence. This improves efficiency by searching for a large match once instead of conducting several smaller searches. This has a precise mathematical construction in algebraic graph transformation [Ehr+06]. Concurrent rules and other forms of rule manipulation have been implemented in AGG's second version [RET11]. The AGG tool also contains a graph parser for testing membership of a graph to a particular graph language, and a consistency checker that tests whether a graph grammar preserves certain user-defined conditons. Finally, AGG offers critical pair analysis, an algebraic graph transformation technique for detecting conflicts between rules. One practical application of critical pair analysis used by AGG is determining conflicts of parallel refactoring operations in object-oriented software [MTR05]. Another use of critical pair analysis is determining confluence of a set of graph transformation rules [Plu93]. If confluence holds, then the system is globally deterministic in spite of local non-determinism.

2.2.4. GROOVE

Graph-based Object-oriented Verification (GROOVE) [Gha+12] is a model checking tool for object-oriented systems based on graph transformation. Graphs model system states and graph transformation rules model transitions. GROOVE's primary concern is the generation, storage and exploration of a complete state space of a graph grammar. Consequently, the focus of research is on improving the efficiency and usability of the state space, for example by symmetry reduction or by the merging of multiple transition steps.

GROOVE operates on directed, labelled graphs with optional node and edge typing. Nodes are labelled with types and boolean flags, while edges have only one textual component to their labels. Nodes can be attributed with an edge from a node pointing to a special node representing a data value [3]. Type graphs

[3] This is how typed attributed graphs are defined in the theory of attributed graph transfor-

are available to formally define a well-typed graph. Rules are based on the algebraic approach with negative application conditions: early versions of GROOVE implemented SPO-based rules with NACs [Ren04], but it is not clear if that has persisted to newer versions of the tool. GROOVE supports some sophisticated constructs for rules. One such construct is a regular expression, which is similar to the path expressions of PROGRES. They allow the matching of paths of arbitrary length. A powerful and distinguishing feature of GROOVE is its nested quantified rules [Ren06b], enabling for instance rules to be matched and applied at all matching subgraphs of the host graph in one step.

Due to its main application domain in state space generation, GROOVE's default behaviour is to arbitrarily apply any rule at any point in time. However, GROOVE also supports priority-based rule application, similar to AGG's layering, as well as a small set of textual control constructs. There are multiple ways to generate and explore the search space. Depth-first and breadth-first strategies are used to explore the full state space, while various linear strategies are available if the state space is too large or if the graph transformation rules are known to be confluent (all computation paths lead to the same result). GROOVE also implements a heuristic isomorphism-checking algorithm based on *graph certificates* that characterise the isomorphism classes of graphs [Ren06a].

2.2.5. GrGEN.NET

GrGEN.NET [JBK10] is a system for programmable graph transformation with a heavy emphasis on high performance. It was initially developed for compiler optimisation, specifically the identification of program structures that could be mapped to *rich instructions*, instructions that efficiently compose operations otherwise achieved by multiple standard instructions [Gei+06; SG07]. It has since been expanded to a general purpose tool for graph rewriting called GrGEN.NET [JBK10], named because it compiles a graph rewriting specification into .NET modules.

GrGEN.NET's graphs are typed, attributed graphs with multiple inheritance on node and edge types. Both directed and undirected graphs are usable. Rules are based on the SPO approach with negative application conditions and conditions on types and attributes. DPO rules and non-injective matching are options, giving users some flexibility in writing rules. In addition, rules are parametrised: rules can return matched and non-deleted host graph items to be passed into a subsequent rule application. Rule application is controlled with a domain-specific language called *graph rewrite sequences* [JBK10]. Rules and their sequential composition are basic graph rewrite sequences; logical constructs and regular expressions are used to combine them. For example, given graph rewrite sequences S_1 and S_2, S_1^+ succeeds if S_1 is executed at least once without failing, and $S_1 \& S_2$ succeeds if both S_1 and S_2 can be executed in sequence.

GRGEN.net offers some powerful optimisations. Rules are automatically concatenated if possible [MG07], similar to AGG's concurrent rules. In addition,

mation (see Part III of [EPT04])

rules may contain variables that match arbitrary subgraphs, allowing rules to recursively match frequently occurring patterns in the host graph [HJG08]. In this way an imperative-like graph rewrite sequence can be replaced by a single graph rewriting rule.

2.2.6. PORGY

PORGY [And+11] is a graph transformation tool aimed at the specification and visualisation of graph rewriting systems. Particular emphasis is placed on interaction with the complete state space through a graphical environment, which includes informative graph layouts, a difficult problem to solve for large host graphs. To address this, PORGY is built on top of of the graph visualisation framework Tulip [Aub+12].

One distinguishing feature of PORGY is its graph model. PORGY operates on *port graphs*, a generalisation of labelled directed multigraphs wherein each node has a set of connection points, or *ports*, to which edges are attached. Informally, each node is assigned a set of ports, and each edge label is assigned a pair of ports, denoting the source and target ports of that edge. The use of port graphs was motivated by case studies for graph-based modelling in the natural sciences. Port graph rewriting is defined in [AK08] along with a semantics formed by encoding port graphs (called *labelled multigraphs with ports* or *multigraphs* in the paper) as a term rewriting system.

Another distinguishing feature is its *strategy language* for rewriting, a concept lifted from term rewriting that generalises the control constructs of the aforementioned graph transformation systems. Strategy-based rewriting is distinct from the approaches seen so far. Rule application is localised to a subgraph of the host graph. More formally, a *located graph* is a graph with a *position*, a subgraph specifying the area for rule application. A rule is only applicable if the match has a non-empty intersection with the position. The strategy language is built from iterated rule applications and expressions that transform the position of the host graph. These are combined with control constructs such as sequential composition and conditional branching. The strategy language provides a precise way to control rule matching in terms of application order and application location without adding any extra syntax to the rules themselves. PORGY's strategy language has an operational semantics [FKP14]; a more informal description is in [FKN12].

2.3. Summary

Over the decades since its inception, graph transformation has developed a strong theoretical framework with several standout approaches. We directed our attention towards algebraic graph transformation, specifically the double-pushout approach. We presented the relevant theory for an extension of the double-pushout approach that supports arbitrary relabelling of nodes and edges, the foundation for the subject of this thesis: the graph programming language GP 2.

Originally introduced for more theory-based research goals, graph transformation has become much more application-oriented, with a particular focus on software engineering and system modelling. Indeed, this was the application driving the development of PROGRES, a sophisticated language and tool set for programmable graph transformation. The persistent research in graph transformation coupled with the rise of high-level modelling approaches to software engineering has resulted in a surge of development of graph transformation tools, of which we have presented a small selection. Each has their own unique characteristics, but we can identify typed and attributed graphs as a commonality, especially useful for software engineering applications. Another common feature is the single pushout theoretical base, while double-pushout rules are either not available or offered as an option. We speculate this is due to the increased flexibility of single pushout rules, and perhaps ease of implementation.

GP 2, the graph programming language and the subject of this thesis, is defined in the next chapter. It is much simpler than the languages presented here, particularly in its type system and its straightforward graph transformation rules. The DPO formalism and the lack of features such as path expressions means that the function of a rule can be precisely determined from an inspection of its graphical representation. GP 2 lacks expressive power but the rules, expressions and control constructs allow the construction of powerful and elegant graph programs.

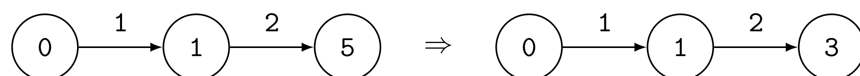
3. The GP 2 Programming Language

3.1. Introduction

In 2001, Habel and Plump showed that nondeterministic application of a rule from a set of double-pushout rules, labelled over a finite label alphabet, is computationally complete using only the control constructs of sequential composition and as-long-as-possible iteration [HP01]. This motivated the design of a small, visual and high-level graph programming language GP 1 [PS04] (then called GP) allowing graph transformation rules to be organised with those control constructs. The language was extended, most significantly with an `if-then-else` conditional branch statement, for reasons of practicality from the programmer's point of view. The extension did not have a significant impact on the simplicity of the language. GP 1 has a small formal semantics to support concrete reasoning on graph programs, one of its primary design goals [Plu09]. Most recently, Plump published a revised and extended graph programming language dubbed GP 2 [Plu12]. The paper defines GP 2 and justifies the changes made to GP 1. In this chapter we provide a definition of GP 2 without reference to older versions of the language, including modifications made since the publication of that paper.

3.2. Conditional Rule Schemata

According to previous definitions, graph transformation rules can only contain fixed labels. They modify graphs structurally and change the labels of nodes and edges. This is useful, but insufficient for a practical graph programming language. For example, when computing the shortest path between two nodes, we might want to perform the following transformation:



This rule could be used in a shortest path algorithm where a node label is the distance of that node from a particular start node (in this case the left-hand node of the rule), and an edge labels is the distance between a pair of nodes. The left-hand side models a possible suboptimal configuration: the distance given by the label of the right-hand node is 5, but the path from left to right has a total distance of 3. The rule updates the right-hand node label to record this improved distance.

To construct a complete graph program to simulate this algorithm, one would have to write such a rule for every possible combination of integers since the

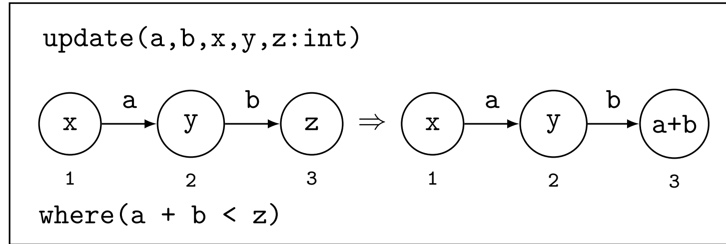


Figure 3.1.: A conditional rule schema

program could be executed on varying input graphs with different node and edge labels. Of course, this would lead to an infinite number of rules. GP 2's conditional rule schemata provide an easy way to overcome this problem: nodes and edges are labelled with expressions which may contain variables. It is also possible to write a condition which forbids the application of the rule if the condition evaluates to false under a particular assignment of variables to values. With these features, the infinite set of rules can be crisply expressed as the single conditional rule schema in Figure 3.1.

Outside of GP 2, this concept is known as *attributed graph transformation*, where computations on labels (attributes) are made available to the programmer. Attributed graph transformation has been formalised in the context of the DPO approach, but the complexity of the formalism[1] conflicts with GP 2's philosophy of a simple syntax and semantics. As we shall see, GP 2 uses unbounded lists of integers and strings as the data type for labels which are easier to formally reason about.

Conditional rule schemata are expressive, graphical and intuitive. The function performed by a rule is clear from the graphical description, and only a basic level of programming knowledge is required to construct such a rule. The presented schema neatly captures the procedure of updating the shortest distance between two nodes. The condition ensures that the label of the right-hand node is updated only if the sum of the edge labels is strictly less than its current label. The rule schema can be applied to any integer-labelled graph.

A label consists of an optional mark and an expression. The finite set of marks allows nodes and edges to be distinguished in a visual way appropriate for programming in a graphical editor. Expressions are typed according to a hierarchical type system. Figure 3.2 shows the subtype relations and the domains of the five types.

The abstract syntax of Figure 3.3 formally defines a GP 2 label. The nonterminals are used as the *sets* of expressions that they define. IVariable, CVariable, SVariable, AVariable, LVariable represent the sets of variables of type `int`, `char`, `string`, `atom` and `list` respectively declared by the rule schema. Node represents the set of left-hand side node identifiers in the schema, and the nonterminal Character represents the fixed character set of GP 2.

[1] For example, the definition of a graph includes an infinite algebra, and special node and edge sets for encoding attributes. See Part III of [Ehr+06] for details.

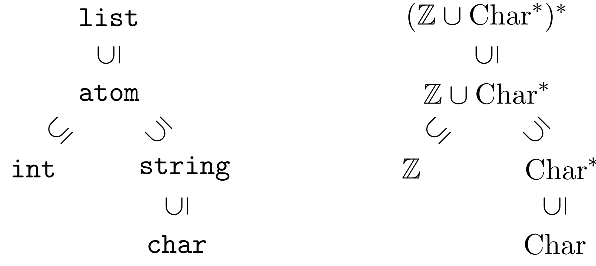


Figure 3.2.: GP 2's type hierarchy

Label ::= List [Mark]
List ::= `empty` | Atom | LVariable | List ':' List
Mark ::= `red` | `green` | `blue` | `grey` | `dashed` | `any`
Atom ::= Integer | String | AVariable
Integer ::= Digit {Digit} | IVariable | '–' Integer | Integer ArithOp Integer | (indeg | outdeg) '(' Node ')' | length '(' (AVariable | SVariable | LVariable) ')'
ArithOp ::= '+' | '-' | '∗' | '/'
String ::= ' " ' {Character} ' " ' | CVariable | SVariable | String '.' String

Figure 3.3.: Abstract syntax of rule schema labels

The marks `grey` and `dashed` are reserved for nodes and edges respectively; the remaining colours are shared. The purpose of the `any` mark is explained shortly. List expressions are formed by concatenating atomic expressions with the colon operator[2] ':' The dot operator '.' is used to concatenate strings. The empty list is signified by the keyword `empty`; it is displayed graphically as a blank label. Integer expressions are variable, constants and the unary operators `indeg`, `outdeg`, and `length` composed with the standard arithmetic operations. The degree operators take a node identifier in the interface and return the appropriate degree of the host graph node to which it is matched. The length operator returns the length of its variable argument according to the variable's type and the value it is assigned during matching. We do not allow these operators in left-hand side labels for cosmetic reasons. We explicitly separate "wordy" textual application conditions from application conditions implied by the structure and variables in the left-hand side. This restriction does not restrict the functionality of rule schemata: a degree operator in a left-hand side label is equivalent to an integer variable in the same location with a schema condition (introduced shortly) requiring the variable to equal the value of the degree operator.

Restrictions are placed on left-hand labels in order to preserve the uniqueness of the match. To illustrate the point, consider the expression $s.t$ with $s, t \in$ SVariable. When matched with the string constant "foo", there are two possible assignments excluding those involving the empty string. Either $s =$ "f"and $t =$

[2]Not to be confused with the `cons` operator in languages such as Haskell, which adds a single element to a list.

"*oo*", or $s =$ "*fo*" and $t =$ "*o*". This is undesirable because if there is an expression on the right-hand side of the schema involving s or t, then the graph produced by the rule will no longer be unique. We wish to preserve the uniqueness given by the double-pushout framework. Thus we define a *simple expression* below and require that expressions in the left-hand side of a rule schema are simple.

Definition 8. An expression $e \in$ List is *simple* if

(i) e contains no arithmetic, degree or length operators.

(ii) e contains at most one occurrence of a list variable.

(iii) each occurrence of a string expression in e contains at most one occurrence of a string variable.

Writing GP 2 programs has motivated the inclusion of two constructs in rules to act as a substitution for multiple rules that match the same pattern except for a difference in mark or edge direction. The first of these is the *bidirectional edge*, represented graphically by an edge without arrows. It matches an edge independent of its direction in the host graph. A similar construct is the `any` mark which was introduced because it is convenient to be able to find a match with any mark. Items marked `any` are called *wildcards*. A wildcard in the left-hand side of a rule can match an appropriate host graph item independent of its mark (but not an unmarked node). Bidirectional edges and wildcards are allowed in the right-hand side if there is a preserved counterpart item in the left-hand side. When such a rule is applied, the direction or mark of the matched host graph item is unchanged. Semantically, these constructs are equivalent to nondeterministic choice from a set of appropriate rules. For example, a rule schema with a bidirectional edge is equivalent to the set of two distinct rule schemata with standard edges such that the edge direction is the same in the left- and right-hand side. We will see later that GP 2 offers nondeterministic choice from a rule set.

The abstract syntax of rule schema conditions is given in Figure 3.4. A single boolean expression is called a *predicate*. Boolean operators compose predicates in the standard way. Conditions are used to impose restrictions on morphisms by querying the structure of the host graph or by interrogating the values assigned to variables. The type predicates (such as `int(`l`)`) check if their List argument has a particular type. Arbitrary list expressions can be compared for equality, and integer expressions can be compared with standard relational operators. The `edge` predicate checks for the existence of an edge between two nodes, but it is normally used in a negative form. It has an optional third argument used to test the label of the edge connecting the two nodes.

We conclude the section with a formal definition of a conditional rule schema.

Definition 9. A *rule schema* $(L \leftarrow K \rightarrow R)$ is a rule such that L and R are graphs over Label, K's nodes are unlabelled, all expressions in L are simple, and all variables in R also occur in L. A *conditional rule schema* $(L \leftarrow K \rightarrow R, c)$ is a rule schema and a condition $c \in$ Condition such that all variables in c also occur in L.

Condition ::= Type '(' List ')' | List ('=' | '!=') List |
Integer RelOp Integer |
`edge` '(' Node ',' Node [',' Label] ')' |
`not` Condition | Condition (`and` | `or`) Condition
Type ::= `int` | `char` | `string` | `atom`
RelOp ::= '>' | '>=' | '<' | '<='

Figure 3.4.: Abstract syntax of rule schemata conditions

Remark 4. We remind the reader that the interface is implicitly denoted by the node identifiers. All nodes with matching identifiers in the left-hand side and right-hand side are present in the interface as unlabelled nodes, and the interface contains no edges.

Remark 5. We sometimes abbreviate *conditional rule schema* to *rule schema* or *rule* when it is clear that we are referring to graph transformation rules in the GP 2 language.

3.3. Semantics of Rule Schemata

Conditional rule schemata differ substantially from regular rules. Standard rules do not contain variables or conditions, so additional mechanisms are required to construct a match and a double-pushout. There are four stages to the application of a rule schema L to a graph G. (1) Find a premorphism $g\colon L \to G$. (2) Check if there exists an assignment α of variables in L to values such that g is label-preserving with respect to α. (3) Check if the condition holds under α. (4) If a valid match and assignment has been found, apply the rule to G by evaluating the labels in R and using the double-pushout construction in Section 2.1.2. This involves using the assignment to evaluate all expressions in the rule schema and relabelling G accordingly.

As seen in the previous section, labels in the rule schema are taken from the syntactic category Label. Input graphs are labelled with values from the semantic domain $\mathcal{L} = (\mathbb{Z} \cup \text{Character}^*)^* \cup ((\mathbb{Z} \cup \text{Character}^*)^* \times \mathbb{M})$, where Character is a finite set of characters and $\mathbb{M}$ is the finite set of marks $\{\text{red}, \text{blue}, \text{green}, \text{grey}, \text{dashed}\}$. To describe the procedure of applying a rule schema, we require the following definition:

Definition 10. An *assignment* is a family of mappings $\alpha = (\alpha_X)_{X \in \{I,C,S,A,L\}}$ where:

- α_I : IVariable $\to \mathbb{Z}$
- α_C : CVariable $\to$ Character
- α_S : SVariable $\to$ Character*
- α_A : AVariable $\to \mathbb{Z} \cup$ Character*

Expression (e)	Value ($e^{g,\alpha}$)
`empty`	The empty sequence.
Digit {Digit}	The integer represented by e.
$-x$	$-x^{g,\alpha}$
$x \oplus y$	$x^{g,\alpha} \oplus_{\mathbb{Z}} y^{g,\alpha}$
`indeg`(n)	The indegree of $g_V(n)$.
`outdeg`(n)	The outdegree of $g_V(n)$.
`length`$(v), v \in$ LVar	The number of atoms in $\alpha(v)$.
`length`$(v), v \in$ AVar	1 if $\alpha(v) \in \mathbb{Z}$; the number of characters in $\alpha(v)$ if $\alpha(v) \in$ Char*.
`length`$(v), v \in$ SVar	The number of characters in $\alpha(v)$.
"{Character}"	The string represented by e.
Variable	$\alpha(e)$
$s_1 \,.\, s_2$	The string concatenation of $s_1^{g,\alpha}$ and $s_2^{g,\alpha}$.
$e_1 : e_2$	The list concatenation of $e_1^{g,\alpha}$ and $e_2^{g,\alpha}$.

Figure 3.5.: Definition of $e^{g,\alpha}$

- α_L : LVariable $\rightarrow (\mathbb{Z} \cup \text{Character}^*)^*$

Given a premorphism g: $L \rightarrow G$, an assignment α, and a label $l = em$ with $e \in$ *List* and $m \in \mathbb{M}$, the value $l^{g,\alpha} \in \mathcal{L}$ is the pair $(e^{g,\alpha}, m)$. $e^{g,\alpha} \in (\mathbb{Z} \cup \text{Character}^*)^*$ is the value of the List e when evaluated with respect to the premorphism g and assignment α. In addition, we define $c^{g,\alpha} \in \mathbb{B}$, which is the value of the rule schema condition when evaluated with respect to the premorphism g and assignment α. Both are defined inductively in Figure 3.5 and Figure 3.6.

In the tables, $e, e_1, e_2 \in$ List, $x, y \in$ Integer, $m, n \in$ Node, $s_1, s_2 \in$ String, and $c_1, c_2 \in$ Condition. The symbol $\oplus_{\mathbb{Z}}$ signifies the integer operation represented by $\oplus$. Division by zero is undefined. Similarly, $\bowtie_{\mathbb{Z}}$ is the integer relation represented by $\bowtie$. Also note that the integer (string) represented by a sequence of digits (characters) is unique.

Definition 11. $r^{g,\alpha} = (L^{g,\alpha} \leftarrow K \rightarrow R^{g,\alpha})$ is the instance of r with respect to g and α, where $L^{g,\alpha}$ and $R^{g,\alpha}$ are the graphs L and R after the replacement of their labels l with $l^{g,\alpha}$.

Condition (c)	$c^{g,\alpha} = \texttt{true} \Leftrightarrow$
$\texttt{int}(e)$	$e^{g,\alpha} \in \mathbb{Z}$
$\texttt{char}(e)$	$e^{g,\alpha} \in \text{Character}^*$
$\texttt{string}(e)$	$e^{g,\alpha} \in \text{Character}^*$
$\texttt{atom}(e)$	$e^{g,\alpha} \in \mathbb{Z} \cup \text{Character}^*$
$e_1 = e_2$	$e_1^{g,\alpha} = e_2^{g,\alpha}$
$e_1 \neq e_2$	$e_1^{g,\alpha} \neq e_2^{g,\alpha}$
$x \bowtie y$	$x^{g,\alpha} \bowtie_{\mathbb{Z}} y^{g,\alpha}$
$\texttt{edge}(m, n)$	$\exists e \in E_G \mid s_G(e) = g_V(m) \wedge t_G(e) = g_V(n)$
$\texttt{edge}(m, n, e)$	$\exists e \in E_G \mid s_G(e) = g_V(m) \wedge t_G(e) = g_V(n) \wedge m_G(e) = e^{g,\alpha}$
$\texttt{not}\ c_1$	$c_1^{g,\alpha} = \texttt{false}$
$c_1\ \texttt{and}\ c_2$	$c_1^{g,\alpha} = \texttt{true} = c_2^{g,\alpha}$
$c_1\ \texttt{or}\ c_2$	$c_1^{g,\alpha} = \texttt{true} \vee c_2^{g,\alpha} = \texttt{true}$

Figure 3.6.: Definition of $c^{g,\alpha}$

Definition 12. Given a conditional rule schema $r = (L \leftarrow K \rightarrow R, c)$, and graphs G, H over $\mathcal{L}$, G *directly derives* r, denoted $G \Rightarrow_{r,g} H$ (or $G \Rightarrow_r H$), if there exists a premorphism $g\colon L \rightarrow G$ and an assignment α such that

(i) g is a morphism $L^{g,\alpha} \rightarrow G$.

(ii) $c^{g,\alpha} = \texttt{true}$.

(iii) $G \Rightarrow_{r^{g,\alpha},g} H$.

We do not introduce new notation when defining direct derivations over conditional rule schemata as opposed to traditional rules. When the notation is used, it will be clear from the context which type of direct derivation is being described.

Even with the extension of traditional rules to conditional rule schemata, uniqueness of double-pushout direct derivations is preserved. Only simple expressions are allowed in L, so for any conditional rule schema and premorphism g, there is at most one assignment that makes g a morphism. Uniqueness (up to isomorphism) of the transformed graph follows from this and the uniqueness of the graph H in the double-pushout diagram of Figure 2.1.

We demonstrate the application of a conditional rule schema to a graph by an illustrative example in Figure 3.7. The rule is declared at the top with a rule identifier followed by a list of variable declarations. Multiple variables of the same type can be declared simultaneously. Declarations of variables of different types must be separated by a semicolon. Node 1 on the left-hand side and node 3 on the right-hand side have the mark `grey`, and the edge $1 \rightarrow 3$ on the right-hand side has the mark `dashed`.

The rule `bridge` is applied to the lower left graph. The upper square of graphs depicts the instantion of `bridge` with respect to the premorphism g_V: $(1 \mapsto$

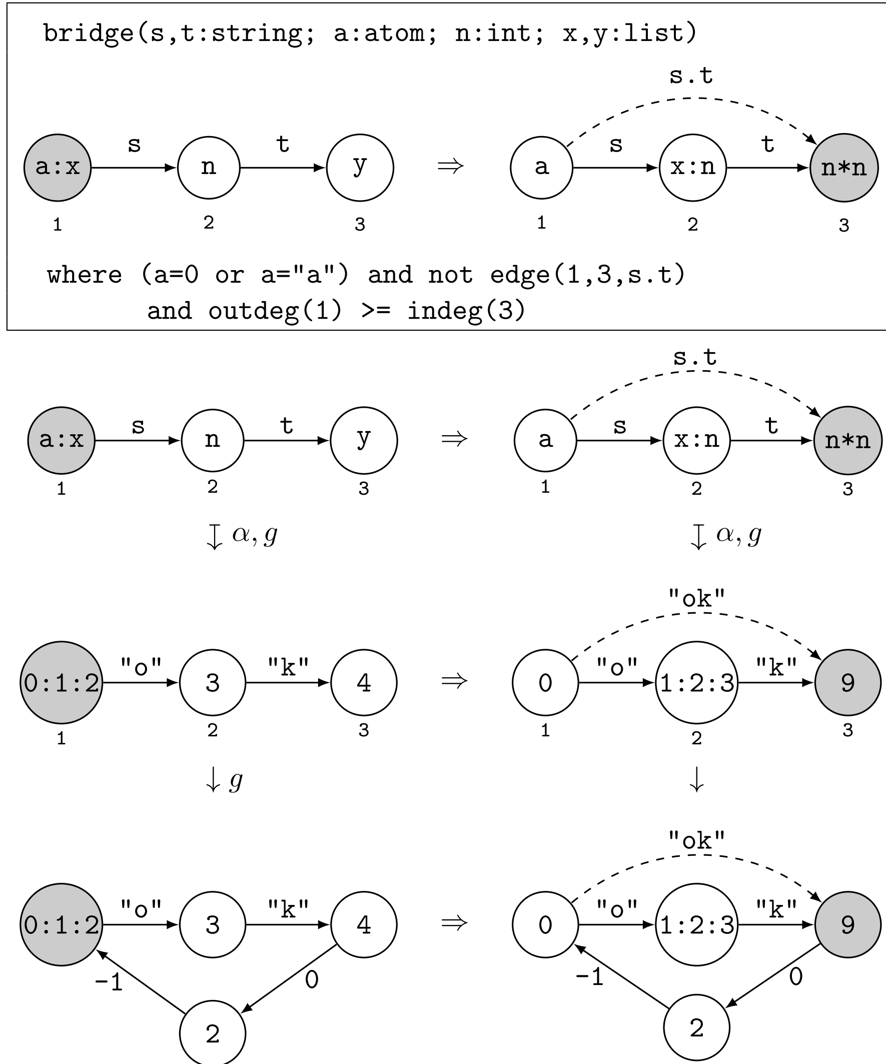


Figure 3.7.: Declaration and application of a conditional rule schemata

$1, 2 \mapsto 2, 3 \mapsto 3$) ($g_E$ defined in the obvious way) and the assignment α: ($a \mapsto 0, x \mapsto 1 : 2, n \mapsto 3, y \mapsto 4, s \mapsto$ "o", $t \mapsto$ "k"). After variable assignment, g is label-preserving and hence a morphism. In addition, the condition clearly holds with respect to g and α. Therefore the rule can be applied to the host graph to give the lower right graph.

3.4. Graph Programs

GP 2 programs are composed by defining rule schemata and organising them using a small set of control constructs. It has already been established that nondeterministically applying rules from a set, sequentially composing rules and iterating subprograms is sufficient for computational completeness [HP01]. However, GP 2 offers more control constructs for usability. The abstract syntax of GP 2 programs is shown in Figure 3.8.

A program is a list of three types of declarations: rule declarations, described in

Prog	::=	Decl {Decl}
Decl	::=	MainDecl | ProcDecl | RuleDecl
MainDecl	::=	`Main` '=' ComSeq
ProcDecl	::=	ProcId '=' [LocalDecl] ComSeq
LocalDecl	::=	(RuleDecl | ProcDecl) { LocalDecl }
ComSeq	::=	Com {';' Com}
Com	::=	RuleSetCall | ProcCall
		| `if` ComSeq `then` Comseq [`else` ComSeq]
		| `try` ComSeq [`then` Comseq] [`else` ComSeq]
		| ComSeq '!'
		| ComSeq `or` ComSeq
		| '(' ComSeq ')'
		| `break` | `skip` | `fail`
RuleSetCall	::=	RuleId | '{' [RuleId { ',' RuleId}] '}'
ProcCall	::=	ProcId

Figure 3.8.: Abstract syntax of GP 2 programs

the previous section; the declaration of the main procedure at which computation starts; and other procedure declarations, which provide a way to organise long programs with local rules and subprograms. There must be exactly one main declaration. At its core, a GP 2 program is a sequence of commands.

The behaviour of the control constructs is informally described here; a formal operational semantics is presented later. The program environment is the host graph which is manipulated by conditional rule schemata. Therefore a rule call is the basic unit of computation. GP 2 provides the rule set call, a nondeterministic choice from a set of rules. Failure occurs if a match does not exist for any rule in the set. A procedure call executes the command sequence of that procedure. Procedures are non-recursive and are essentially macros, in that a procedure call can be substituted by its command sequence up to relabelling of rule names (this point is elaborated shortly). There are two conditional branching statements. Their behaviour is unlike in a typical programming language because the condition is an arbitrary command sequence as opposed to the more common boolean expression. Branching depends on the success or failure of the condition. The condition can modify the host graph; the two branches differ in how the host graph is handled after the condition finishes execution. The `if-then-else` branch discards the changes made during execution of the condition regardless of the branch taken, while `try-then-else` keeps the changes if the condition terminates with a valid graph. GP 2 introduced the `try-then-else` statement to allow preservation of changes made by the conditions of branching statements. The loop command '!' iterates its command sequence for as long as possible. The choice command `or` nondeterministically chooses one of its two subprograms. The `break` command is used to exit a loop without discarding changes to the host graph made in the current loop iteration. The final two com-

mands, `skip` and `fail`, are a convenience to the user to simplify some common patterns. An example of a small GP program, with rule schemata omitted, is given below.

```
Main = start; Middle; end
Middle = {foo; bar}!
```

Program execution starts at the main procedure's command sequence. The rule `start` is applied, followed by an execution of the procedure `Middle`. Procedures with no local declarations offer no extra functionality. They are used to enhance readability. The program would operate equivalently if the procedure identifier `Middle` were replaced by its command sequence (`{foo; bar}!`). `Middle` nondeterministically repeatedly applies either `foo` or `bar` until neither `foo` nor `bar` can be applied to the working graph. If this happens, the loop terminates, and the rule `end` is applied once. The program then terminates since there are no commands remaining.

In the example, the procedure acts only as a textual placeholder for an intermediate command sequence. Procedures can declare their own rules and procedures. Declarations in the Main procedure have global scope and can be seen by any procedure. Declarations within any other procedure are visible only to that procedure. This allows definitions of rules with the same name in multiple places. When executing a procedure's command sequence, a rule local to that procedure has precedence over a global rule with the same name. We have yet to use local procedures and local rules, but we anticipate that this will be useful for complex GP 2 programs. In particular, rules that perform "garbage collection" on the host graph are frequently used; it would be cumbersome to give a unique name to each cleaning-up rule for multiple procedures.

3.5. Operational Semantics

GP2 has a formal semantics, presented here in the style of Plotkin's structural operational semantics [Plo04]. The inference rules, shown in Figure 3.9 and Figure 3.10, inductively define a small-step transition relation $\rightarrow$ on configurations. A configuration represents a program state during any stage of program execution. This could be either an unfinished program execution, represented by a command sequence and the current graph; the final graph, after all commands have been executed; or a failure state, represented by the special element `fail`:

$$\rightarrow \subseteq (\text{ComSeq} \times G_{\mathcal{L}}) \times ((\text{ComSeq} \times G_{\mathcal{L}}) \cup G_{\mathcal{L}} \cup \{\texttt{fail}\}).^3$$

The rules contain meta-variables, considered to be universally quantified. R stands for a rule set call, C, P, P' and Q stand for command sequences, and G and H stand for graphs in $G_{\mathcal{L}}$. The notation $G \not\Rightarrow_R$ means that there does not exist a graph H such that $G \Rightarrow_R H$. Each rule has a premise and a conclusion separated by a horizontal bar. $\rightarrow^+$ is the transitive closure of $\rightarrow$. For example, [Call_1] reads: if the working graph G directly derives H by R, then the command

[3] $G_{\mathcal{L}}$ is the class of totally labelled graphs over the label alphabet $\mathcal{L}$.

$$[\text{Call}_1]\frac{G \Rightarrow_R H}{\langle R, G\rangle \to H} \qquad [\text{Call}_2]\frac{G \not\Rightarrow_R}{\langle R, G\rangle \to \texttt{fail}}$$

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

$$[\text{Seq}_3]\frac{\langle P, G\rangle \to \texttt{fail}}{\langle P; Q, G\rangle \to \texttt{fail}} \qquad [\text{Break}]\ \frac{\langle \text{break}; P,\ G\rangle}{\langle \text{break},\ G\rangle}$$

$$[\text{If}_1]\frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, G\rangle} \qquad [\text{If}_2]\frac{\langle C, G\rangle \to^+ \texttt{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{Try}_1]\frac{\langle C, G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle P, H\rangle} \qquad [\text{Try}_2]\frac{\langle C, G\rangle \to^+ \texttt{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q, G\rangle \to \langle Q, G\rangle}$$

$$[\text{Loop}_1]\frac{\langle P, G\rangle \to^+ H}{\langle P!, G\rangle \to \langle P!, H\rangle} \qquad [\text{Loop}_2]\frac{\langle P, G\rangle \to^+ \texttt{fail}}{\langle P!, G\rangle \to G}$$

$$[\text{Loop}_3]\ \frac{\langle P,\ G\rangle \to^* \langle \text{break}, H\rangle}{\langle P!,\ G\rangle \to H}$$

Figure 3.9.: Inference rules for core commands [Plu12]

sequence R executed on G gives the graph H.

Reading the inference rules for the conditional branching commands `if-then-else` and `try-then-else` tells us the following: [If_1]: If the application of the command sequence C to the graph G succeeds, generating the graph H, then continue by applying the command sequence P to G. [If_2]: If the first attempted application of the command sequence C to the graph G fails, then continue by applying the command sequence Q to G. [Try_1]: If the first attempted application of the command sequence C to the graph G succeeds, generating the graph H, then continue by applying the command sequence P to H. [Try_2]: If the first attempted application of the command sequence C to the graph G fails, then continue by applying the command sequence Q to G.

The difference is the behaviour on success of the condition. `if-then-else` is non-destructive: any changes made to G by C are ignored before the then branch is executed. On the other hand, `try-then-else` will retain the modified graph H, but it uses the initial graph G if C fails.

The inference rule abstracts away from the structure of the condition: since rule application is nondeterministic, applying the command sequence C to a graph G may produce different results. In particular, it possible that some executions fail while others succeed. However, it would be sound for an implementation to examine only one computation path and act on the result. Conditions are written to control program flow, so it would be expected that the condition is written in a rigorous manner, namely that it either always succeeds or always fails. It is unnecessary to explicitly program nondeterminism into a condition; this behaviour can be achieved more concisely with the `or` command.

The loop command also contains some subtleties. Consider a looped sequence

$[\text{Or}_1]\ \langle P\ \texttt{or}\ Q, G\rangle \to \langle P, G\rangle$ $\qquad$ $[\text{Or}_2]\ \langle P\ \texttt{or}\ Q, G\rangle \to \langle Q, G\rangle$

$[\text{Skip}]\ \langle \texttt{skip}, G\rangle \to G$ $\qquad$ $[\text{Fail}]\ \langle \texttt{fail}, G\rangle \to \texttt{fail}$

$[\text{If}_3]\ \langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$

$[\text{Try}_3]\ \langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else skip}, G\rangle$

$[\text{Try}_4]\ \langle \texttt{try}\ C\ \texttt{else}\ Q, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else}\ Q, G\rangle$

$[\text{Try}_5]\ \langle \texttt{try}\ C, G\rangle \to \langle \texttt{try}\ C\ \texttt{then skip else skip}, G\rangle$

Figure 3.10.: Inference rules for derived commands [Plu12]

of three rule applications `(r1; r2; r3)!`. [Loop$_1$]'s meaning is unsurprising: if executing the loop body on G gives the graph H, execute the loop body again on H. [Loop$_2$] states that if the loop body fails at any point, exit with graph G. Then, if $G \Rightarrow_{r1} H$ and $G \not\Rightarrow_{r2} H$, the loop exits after failing to match `r2` while discarding the changes made by `r1`. The `break` statement is provided to retain any intermediate changes in a loop body. To achieve this, one can write `(r1; try r2 else break; r3)!`. If `r1` succeeds and `r2` fails, then the loop body reduces to `(break; r3)!` after applying the inference rules [Seq$_2$] and [Try$_2$]. The [Break] rule is used to obtain the premise for [Loop$_3$], which exits the loop while retaining the working graph H.

Derived commands are those which can be expressed by a command sequence using the core commands. In other words, they are abbreviated forms of common GP 2 control mechanisms. After defining the semantic function we will show how these commands are equivalent to expressions consisting of only core commands.

The semantic function $[\![_]\!]$ defines the meaning of GP 2 programs by mapping an input graph G to the set of all possible results of executing a program P on G. The application of $[\![P]\!]$ to G is written $[\![P]\!]G$. The result set may contain, besides proper results in the form of graphs, the special values `fail` and $\perp$. The value `fail` indicates a failed program run while $\perp$ indicates a run that does not terminate or gets stuck. Program P *can diverge from* G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$ Also, P *can get stuck from* G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

Definition 13 (Semantic function)**.** The *semantic function* $[\![_]\!]\colon \text{ComSeq} \to (\mathcal{G}(\mathcal{L}) \to 2^{\mathcal{G}(\mathcal{L}) \cup \{\text{fail}, \perp\}})$ is defined by

$$[\![P]\!]G = \{X \in (\mathcal{G}(\mathcal{L}) \cup \{\text{fail}\}) \mid \langle P, G\rangle \overset{+}{\to} X\} \cup \{\perp \mid P \text{ can diverge or get stuck from } G\}.$$

A program can get stuck in two situations: (1) it contains a command `if` C `then` P `else` Q or `try` C `then` P `else` Q such that C can diverge from a graph G, or (2) it contains a loop $B!$ whose body B can diverge from a graph

G. The evaluation of such commands gets stuck because none of the inference rules for `if-then-else`, `try-then-else` or looping are applicable. Getting stuck always signals some form of divergence.

The semantic function of Definition 13 suggests a straightforward notion of program equivalence.

Definition 14 (Semantic equivalence)**.** Two programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$.

It is easy to see that the following equivalences between derived commands and core commands hold. Let `null` be the rule schema $\emptyset \Rightarrow \emptyset$ where $\emptyset$ is the empty graph. Then, for all programs C and P:

- `skip` $\equiv$ `null`,
- `fail` $\equiv \{\}$, the empty set of rule schemata;
- `if` C `then` $P \equiv$ `if` C `then` P `else null`;
- `try` C `then` $P \equiv$ `try` C `then` P `else null`.
- `try` $C \equiv$ `try` C `then null else null`.

A non-trivial equivalence is required to show that `or` is a derived command:

$$P \,\texttt{or}\, Q \equiv \texttt{if remove!; \{create, null\}; zero then } P \texttt{ else } Q,$$

Here `remove` is a set of three rule schemata that delete arbitrary edges, loops and isolated nodes, `create` is the rule schema that creates a single 0-labelled node, and `zero` matches a single 0-labelled node and does nothing. The condition uses the non-determinism of the rule set call to select a branch, leading to the non-deterministic choice between P and Q.

Finally, it may appear as though the `if-then-else` command can be used in a straightforward way to simulate the `try-then-else` command. However, this is not possible:

$$\texttt{try } C \texttt{ then } P \texttt{ else } Q \not\equiv \texttt{if } C \texttt{ then } C; P \texttt{ else } Q.$$

The command sequences $C =$ `skip or fail`, $P =$ `skip` and $Q =$ `skip` form a counterexample to semantic equivalence. `try-then-else` simplifies to `skip` and hence cannot fail, but `if-then-else` can fail.

3.6. Summary and Discussion

We have described GP 2, a graph programming language that is smaller and simpler than related languages and tools. GP 2 programs are based on high-level graph transformation rules that are written graphically with a small set of textual syntax for conditions, labels, and control flow. The underlying DPO formalism ensures that the behaviour of these rules is local, free from side-effects, and

easily understandable from examination of the two rule graphs. These features make it straightforward for a programmer to construct graph programs without needing to delve into lower-level code such as the notoriously error-prone pointer structure in a language such as C. GP 2's lack of complexity facilitates a small abstract syntax and complete formal semantics for the language which provides a solid base for language implementors and for formal reasoning. Furthermore, we believe that GP 2 is a very accessible language, an important factor because graph transformation is currently quite obscure as a programming paradigm. Indeed, the area may be intimidating to newcomers because of its substantial theoretical foundations. Hence GP 2 may not only serve as a programming environment, but as a fun method of teaching graph transformation.

From a practical point of view, GP 2 is quite flexible as a programming language. It is capable of solving graph algorithms in an elegant and declarative way, an area with a wide berth of applications but one that is relatively unexplored in the graph transformation field. Published GP solutions to graph algorithms include Dijkstra's shortest path algorithm [PS04], vertex colouring [Plu09] and the computation of Euler cycles [Plu12], with more examples later in the thesis. Another common class of GP programs is recognition of graphs by reduction, one example being acyclic graph recognition [Plu12]. A potential use case for graph reduction is the specification and verification of pointer structures in imperative languages [DP06a; Dod08]. A more involved GP program is minimisation of finite automata [PSS11]. A limitation of the language is that its simple type system does not make GP 2 especially suitable to applications in software engineering, especially in comparison to other existing graph transformation systems.

4. Extension to Rooted Graph Programs

4.1. Introduction

The bottleneck for using graph transformation rules in programming is the inefficiency of graph matching. In general, to match the left-hand graph L of a rule within a host graph G requires time $\mathrm{size}(G)^{\mathrm{size}(L)}$. As a consequence, linear graph algorithms are slowed down to polynomial complexity when they are recast as programmed graph transformation systems.

One way to speed up graph matching, going back to Dörr's book on efficient graph rewriting [Dör95], is to equip rules and host graphs with distinguished nodes, so-called roots, and to match roots in rules with roots in host graphs. The same idea underlies Fujaba's requirement that each method must have a "this" node at which graph matching starts [NNZ00; Det+12]. A related concept in GrGen is rules that return graph elements to restrict the location of subsequent rule applications [Gei+06]. The PORGY environment uses a similar model: rules can only be applied at a certain position, where a position is a subgraph of the host graph. The position can be moved as part of the program [And+11].

Dodds and Plump [DP06b; Dod08] have considered rooted graph transformation by using uniquely labelled nodes as roots. They show that graph matching can be achieved in constant time if rules have a connected left-hand graph and host graphs have bounded node degrees. In addition, they use rooted rules in a rule-based extension of C that allows to check the shape safety of pointer manipulations [DP06a]. We generalise the approach of [DP06b; Dod08] from graph transformation rules to graph programs by extending GP 2 with rooted rule schemata. We present a fast and complete algorithm for matching rooted rule schemata.

The main contribution of this chapter is to identify *fast* rule schemata, a large class of rooted conditional rule schemata, and to prove that they can be applied in constant time if host graphs have a bounded node degree. In practice, the latter assumption is often satisfied. For example, traffic networks, digital circuits and social networks often have an upper bound on the number of edges attached to nodes. The subsequent chapters demonstrate that high performance rooted graph programs can be written, in some cases approaching the runtime of tailored C implementations.

4.2. Rooted Graph Transformation

We extend the definitions of Section 2.1.1 and Section 2.1.2 to include distinguished root nodes in both rules and host graphs. Our approach is to treat

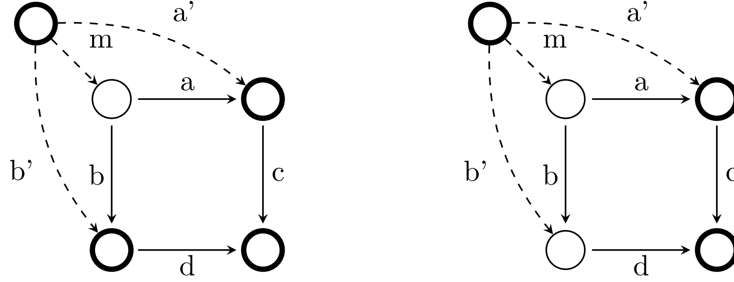


Figure 4.1.: A non-natural pushout and a natural pushout

rooted graphs and root-preserving morphisms as "first-class citizens" instead of encoding roots by unique labels. Unlike [DP06b; Dod08], we allow multiple roots in rule schemata and host graphs. This is useful in certain situations. For example, for host graphs with disconnected components, it may be desirable to perform a rooted computation on each component.

Definition 1. A *rooted graph* is a pair $\langle G, P_G \rangle$ where G is a graph and $P_G \subseteq V_G$ is a set of *roots*. A morphism $g : G \to H$ is *root-preserving* if $g(P_G) \subseteq P_H$.

Remark 6. Rooted graphs (over some label set) and root-preserving morphisms form a category.

Definition 2. A *rooted rule* $r = \langle \langle L, P_L \rangle \leftarrow \langle K, P_K \rangle \to \langle R, P_R \rangle \rangle$ is a pair of root-preserving inclusions $\langle K, P_K \rangle \to \langle L, P_L \rangle$ and $\langle K, P_K \rangle \to \langle R, P_R \rangle$ where L and R are totally labelled.

In Section 2.1.2 we observed that generalising from totally-labelled graphs to partially-labelled graphs introduces ambiguity because two distinct double-pushouts can be constructed from the same rule. To guarantee uniqueness, we enforce a stricter condition on pushouts, namely that they are also pullbacks. The same issue occurs with rooted graphs. Figure 4.1 is the same diagram as Figure 2.2 except that labelled nodes are replaced by root nodes, where root nodes are nodes with thick borders. The left diagram is not a pullback because there exist root-preserving morphisms a' and b', while a root-preserving morphism m does not exist. The right diagram is a pullback because there does not exist a root-preserving morphism b'.

Let $\mathcal{C}$ be the category of partially labelled rooted graphs and root-preserving graph morphisms over some fixed label alphabet $\mathcal{L}$. Let $\overline{\mathcal{C}}$ be the category of partially labelled unrooted graphs and graph morphisms. Given a graph G in $\mathcal{C}$, we write $\overline{G}$ for the underlying unrooted graph. By choosing the root set to be empty, we see that $\overline{\mathcal{C}}$ is a subcategory of $\mathcal{C}$.

Lemma 2. Given morphisms $C \stackrel{c}{\leftarrow} A \stackrel{b}{\to} B$ in $\mathcal{C}$ such that square (1) is a pushout in $\overline{\mathcal{C}}$, square (2) is a pushout in $\mathcal{C}$ if $P_D = b'(P_C) \cup c'(P_B)$.

$$
\begin{array}{ccc}
\overline{A} & \xrightarrow{\text{b}} & \overline{B} \\
\text{c} \downarrow & (1) & \downarrow \text{c'} \\
\overline{C} & \xrightarrow[\text{b'}]{} & \overline{D}
\end{array}
\qquad\qquad
\begin{array}{ccc}
A & \xrightarrow{\text{b}} & B \\
\text{c} \downarrow & (2) & \downarrow \text{c'} \\
C & \xrightarrow[\text{b'}]{} & D
\end{array}
$$

Proof. Let $P_D = b'(P_C) \cup c'(P_B)$. Then b' and c' are root-preserving. Commutativity of (1) implies commutativity of (2). To show that (2) satisfies the universal property, consider root-preserving morphisms $C \overset{\overline{c}}{\rightarrow} E \overset{\overline{b}}{\leftarrow} B$ such that $\overline{c} \circ c = \overline{b} \circ b$. By the universal property of (1), there is a unique morphism $d : \overline{D} \rightarrow \overline{E}$ such that $d \circ b' = \overline{c}$ and $d \circ c' = \overline{b}$. We have to show that d is root-preserving. Consider a root x in D. Then, by assumption, $x \in b'(P_C)$ or $x \in c'(P_B)$. Without loss of generality, assume the former. Then there is a root x' in C such that $b'(x') = x$. Hence $d(x) = d(b'(x')) = \overline{c}(x')$, so $d(x)$ is a root in E. $\square$

Lemma 2 allows us to extend the algorithmic pushout construction given in Section 2.1.2 (repeated below) to rooted graphs. Given a rooted graph G and a root-preserving injective morphism $g\colon L \rightarrow G$ satisfying the dangling condition, a direct derivation $G \Rightarrow_{r,g} H$ is constructed as follows:

1. To obtain D, remove all nodes and edges in $g(L - K)$ from G. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_D(g_V(v)) = \bot$. Define $P_D = P_G - g_V(P_L - P_K)$.
2. Add all nodes and edges, with their labels, from $R - K$ to D. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(E) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.
3. For all $v \in V_K$ with $l_K(v) = \bot$, define $l_H(g_V(v)) = l_R(v)$. The resulting graph is H.
4. Define $P_H = P_D \cup h_V(P_R - P_K)$ where h is the morphism $R \rightarrow H$.

The first and fourth steps extend the original construction with the specification of the root nodes of D and H. This construction can be characterised by a double-pushout in the category of rooted graphs and root-preserving morphisms, where the left pushout is natural. We illustrate this with an example. Consider the rule `rooted_rule` at the top of Figure 4.2. Applying this rule to the graph G containing a single root node gives two double-pushouts, shown at the bottom of Figure 4.2 [1]. Only the left double-pushout is natural, and this is the double-pushout obtained by the construction above. Explicitly, $P_D = P_G - g_V(P_L - P_K) = \{v\} - g_V(\{v\} - \emptyset) = \emptyset$, and $P_H = P_D \cup h_V(P_R - P_K) = \emptyset \cup \emptyset = \emptyset$, where v is the node in G.

Now that we have established a DPO-based framework for rooted graphs and rooted rule application, we can trivially extend GP 2 to support rooted host graphs and *rooted conditional rule schemata* (abbreviated to *rooted rule schemata*

[1] By convention, interface nodes are unrooted.

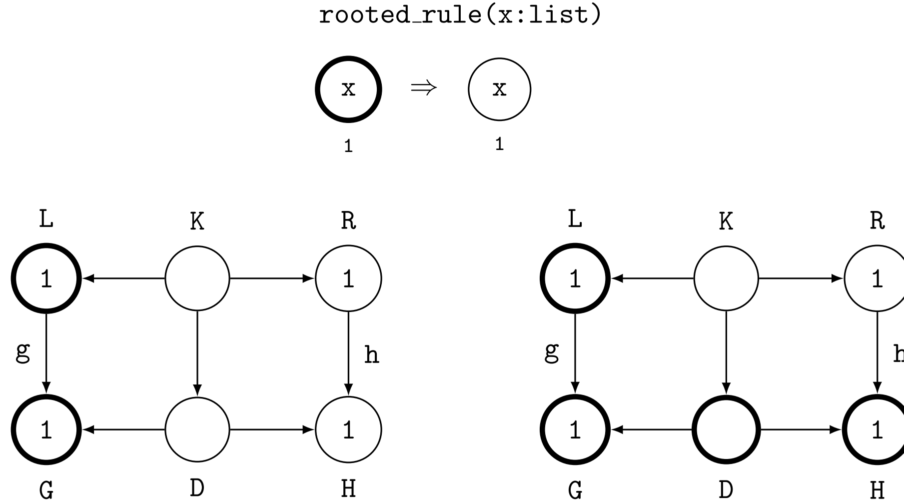


Figure 4.2.: A rooted rule schema and two double-pushouts

or *rooted rules*). We have seen an example of a rooted rule at the top of Figure 4.2. Root-preserving morphisms force rule roots to match only rooted host graph nodes. Consequently, efficient rooted graph matching can be implemented straightforwardly by adding a dedicated data structure to store only the root nodes of the host graph for fast querying, which coordinates nicely with the formal definition of a rooted graph.

The rewards of rooted graph matching cannot be fully reaped if the host graph contains many root nodes. One potential cause is rules that create new root nodes. It is important to classify which classes of rooted rules and rooted graph programs admit a theoretically efficient execution time in order to write fast graph programs. This topic is explored in the rest of this chapter. First we consider matching rules structurally, before adding label matching in order to extend the matching algorithm to GP 2's rule schemata.

4.3. A Matching Algorithm for Rooted Rules

We present a matching algorithm for rooted rules adapted from the matching algorithms in [DP06b; Dod08]. The cited papers assume a single root: we extend the previous algorithms by allowing multiple roots in rules and host graphs. Moreover, it is possible to designate arbitrary nodes as roots, in contrast to using a unique label to identify a root node.

First we introduce some notation used in the algorithm. We write $\mathrm{Dom}(g_V)$ and $\mathrm{Dom}(g_E)$ for the sets of nodes and edges on which a premorphism g is defined. Given partial premorphisms $f, g\colon G \to H$, f *extends* g *by a node* v if $\mathrm{Dom}(f_V) = \mathrm{Dom}(g_V) \cup \{v\}$ and $\mathrm{Dom}(f_E) = \mathrm{Dom}(g_E)$. Also, f *extends* g *by an edge* e if $\mathrm{Dom}(f_E) = \mathrm{Dom}(g_E) \cup \{e\}$ and $\mathrm{Dom}(f_V) = \mathrm{Dom}(g_V) \cup \{s_G(e), t_G(e)\}$. Given a rooted graph $\langle L, P_L \rangle$ and $p \in P_L$, an *edge enumeration* for p is a list of edges $e_1, \ldots, e_n$ such that $\{e_1, \ldots, e_n\}$ is the set of all edges undirectly reachable from p, e_1 is incident to p, and for $i = 2, \ldots, n$, e_i is incident to the source or target of

Rooted Rule Matching Algorithm

Input: rooted graph $\langle G, P_G \rangle$, left-hand side of a rooted rule $\langle L, P_L \rangle$, edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$.

Output: Set A of all injective root-preserving premorphisms $L \to G$.

1: $A \leftarrow \{h \colon L \xrightarrow{par} G \mid \mathrm{Dom}(h) = \emptyset\}$
2: **while** there exists an untagged root $p \in P_L$ **do**
3: $\quad A_0 \leftarrow \{h \colon L \xrightarrow{par} G \mid h$ is injective and root-preserving, and
4: $\qquad$ there exists h' in A such that h extends h' by $p\}$
5: $\quad$ tag p
6: $\quad$ **for** $i = 1$ to n **do**
7: $\qquad A_{p_i} \leftarrow \{h \colon L \xrightarrow{par} G \mid h$ is injective and root-preserving, and
8: $\qquad\quad$ there exists h' in $A_{p_i - 1}$ such that h extends h' by $e_{p_i}\}$
9: $\qquad$ **if** $s(e_{p_i}) \in P_L$ **then** tag $s(e_{p_i})$
10: $\qquad$ **if** $t(e_{p_i}) \in P_L$ **then** tag $t(e_{p_i})$
11: $\quad$ **end for**
12: $\quad A \leftarrow A_{p_n}$
13: **end while**
14: **return** A

Figure 4.3.: The Rooted Graph Matching Algorithm

some edge in $\{e_1, \ldots, e_{i-1}\}$.

The algorithm of Figure 4.4 takes as input a rooted host graph $\langle G, P_G \rangle$, the left-hand side $\langle L, P_L \rangle$ of a fixed rooted rule, and an edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$. We assume that each node in L is reachable from some root, hence the edge enumeration contains all edges in L. The algorithm computes all matches of $\langle L, P_L \rangle$ in $\langle G, P_G \rangle$ by incrementally constructing a set A of partial root-preserving premorphisms $h \colon L \xrightarrow{par} G$. The roots in L are tagged when they are matched; initially they are all untagged.

Proposition 1 (Correctness of Rooted Graph Matching)**.** Given a rooted host graph $\langle G, P_G \rangle$, the left-hand side $\langle L, P_L \rangle$ of a fixed rooted rule schema in which all nodes are undirectly reachable from a root, and an edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$, the Rooted Rule Matching algorithm returns the set of all injective root-preserving premorphisms $g \colon L \to G$.

Proof. First, the algorithm is guaranteed to terminate since $|P_L|$ is finite, and there are only a finite number of premorphism extensions possible as L has a finite number of nodes and edges.

By induction, we show that once the algorithm terminates, A contains all total root-preserving injections $L \to G$. Let $\{p_1, \ldots, p_r\} \subseteq P_L$ be a set of root nodes in L that are not reachable from one another. Define L_i to be the subgraph of L consisting of all nodes and edges reachable from $p_1, \ldots, p_i$. Note that if two or more of L's roots are connected, L_i may contain more than i roots, but it cannot

contain more than i roots from the set $\{p_1, \ldots, p_r\}$ by construction. We show that after the ith iteration of the while loop, the following statement holds:

$$\{h : L_i \to G \mid h \text{ is injective and root-preserving}\} \subseteq A.$$

Without loss of generality, let p_i be the ith root node of L chosen by the algorithm. Consider $i = 1$. First, A_0 becomes the set of all premorphisms that map $p = p_1$ to a root node in G. Then the algorithm enters the for loop. In the first iteration, the premorphism set A_1 becomes the set of all injective, root-preserving extensions to premorphisms in A_0 by e_1, the first edge in the edge enumeration of p_1. If a premorphism in A_0 has no such extension, then it is discarded and can no longer be considered. This process repeats, extending the premorphisms edge by edge and pruning non-injective and non-root-preserving premorphisms until the for loop terminates. At this point, by definition of an edge enumeration, all nodes and edges in L_1 have been considered. A is assigned the set A_n, which is precisely the set of all total injective, root-preserving premorphisms $L_i \to G$. Hence the statement holds.

Next, assume the statement holds for L_k where $k < r$. That is, A contains all total injective root-preserving premorphisms $L_k \to G$. Now consider the $k+1$st iteration of the while loop. p_{k+1} is not reachable from any of $\{p_1, \ldots, p_k\}$ because all root nodes reachable from them were tagged in a previous iteration of the while loop. Therefore p_{k+1} is untagged, and not in the domain of any premorphism in A. A mapping from p_{k+1} to an unmatched root in G is added to all premorphisms in A. Then, as before, nodes and edges are added until all edges in the enumeration for (L, p_{k+1}) and their incident nodes have been matched, after which A becomes the set of injective root-preserving morphisms from $L_{k+1} \to G$.

The program terminates after the rth iteration of the while loop. All nodes in L are undirectly reachable from some root, hence $L_r = L$. It follows that A contains all total injective root-preserving premorphisms $L \to G$. □

Remark 7. We assume that rules are fixed because the context is the execution of graph programs. In algorithm analysis, it is customary for programs (containing rules) to be fixed and running time to be measured in terms of the input (host graph) size. We further assume internal data structures and functions that allow unit-time execution of the following operations:

1. Integer and character comparisons.
2. Adding a variable-to-value mapping to an assignment.
3. Adding a rule-item-to-host-graph-item mapping to a morphism.
4. Mapping a variable to a value within an assignment.
5. Labelling a right-hand side item with the value from a list or string variable that is not repeated in the right-hand side.
6. Checking that an assignment contains a value for a specific variable.

These are all reasonable assumptions that could be supported by standard data structures. We elaborate on points 4 and 5: if lists in host graph labels are stored in a doubly-linked list data structure, a list value in an assignment can be represented by assigning a pointer to the first element of that list.

Theorem 1. Let $L \Rightarrow R$ be a rooted rule such that each node in L is undirectly reachable from some root node. The algorithm Rooted Graph Matching runs in constant time on $L \Rightarrow R$ if there are upper bounds on the maximal node degree and the number of roots in host graphs.

Proof. Consider a host graph G. Let l be the number of roots in L. Let b and r be upper bounds on the node degree and the number of roots in G respectively.

We count the number of times the set of partial premorphisms $L \xrightarrow{par} G$ is updated. There are at most l iterations of the while loop and, within each iteration, at most $m = |E_L|$ iterations of the for loop. Note that both l and m are constants by our fixed-rule assumption.

Consider the execution of the first iteration of the while loop. First, a single root from L is matched with all unmatched roots in G. Since no roots have been matched yet, r partial morphisms are created. Then, in each iteration, either a single edge or an edge and a node is added to the domain of one of more morphisms in the current set. Since node degrees in G are bounded by b, no more than b additions can take place. This gives a worst-case running time of $r + b|A_0| + b|A_1| + ... + b|A_{m-1}|$. The set A_0 contains at most r morphisms, A_1 contains at most br morphisms, etc. It follows that the running time is

$$r + br + b^2r + \ldots + b^m r = r \sum_{i=0}^{m} b^i.$$

Next, the second root of L is matched. One root in G has already been matched, so the maximum size of the new morphism set is $b^m r(r-1)$. Hence, by the same argument as before, the execution time after the second iteration of the while loop is

$$r \sum_{i=0}^{m} b^i + r(r-1) \sum_{i=m}^{2m} b^i.$$

After the l-th and final iteration of the while loop, the total execution time is bounded by

$$r \sum_{i=0}^{m} b^i + r(r-1) \sum_{i=m}^{2m} b^i + \ldots + r(r-1)\ldots(r-l+1) \sum_{i=(l-1)m}^{lm} b^i.$$

□

This is an intimidating expression, but it is a constant. It is not a sharp bound for rules with more than one root node. To simplify the proof, we made some

Rooted Rule Schema Matching Algorithm

Input: rooted graph $\langle G, P_G \rangle$, left-hand side of a rooted rule $\langle L, P_L \rangle$, edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$.

Output: Set A of all pairs of injective root-preserving premorphisms $L \to G$ and and total assignments α.

```
1: A ← {⟨h: L --par--> G, ∅⟩ | Dom(h) = ∅}
2: while there exists an untagged root p ∈ P_L do
3:     A_0 ← {⟨h: L --par--> G, α_h'⟩ | h is injective and root-preserving, and
4:             there exists ⟨h', α_h'⟩ in A such that h extends h' by p}
5:     tag p
6:     Update Assignment(A_0)
7:     for i = 1 to n do
8:         A_i ← {⟨h: L --par--> G, α_h'⟩ | h is injective and root-preserving, and
9:                 there exists ⟨h', α_h'⟩ in A_{i−1} such that h extends h' by e_i}
10:        if s(e_i) ∈ P_L then tag s(e_i)
11:        if t(e_i) ∈ P_L then tag t(e_i)
12:        Update Assignment(A_i)
13:    end for
14:    A ← A_{p_n}
15: end while
16: return A
```

Figure 4.4.: The Rooted Graph Matching Algorithm

assumptions for worst-case running time that are mutually exclusive in the case $|P_L| > 1$. If the while loop is executed once for each root, then no roots in L are connected, hence it is impossible for the for loop to ever execute $|E_L| = m$ times. Conversely, if the for loop executes $|E_L| = m$ times, then all roots must be connected, implying a single execution of the while loop. Therefore, for left-hand sides with more than one root, the derived bound will never be reached.

4.4. Extension to Rooted Rule Schemata

We extend the rooted rule matching algorithm to match GP 2 labels. Specifically, the algorithm must compare expressions of labels in the left-hand side with values in host graph labels, and compute assignments of values to variables. These assignments are used when evaluating the application condition of the rule schema and when calculating the labels of added and relabelled items during rule application.

The revised algorithm of Figure 4.4 takes the same input. However, the output is now a set of pairs. The algorithm incrementally constructs a set A of pairs of partial *morphisms* $h\colon L \xrightarrow{par} G$ and partial assignments α_h. By a *partial assignment* we mean a partial function $\mathrm{Var}(L) \to (\mathbb{Z} \cup \mathrm{Char}^*)^*$, where $\mathrm{Var}(L)$ is

the set of variables occurring in L. The structure of the algorithm is the same. The differences are the contents of the set assigned to the A_i variables, and the two calls to the auxiliary procedure *Update Assignment*. Its purpose is to compare two corresponding labels in existing partial matches and update the assignment if the labels match.

Update Assignment is defined in Figure 4.5. It uses its own auxiliary procedure *Check*, defined in Figure 4.7. In the pseudocode, any right-justified text is a comment, and & operator returns the position in the list of its argument.

Both procedures use the restriction on list and string variables in left-hand side labels as defined in Definition 8. Concretely, lists in L are of the form $a{:}l{:}a'$ where a and a' are possibly empty sequences of atoms and l is an optional list variable. Similarly, string expressions are either a string constant, a character variable or $w.s.w'$ where w and w' are sequences of string constants and character variables, and s is a string variable. To verify a label in the left-hand side with a host graph label x, it suffices to check if x has a prefix that matches with a and a suffix that matches with a'. Then the list variable l can be assigned to the possibly empty remainder of x without further checking. The matching of string expressions is analogous.

Update Assignment iterates over its input, a set of pairs of partial premorphisms and partial assignments. For each pair $\langle h, \alpha \rangle$, it iterates over all untested labels l in the domain of h and compares them with the label of their images $h(l)$. First, the marks of the labels are tested for compatibility. If the marks differ, or l's mark is `any` and h is unmarked, then the labels do not match and the Reject subprocedure is called, which removes the working premorphism-partial assignment pair from the set and exits the inner for loop. Otherwise, a while loop is used to compare the labels one atomic expression at a time.

It is illustrative to demonstrate the key features of the algorithm with an example. Let $x = 1 : 2 : m : 4 : 5$ be a label in the left-hand side, where m is a variable of type `list`. Let $y_1 = 1 : 2 : 3 : 4 : 5$ and $y_2 = 1 : 2 : 5$ be two host graph labels, corresponding to the notation in line 4 of the algorithm. By inspection, we see that x matches y_1 with the assignment $m = 3$. On the other hand, x and y_2 do not match because y_2 contains three atoms while any label which matches x contains a minimum of four atoms. We step through Update Assignment comparing x with both y_1 and y_2 starting at line 8. This is summarised in Figure 4.6. Each row describes an iteration of one of the two while loops in the algorithm. The first five columns of the table are the values of the variables in the pseudocode. Variables a, b_1 and b_2 iterate through the lists x, y_1 and y_2 respectively. They are always at the same position in their respective list. The operations column displays the comparison of the atoms and the variable assignments that precede the following row, with reference to their line numbers in Figure 4.6. We have yet to present the Check procedure. For now it suffices to state that if Check is passed two constants as its first two arguments, it tests their equality.

In the first two iterations both b_1 and b_2 match a because the values are the same integer constant. Once the list variable in x is reached in the third row,

Update Assignment

Input: Set A of pairs of partial injective root-preserving premorphisms $L \to G$ and and partial assignments α.

Output: Set A of pairs of partial injective root-preserving morphisms $L \to G$ and and partial assignments α.

1: **Update Assignment**
2: **for** each (h, α) in the input set **do**
3:     **for** untagged items $l \in \text{Dom}(h)$ **do**
4:         $x \leftarrow \text{label}(l)$; $y \leftarrow \text{label}(h(l))$
5:         $\triangleright$ label $= l_G(x)$ if $x \in V_G$; label $= m_G(x)$ if $x \in E_G$
6:         **if** $(\text{mark}(x) \neq \text{mark}(y)) \vee (\text{mark}(x) = \texttt{any} \wedge \text{mark}(y) = \text{none})$
7:             **then** Reject
8:         atom $a \leftarrow x.first$; atom $b \leftarrow y.first$;
9:         **while** $a \neq$ NULL **do**
10:             **if** a is a list variable **then break**
11:             **if** $b =$ NULL **then** Reject $\triangleright$ $|x| > |y|$
12:             **if** $\neg\text{Check}(a, b, \alpha)$ **then** Reject
13:             $a \leftarrow a.next$; $b \leftarrow b.next$
14:         **end while**
15:         **if** $a =$ NULL **then** $\triangleright$ check if $|x| = |y|$
16:             **if** $b =$ NULL **then exit else** Reject
17:         **else**
18:             atom $temp \leftarrow b$;
19:             $a \leftarrow x.last$; $b \leftarrow y.last$;
20:             **while** a not a list variable **do**
21:                 **if** $\&b = \&temp.prev$ **then** Reject $\triangleright$ $|x| > |y|$
22:                 **if** $\neg\text{Check}(a, b, \alpha)$ **then** Reject
23:                 $a \leftarrow a.prev$; $b \leftarrow b.prev$
24:             **end while**
25:             $\alpha \leftarrow \alpha \cup \{(a.first \mapsto temp), (a.last \mapsto b)\}$
26:             **exit**
27:     **end for**
28:     Tag l
29: **end for**

Figure 4.5.: The Update Assignment procedure

a	b_1	$temp_1$	b_2	$temp_2$	Operations
1	1	-	1	-	L12: Check$(1, 1, \alpha)$ = True L13: $a \leftarrow a.next$; $b \leftarrow b.next$
2	2	-	2	-	L12: Check$(2, 2, \alpha)$ = True L13: $a \leftarrow a.next$; $b \leftarrow b.next$
m	3	-	5	-	L10: a is a list variable: **break** L18: $temp \leftarrow b$ L19: $a \leftarrow x.last$; $b \leftarrow y.last$
5	5	3	5	5	L22: Check$(5, 5, \alpha)$ = True L23: $a \leftarrow a.prev$; $b \leftarrow b.prev$
4	4	3	2	5	L21: $\&b_2 = \&temp.prev$: Reject y_2 L22: Check$(4, 4, \alpha)$ = True L23: $a \leftarrow a.prev$; $b_1 \leftarrow b_1.prev$
m	3	3	-	-	L25: $m.first \mapsto 3, m.last \mapsto 3$

Figure 4.6.: Executions of Update Assignment

the first loop breaks explicitly. A variable *temp* is assigned the current value of b. The purpose of *temp* is to keep track of the start position of the list to which m will be assigned if the labels successfully match. Note that every atom to the left of *temp* in y has been explicitly checked against the corresponding atom in x. If any of these atoms are checked again, the host list is too short to match the rule list. The algorithm continues in the fourth row in which a, b_1 and b_2 store the last element in their respective lists. The check passes and the variables are moved backwards through the list. In the fifth row, b_2 now refers to the predecessor of $temp_2$. This means that the atom 2 in the second position of y_2 has been reached a second time, therefore y_2 must contain fewer atoms than x, and a match cannot exist. y_2 is rejected. On the other hand, y_1 is still in coordination with x as $a = 4 = b_1$. The variables a and b_1 are again moved back through their lists, and the list variable m in x is reached for a second time. This means that all atoms in the list have been compared. The list variable m is assigned the unexamined sublist of y_1 which is precisely $temp : \ldots : b$. In this example, $temp = b$, so m is a list with one atom. m could be assigned a list of arbitrary length, but it suffices to supply the assignment of m with since the other elements can be accessed through the list operators in y.

Some aspects of the algorithm have not been covered by the examples. The first while loop terminates if the end of x is reached (a = NULL) without encountering a list variable (line 9). In this case, if b is not NULL, then y contains more elements than x, hence the two lists cannot be matched and the algorithm calls Reject. Otherwise, the algorithm exits without further action as there are no new variable assignments to add to the mapping (line 16). Similarly, if the end of y is reached (b = NULL) before the end of x, then Reject is called (line 11). We assume that variable-value mappings are implicitly checked for conflicts against

Check

Input: An atomic expression a in a left-hand side label, an atom b in a host graph label, and a partial assignment α.

Output: True if a and b can be matched. False otherwise.

1: **Check**
2: **case a** is
3: integer constant: **if** $b \in \mathbb{Z}$ **then return** $(a = b)$ **else return** false
4: string constant: **if** $b \in \text{Char}^*$ **then return** $(a = b)$ **else return** false
5: character variable: **if** $b \in \text{Char}$ **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$; **return** true
6: **else return** false
7: integer variable: **if** $b \in \mathbb{Z}$ **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$; **return** true
8: **else return** false
9: atomic variable: **if** $b \in \mathbb{Z} \cup \text{Char}^*$ **then** $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$; **return** true
10: **else return** false
11: string expression $w.s.w'$ where w, w': sequence of characters
12: and character variables, s: string variable:
13: **if** $b \in \mathbb{Z}$ **then return** false
14: char $c \leftarrow a.first$; chard $\leftarrow b.first$;
15: **while** $c \neq s$ **do**
16: **if** $d =$ NULL **then** Reject
17: **if** c is a character variable **then** $\alpha \leftarrow \alpha \cup \{(c \mapsto d)\}$;
18: **else if** $c \neq d$ **then return** false
19: $c \leftarrow c.next$; $d \leftarrow d.next$
20: **end while**
21: $temp \leftarrow d$; $c \leftarrow a.last$; $d \leftarrow b.last$
22: **else while** $c \neq s$ **do**
23: **if** $\&d = \&temp.prev$ **then return** false
24: **if** c is a character variable **then** $\alpha \leftarrow \alpha \cup \{(c \mapsto d)\}$;
25: **else if** $c \neq d$ **then return** false
26: $c \leftarrow c.prev$; $d \leftarrow d.prev$
27: **end while**
28: $\alpha \leftarrow \alpha \cup \{(a \mapsto b)\}$ ▷ $s.first \mapsto temp$, $s.last \mapsto d$
29: **return** true

Figure 4.7.: The Check procedure

an existing mapping in the assignment at the point which they are added.

Check takes three arguments. The first argument a is an atomic expression in l, the second argument b is the corresponding atom expression in $h(l)$, and the third is the partial assignment α passed from Update Assignment. Check returns false if the two expressions cannot be matched, otherwise it updates the assignment accordingly and returns true.

In five of the six cases, Check either compares constants or performs a simple type check. If the first argument is a variable, it updates the assignment. The most interesting case is when a is a string expression with a variable. The expressions on either side of the variable are deconstructed as sequences of character constants and character variables. If b is also a string, then they are compared analogously to the list comparison of Update Assignment: moving through both lists and comparing or assigning characters, working backwards after the string variable is located. As with list variables, string variable mappings are specified by assigning locations to the *first* and *last* pointers of the variable.

Proposition 2 (Correctness of Rooted Schema Graph Matching)**.** Given a rooted host graph $\langle G, P_G \rangle$, the left-hand side $\langle L, P_L \rangle$ of a fixed rooted rule schema in which all nodes are reachable from a root, and an edge enumeration $e_{p_1}, \ldots, e_{p_n}$ for each $p \in P_L$, the algorithm Rooted Graph Matching returns the set of all pairs $\langle g, \alpha \rangle$ where $g \colon L \to G$ is an injective root-preserving premorphism and $\alpha \colon \mathrm{Var}(L) \to (\mathbb{Z} \cup \mathrm{Char}^*)^*$ is a total assignment such that $g \colon L^\alpha \to G$ is label-preserving.

Here L^α is the graph obtained from L by replacing each variable x with the value $\alpha(x)$. According to the semantics of GP 2 (see Section section 3.3), g must be label-preserving after this replacement, that is, it must be a graph morphism $L^\alpha \to G$.

Proof. By Proposition 1, the premorphisms in the output pairs are injective and root-preserving. The procedure Update Assignment tests all labels in L against their images in G and generates appropriate mappings from variables to values, hence all assignments are label-preserving. □

4.5. Complexity of Rooted Rule Schemata

In this section, we analyse the complexity of the rooted graph matching algorithm and the complexity of applying a conditional rule schema with a given match. Here we assume that integer operations and character comparisons are computed in unit time, which is consistent with the uniform cost criterion for random access machines, the standard complexity model in algorithm analysis [AHU74; Ski08].

4.5.1. Fast Rule Schemata

Our matching algorithm assumes that each node in a left-hand graph is reachable from some root. This alone does not guarantee that rule schemata can be applied in time independent of the size of the host graph. To achieve this, we need to

impose some restrictions on the form of rooted rule schemata. We will show that, under mild assumptions on host graphs, rule schemata of the following form can be applied in constant time.

Definition 3 (Fast rule schema)**.**
A conditional rule schema $L \Rightarrow R$ is *fast* if:

1. Each node in L is undirectly reachable from some root,
2. Neither L nor R contain repeated list, string or atom variables
3. The condition c contains neither the `edge` predicate nor a test $e_1{=}e_2$ or $e_1{!=}e_2$ where both e_1 and e_2 contain a list, string or atom variable.

The first condition ensures that matches can only occur in the neighbourhood of roots. The second condition makes it unnecessary to check the equality of lists or strings, or to copy them. The third condition rules out tests that require more than constant time in the worst case.

Applying a conditional rule schema $L \Rightarrow R$ to a host graph G requires several phases: finding a root-preserving match of L in G and constructing the induced variable assignment; checking the dangling condition and the application condition; removing items from $L-K$; adding items from $R-K$; and relabelling nodes. In the following we focus on the complexity of the matching phase because, in the worst case, it is far slower than the other phases.

Lemma 3. Given a fast rule schema $L \Rightarrow R$ and a host graph G, the procedure Update Assignment compares each label in L in constant time with the corresponding label in G.

Proof. Let s be the maximum number of characters in a single string expression in L, and let t be the maximum number of non-list variable atoms in a single list expression in L. By our assumption that L is fixed, s and t are constant.

In the worst case, the rule label l is a list containing a list variable and t non-list variable atoms. Each of those atoms is a string expression with a string variable and s characters. The whole list is a valid match to the corresponding host label h, so all characters and atoms are checked.

Let us consider the execution of Check on a string expression as described above. The number of character comparisons, pointer traversals and pointer address comparisons are linear in s. All these operations take unit time. There are t calls to Check, and a single assignment of the list variable to the unevaluated sublist of h. By assumption, this takes unit time. Moreover, the list and string variables do not occur anywhere else in L because $L \Rightarrow R$ is a fast rule schema, so verifying consistency of the assignment also takes unit time. Overall the running time is $O(st)$, a constant. □

Note that replacing the character constants with character variables would not affect the complexity. We assume that adding a character assignment and comparing against an existing character assignment takes unit time.

Using Lemma 3, we can now show that fast rule schemata are matched in constant time if both node degrees and the number of roots in host graphs are bounded. The *degree* of a node v is the sum of the number of edges with source v and the number of edges with target v.

Theorem 2. The Rooted Rule Schema Matching algorithm runs in constant time for fast rule schemata if there are upper bounds on the maximal node degree and the number of roots in host graphs.

Proof. The proof of Theorem 1 calculated an upper bound for the number of times the set of premorphisms is updated. This is constant if the node degree and number of roots in the host graph is bounded. A premorphism update adds at most two items, a node and an edge, so each execution of Update Assignment checks up to two labels for every premorphism in the set. By Lemma 3, these executions take constant time. Therefore the total execution time is bounded above by a constant. □

Given a match of the left-hand side of a fast rule schema, checking the application condition and the dangling condition, and deleting, adding and relabelling items can be done in constant time. Hence we obtain the following corollary of Theorem 2.

Corollary 1. Fast rule schemata can be applied in constant time if there are upper bounds on the maximal node degree and the number of roots in host graphs.

Proof sketch. Consider again a fast rule schema $L \Rightarrow R$ with condition c and a host graph G. By Theorem 2, constructing a premorphism $g\colon L \to G$ and induced variable assignment α (or determining there is no such pair) requires only constant time. We need to prove that the remaining phases of rule schema application can be executed in constant time, too.

By Definition 3, the condition c is a boolean combination of predicates each of which is either (1) a relational operator applied to integer expressions, or (2) a test e_1=e_2 or e_1!=e_2 where e_1 and e_2 do not both contain list, string or atom variables, or (3) a type check `int`(e), `char`(e), `string`(e) or `atom`(e). Under our assumptions on the underlying operations, these checks can be performed in constant time. Predicates of the form in (2) take constant time because no comparisons are made between atom, string or list variables.

The dangling condition for an injective premorphism $g\colon L \to G$ can be checked by comparing the degree of each node v in $L - K$ with the degree of its image $g(v)$. We assume a graph representation where nodes are stored together with their indegree and outdegree. This operation then takes time of order $|V_L|$, a constant.

Given a match satisfying the dangling condition, removing the items in $g(L-K)$ can be executed in time proportional to $|L|-|K|$. Similarly, the addition of nodes and edges takes time proportional to $|R| - |K|$. Finally, relabelling is a constant

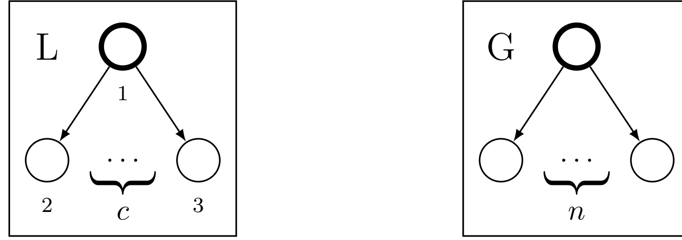


Figure 4.8.: A left-hand side L and a host graph G

time definition because there are no repeated string or list variables in the right-hand side of a fast rule schema. There are at most $|V_K|$ relabellings, so the execution time is proportional to $|V_K|$. □

Some concessions must be made in order for rooted rule schemata application to operate in constant time. Bounded node degree is the first of these, but it is often satisfied in practice. For example, traffic networks, digital circuits and social networks often have an upper bound on the number of edges attached to nodes. Furthermore, the overall time complexity is largely determined by the number of root nodes in both the rule and the host graph. This is to be expected since the number of root nodes available for matching will increase the nondeterminism of the matching process. Indeed, if all host graph nodes were roots, then rooted matching would be identical to traditional graph matching. For this reason, in practice, we aim to minimise the number of root nodes.

4.5.2. Unbounded Node Degree

A topic of interest is the complexity of rooted graph matching when no restriction is imposed on the node degree of the host graph. Consider the pattern graph L and host graph G in Figure 4.8.

As L is fixed in our model, c is a constant. Lifting the bound on node degree means that n is not a constant. Consider generating all premorphisms $L \to G$ with the given matching algorithm. Labels are empty for all items in both graphs. The roots are matched in unit time. There are n premorphisms for the first edge in L. For each of those premorphisms, a further $n-1$ can be generated from the second edge. It is easy to see this gives $\sum_{i=0}^{c}(n-i) = O(n^c)$. This is polynomial in the branching factor. Observe that one could extend G with an arbitrary number of edges incident to non-root nodes without increasing the time taken to find a match: all of L's edges are connected to the root node, therefore edges not incident to the root node in G are not considered during search. Hence the complexity is not a function of the size of the host graph, but a function of the maximum node degree. This example is supported by the following theorem.

Theorem 3. The algorithm Rooted Graph Matching runs in polynomial time for fast rule schemata if there is an upper bound on the number of roots in host graphs.

Proof. Following the proof of Theorem 1, and ignoring the constant factor of label matching for simplicity, we arrive at the following expression for the algorithm complexity.

$$r\sum_{i=0}^{m} b^i + r(r-1)\sum_{i=m}^{2m} b^i + \ldots + r(r-1)\ldots(r-l+1)\sum_{i=(l-1)m}^{lm} b^i.$$

In this case, b is no longer constant as we do not assume a bound on the node degree.

Using the equivalence

$$\sum_{i=km}^{(k+1)m} b^i \equiv \frac{(b^{m+1}-1)b^{km}}{b-1}$$

we can write the above expression as

$$r\frac{b^{m+1}-1}{b-1} + \ldots + r(r-1)\ldots(r-l+1)\frac{(b^{m+1}-1)b^{(l-1)m}}{b-1} = O(b^{(lm+1)})$$

Recall from Theorem 1 that l is the number of roots in L, $m = |E_L|$ and r is the number of roots in the host graph. These are all constants. Therefore the algorithm is polynomial in the degree of the host graph. □

4.6. Summary and Discussion

We have defined rooted graph transformation, which augments host graphs and rule graphs by the addition of an explicit subset of the node set of a graph. The underlying idea is to support efficient graph matching by requiring that a root node must only match a root node in the host graph. This is formalised by the root-preserving morphism. We show that this fits neatly into the DPO foundation of GP 2, and consequently augment GP 2's graphs with root nodes. The abstract nature of rooted graph transformation means that it can be adopted by any graph transformation tool based on the algebraic approach [2].

By defining an abstract matching algorithm to generate all injective root-preserving morphisms from left-hand sides of rules to host graphs, we identified a class of fast rule schema and demonstrate that they are match in constant time under certain restrictions on the host graph: the number of roots is bounded and the maximum node degree is bounded. The first restriction is certainly reasonable: a graph programmer who wishes to exploit root nodes would make an effort to restrict the number of root nodes in the host graph. As the number of root nodes approaches the number of nodes in the graph, the closer matching a fast rule schema comes to matching an rule without root nodes. The second restriction is a more significant concession, but we note that in practice host graphs often have a bound on the node degree. For example, digital circuits, software

[2] We see no reason why rooted graph transformation cannot be transferred to SPO.

models and intermediate representations of functional programs typically have an upper bound on the number of edges attached to nodes.

Theoretical analysis of algorithm complexity considers the worst-case execution time, which is seldom an accurate reflection of the cases handled in practice. We highlight the restrictions on fast rule schema that prevent the comparison or duplication of lists and string. In theory, GP 2's lists are unbounded, but in practice, lists and strings are frequently short enough to be manipulated very quickly. If each node in the left-hand side of a rooted rule schemata is undirectly reachable from some root, we expect it to perform well even when theoretically ineffecient label computations are present. Furthermore, the algorithm presented in this chapter is not practical for finding one match. An efficient implementation would only seek one match with a depth-first search, in contrast to the breadth-first algorithm that incrementally generated all matches. Chapter 6 puts the theory into practice by executing fast rule schemata using the implementation of GP 2 described in Chapter 5.

5. Implementing GP 2

5.1. Introduction

The GP 2 language defined in the previous two chapters is a high-level, graphical language with minimal textual syntax. There is a very large gap from GP 2 source code to machine code, so a direct compilation to assembly, or even a slightly more abstract representation such as LLVM's intermediate representation, would be a tremendous undertaking. Instead we use established higher level languages to execute GP 2 programs, making use of the efficient optimising compilers developed for those languages. With that in mind, this chapter describes two implementations of GP 2. The first is the GP 2 Reference Interpreter, a Haskell program that parses and interprets GP 2 programs. As the name suggests, the goal of this implementation is to provide a reference for future implementations and for language developers. Therefore, the focus is not on performance, but on concise, maintainable code with the capability of generating all solutions to a graph program for verifying other implementations. The second is the GP 2 Compiler, which compiles GP 2 programs into C code, using the C backend and runtime system to execute the program.

Remark 8. Through this chapter we use the term *backtracking* in various contexts. One type of backtracking, which we call *rule backtracking*, is undoing a rule application in order to search for another match because a previous choice resulted in a failure. Another type is *graph backtracking*, where a number of changes made to the host graph are reversed in order to respect the language semantics. The third type is *match backtracking*, where backtracking is performed during an item-by-item search for a match.

We note here that first version of GP, GP 1, was implemented with a low-level abstract machine for graph transformation called the York Abstract Machine (YAM) [MP08a; MP08b]. The YAM is a bytecode interpreter that executes YAM bytecode for graph transformation. This presented us with a decision to make: should we extend GP 1 to support the new features introduced in GP 2, or would we build a fresh implementation from scratch? We decided to abandon the graph transformation abstract machine/bytecode interpreter model in favour of direct compilation to C. Our justification is as follows:

Updated semantics. A fundamental difference between GP 1 and GP 2 is the way the semantics handles nondeterminism. GP 1's semantics [Plu09] are heavily influenced by the nondeterministic behaviour of graph programs, with an emphasis on completeness. For instance, a loop or a condition in a branching statement fails if *every* execution of the loop body or condition respectively fails.

Furthermore, the default behaviour of the GP 1 tool was to return all output graphs, requiring large amounts of rule backtracking for most graph programs. As a consequence, GP 1's implementation was designed with rule backtracking functionality as its chief priority. The YAM was heavily influenced by Warren's Abstract Machine, the stack-based backtracking abstract machine for Prolog. While rule backtracking is desirable in some circumstances, it becomes impractical for complex programs and large host graphs. In addition, it is not especially useful in practice: PROGRES implemented rule backtracking, but its successor, FUJABA, removed the backtracking mechanics "since extensive experiences have shown that it is seldom used" [Fis+00]. In their case, the lack of rule backtracking also enabled the translation of their rules into object-oriented Java code.

GP 2's semantics no longer enforces rule backtracking. The reason for this is to allow for a more efficient implementation. This becomes clear upon examination of the behaviour of conditional branching statements. If an `if-then-else` statement contains a condition that always fails, the GP 1 semantics forces the execution of every nondeterministic execution before taking the else branch. This is a source of great inefficiency in complex GP programs. On the other hand, a GP 2 implementation may pick a single nondeterministic execution path and proceed according to its outcome. This allows some interesting behaviour, for instance the branch taken by an `if-then-else` statement may be chosen nondeterministically if it were possible for the condition to fail and succeed under different execution paths. We do not deem this to be problematic as we expect users to write conditional branches that act in a controlled way. Due to the changes in the language semantics, a sound implementation of GP 2 (that is, one that respects the semantics) would not be required to provide rule backtracking. This distinction in design philosophy means that an abstract machine tailored for backtracking is not the most appropriate way to program a non-backtracking language. Furthermore, the significant reduction in non-determinism makes generating equivalent C code more straightforward than a backtracking semantics would allow.

Cutting out the middleman. Our primary goal for this implementation is to investigate how efficiently we can execute graph programs from very high-level source code. A bytecode interpreter such as the YAM causes some runtime overhead in reading and decoding the bytecode. In contrast, a C program generated by a smart compiler has hard-coded information about the rules and control constructs of the source program, requiring little or no interpretation at runtime. Another potential gain is that the generated C code is tailored for a specific GP 2 program, which could be extracted and used in a separate application domain. If used in this way, the GP 2 system acts as a "graph algorithm generator", taking a high-level specification of a graph algorithm and producing C code to execute the algorithm, in a similar way to Bison generating a C parser from a BNF-like specification. Another benefit to the direct-to-C approach is that individual rules and programs can be compiled independently to separate transformation units. This would speed up compilation for users testing a single rule in a complex

program, an option that is not provided by a bytecode interpreter.

We note that generating bytecode does not remove the possibility of a compiled implementation; bytecode can also be compiled to a C program. This approach has benefits in providing a clean interface between the front-end and the back-end. This is an equally valid approch, perhaps even another means to the same end. However, a two-step translation process may add unnecessary complications. With a semantics that does not force rule backtracking in the implementation, GP 2's imperative-style control constructs map almost directly to the analogous C constructs. Thus the translation step to C isn't especially challenging for a portion of the GP 2 language, and a direct compiler may enable more fine-grained use of C's low-level operations and flexible memory management than a byte code compiler.

Deficiencies of the GP 1 implementation. The YAM takes as input a GP 1 program compiled to YAM bytecode and an internal representation of the host graph. It interprets and executes the bytecode on the host graph. It was written primarily with performance in mind. In that respect, it can be considered a success: it performed very well in a published benchmark with other graph transformation tools on computing a graph transformation problem of exponential complexity [Tae+08].

The highly-optimised implementation comes at the cost of readability, portability and maintainability. One of the goals of the YAM project was to provide a general backend for graph transformation systems, including future versions of GP, theoretically allowing any compatible system to use the YAM by compiling its graph transformation rules and control constructs to YAM bytecode. This is infeasible for several reasons. First, the YAM bytecode and internal graph representation lack a formal syntax or general documentation, making it impractical to generate input to the YAM. Second, the source code of both the compiler and the YAM is extremely difficult to understand and maintain because of its untidiness and lack of documentation. Third, the code is outdated. The implementation was finished in 2008. Since then, new standards and definitions of the implementation languages (Haskell and C) have emerged. Because of this, and for other reasons[1], the source code does not compile. Furthermore, the binaries existing on the University of York departmental machines crash on large computations, including the published GP 1 program for generating Sierpinski triangles [Tae+08]. All of this means that the GP 1 system is no longer usable, and extending it to support GP 2's features, namely recoding the YAM and writing a compiler from GP 2 to the YAM intermediate formats, would be far from a trivial task, and would likely involve a complete reimplementation of the abstract machine along with a compiler backend. While this would be a legitimate road to take from an effiency point of view, from a research perspective it is more interesting to explore a different path.

Before introducing the implementations, we present the underlying textual format

[1]GCC now reports errors when trying to compile the source code that we suspect may have been (permissible but possibly risky) warnings in an older version.

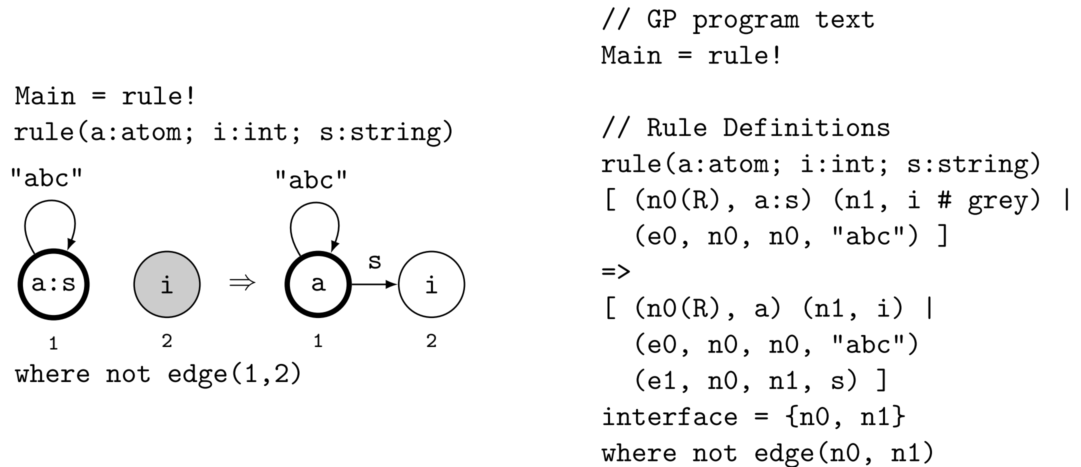


Figure 5.1.: A GP 2 program in graphical and textual format

that acts as input to the Reference Interpreter and to the GP 2 Compiler.

5.2. GP 2 Textual Format

The GP 2 textual syntax is an adaptation of the format first proposed by Sandra Steinert in her PhD thesis [Ste07]. A small GP 2 program with its textual representation is given in Figure 5.1. Textual components of the program such as the procedure names, control structures, variable declarations, lists and conditions are equal in both formats. Graphs are specified by a square-bracketed list of nodes and a list of edges separated by a vertical bar. A node has two components: its identifier and its label. The optional marker `(R)` after a node's identifier is used to declare a root node (similarly `(B)` for bidirectional edges). An edge contains its identifier, the identifier of its source and target, and its label. If an item has a mark, a hash (#) separates the list from the mark. The host graph is represented using the same syntax as the rule graph. The interface is implicitly represented by the numbered nodes in the graphical representation but explicitly stated as a set of node identifiers in the text. The format adheres to the convention that only nodes are contained in the interface. Finally, C-style single-line comments are allowed. A defines the concrete syntax and the context conditions of GP 2's textual format.

5.3. The GP 2 Reference Interpreter

We remind the reader of the declaration in Section **??**. *This section contains extracts from a paper written by multiple authors [Bak+15], including the author of this thesis.*

5.3.1. Reference Interpreters: Uses and Requirements

A reference interpreter for a new programming language such as GP 2 has several potential uses. Each has consequences for the way the reference interpreter is

written and the facilities it provides.

An arbiter for programmers. A programmer working in a new language needs to know whether what they are writing is a valid program, and whether the effect of executing it is the effect they intend. To resolve such issues, the programmer may want to use a reference interpreter as a black box, checking the output it produces given their program as input. Or they may wish to look at a salient part of the source-code for the interpreter, to confirm some aspect of the language they are unsure about.

It follows that a reference interpreter should provide as output at least a report whether a program is valid, and if so a clear representation of the result when it is evaluated. It also follows that the source code for a reference interpreter should be organised in such a way that salient components are easy to identify. For ease of reading it should be written using a consistent style in a modest subset of a suitable high-level language.

An arbiter for implementers. An implementer of a programming language, developing their own interpreter or compiler, needs a standard against which to test the correctness of their implementation. There are two main respects in which any implementation should agree with a reference interpreter as a defining standard. They should agree which programs are valid, and for valid programs they should agree the results of executing them. Like application programmers, implementers too may wish sometimes to use the reference interpreter as a black box, but at other times to consult its internal definitions.

There are additional requirements for this use, bearing in mind the likely development or generation of many test programs. The representation of the reference interpreter's results for such programs should be amenable to automated comparison. This comparison presents particular challenges in GP 2 since behaviour of programs may be non-deterministic, or programs may not terminate, or both. The number of test programs may be large — there may even be arbitrarily many test programs generated dynamically. So although performance is not a design goal for the reference interpreter, its performance should be good enough to make such multi-test comparisons feasible.

A prototype for application developers. If no production compiler has been developed for the language, or none is yet available to an application developer, they may need to use a reference interpreter as an initial development platform. During the development of application programs, errors are common. So, for this use, a reference interpreter should provide not only a check for valid programs, but a rapid check with informative reports of errors. Yet elaborate error handling must not obscure the definitional style in which the interpreter is written. Similarly, it is desirable to have the option of some kind of trace or other informative report to shed light on failures or unexpected results when a program is evaluated. Here again, the machinery must not obscure the basic definitions for evaluation, nor should it impose heavy performance costs when performance of the interpreter has already been sacrificed in favour of simplicity.

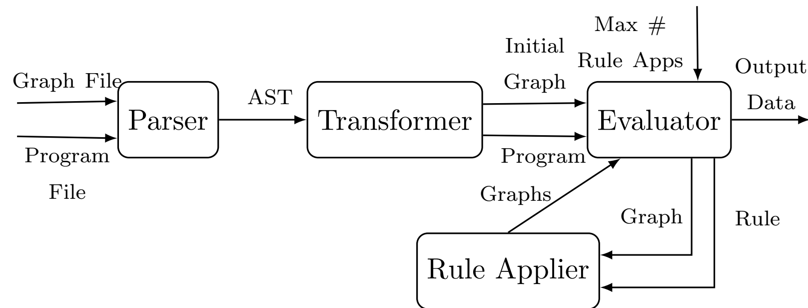


Figure 5.2.: Data flow of the reference interpreter

A prototype for implementation developers. As well as using a reference interpreter to verify correctness, implementers may wish to use it as the starting point in the development of another interpreter or a compiler. The whole course of such a development might even be defined as the successive replacement of interpreter components by alternatives giving higher performance, or richer information, at the cost of greater complexity. The advantage of this approach is that as each replacement is introduced it can be checked as a new component in an already tried system.

This use of a reference interpreter requires a modular design with simple and clearly defined interfaces between components. Concerns should be separated so far as possible, avoiding dependencies that are not strictly necessary. Options for development by successive replacement may be further increased by choosing a host programming system for the reference interpreter that has a well-developed foreign-language interface.

5.3.2. Implementation

We describe the key components of the reference interpreter with the aim of illustrating the simplicity, clarity, and conciseness of the implementation. A basic knowledge of Haskell is useful but not essential to understand the content in the following sections.

Figure 5.2 shows a data flowchart of the reference interpreter. It takes three inputs: (1) a file containing the textual representation of a GP 2 program, (2) a file containing the textual representation of a host graph, and (3) an upper limit on the number of rule applications to be made before halting program execution. It runs the program on the host graph, traversing either all nondeterministic branches of the program or a single branch, at the behest of the user. The output data is a complete description of all possible outputs.

The interpreter contains approximately 1,000 lines of Haskell source code. Figure 5.3 shows the module dependency structure of the interpreter and an indication of module sizes.

Parser. The parser has two components: (1) a host graph parser and (2) a program text parser. Each individual parsing function takes a string as input and attempts to match a prefix of the string to a particular syntactic unit. It

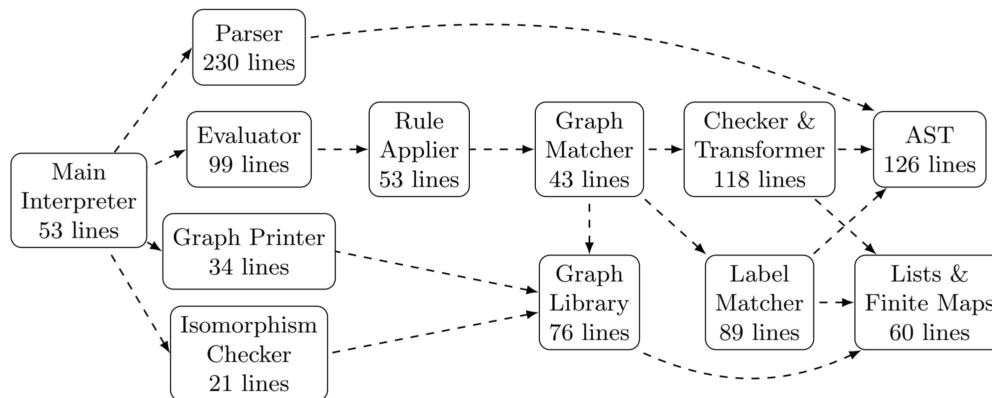


Figure 5.3.: Module dependencies. A module points to any modules on which it depends. Line counts exclude blank lines and comment-only lines

uses a library of parser combinators. Their purpose is to neatly compose the parsing functions to cover standard parsing requirements such as alternation and repetition. The parsing code is very similar in appearance to GP 2's context-free grammar: each nonterminal of the grammar is represented by a Haskell function that parses the right-hand side of the grammar rule. For example:

```
gpMain :: Parser Main
gpMain = keyword "Main" |> keyword "=" |> pure Main <*>
         commandSequence
```

The operators `|>` and `<*>` are binary functions: `|>` ignores the output of its left parser and `<*>` sequences two parsers. Applications of `keyword` recognise and discard a string argument, and `commandSequence` is another parsing function. `Main` is a data constructor for the main node of GP 2's abstract syntax tree.

Checking & Transformation. The checking and transformation phase extracts semantic information from the AST, such as the types of variables specified in a rule schema's parameter list, and transforms both rule graphs and the host graph into the data structure defined in the graph library. The internal graph representation is a pair of maps from keys to labels for each of nodes and edges separately. Node keys are integers. Edge keys are triples: source key, target key and an integer. Node and edge labels are encoded into the node and edge data types. Operations on graphs are concisely represented using Haskell functions from the Haskell library `Data.Map` which implements maps efficiently as balanced binary trees. Node and edge enumeration functions also support the use of Haskell's strong list-processing.

Label Matching. The label matching algorithm establishes whether a label from a rule's left-hand side can be matched with a label from the host graph. It takes as input the current *environment*, the set of bindings for label variables, and the two labels to be compared.

GP 2's marks are encoded as an abstract data type and are directly comparable. Lists are naturally encoded as Haskell lists, where each element is a GP 2 atom.

Atoms occurring in the host graph are constants (integers, characters or strings), while rule atoms are either constants, variables or a concatenated string. If a match binds a variable, the binding must define a compatible extension of the environment.

When comparing atoms, the interesting case occurs if a list variable is encountered. Since at most one list variable is allowed in a left-hand side, it is bound to a host-label segment of determined length, by comparing the lengths of the remainder of the rule label and the remainder of the host label. Matching fails if too few host atoms remain.

Graph Matching. Given a rule graph L and a host graph G, the graph matcher lazily constructs a list of `GraphMorphisms`. A `GraphMorphism` is a data structure containing an environment, a mapping between nodes in L and the corresponding nodes in G, and a similar edge map. We use association lists to represent these small mappings, for simplicity and amenability to list processing. Morphisms are generated in two stages. First the candidate `NodeMorphisms` are identified, where a `NodeMorphism` is an environment and a node mapping. For each such `NodeMorphism`, the matcher searches for compatible edge mappings and environment extensions to form a set of complete `GraphMorphisms`.

For each node $l_k \in L$, the matcher constructs the list of all host nodes $[h_{k_1}, \ldots, h_{k_m}]$ that match l_k with respect to label matching and rootedness. An environment is paired with each host node. The result is a list of lists $[[h_{1_1}, \ldots, h_{1_m}], \ldots, [h_{n_1}, \ldots, h_{n_m}]]$ where n is the number of nodes in L. A candidate node mapping is found by injectively selecting one item from each list. The final step is to test each candidate mapping for compatibility with respect to its environment. Haskell's list comprehensions are perfectly suited for this task: the list of lists is computed with a single nested list comprehension, while a second list comprehension is responsible for collating the valid candidate mappings.

For each edge in L, we use a candidate node morphism to determine the required source and target for a corresponding edge in the host graph. The list of candidate host edges is the list of host edges from that source to that target. Each rule edge is checked against each candidate host edge for label compatibility, supported by the environment passed from the node morphism.

Rule Application. Each of the `GraphMorphisms` produced by the graph matcher is checked against a *dangling condition* and any rule conditions. If these checks succeed, the rule application is performed in the following steps: delete edges, delete nodes, relabel nodes, add nodes, relabel edges, add edges. For relabelling, variables take their values from a `GraphMorphism`'s environment.

The dangling condition can be elegantly expressed as follows.

```
danglingCondition :: HostGraph -> EdgeMatches -> [NodeId] -> Bool
danglingCondition h ems delns =
         null [e | hn <- delns, e <- incidentEdges h hn \\ rng ems]
```

The second argument is an edge map, obtained from a `GraphMorphism`. The third argument is the set of nodes deleted by the rule. The function body specifies

that no host edge e incident to any deleted node n may lie outside of the range of the edge map ems.

The Evaluator. The evaluator applies a GP 2 program to a host graph, subject to an upper bound on the number of rule applications. Often the same graph can be reached through several distinct computational branches. Therefore, when program execution is complete, an isomorphism checker is used to collate the list of output graphs into its isomorphism classes. The output is as follows:

1. A list of unique output graphs, up to isomorphism, with a count of how many isomorphic copies of each graph were generated.
2. The number of failures.
3. The number of unfinished computations. A computation is unfinished if the bound on rule applications is reached before the end of the main command sequence.

During program execution the evaluator maintains a list of GraphStates, one for each nondeterministic branch of the computation so far. A GraphState is one of: (1) a graph with its rule application count, (2) a failure symbol with its rule application count, and (3) an unfinished symbol. Each GP 2 control construct is evaluated by a function that takes as input a single GraphState and some program data, returning a list of GraphStates. Only the application of a rule can yield a GraphState with a changed graph. The rule application process is the workhorse of the interpreter, so here by way of illustration is the top-level defining equation for the evaluation of a rule-call command:

```
evalSimpleCommand max ds (RuleCall rs) (GS g rc) =
   if rc == max then [Unfinished]
   else case [h | r <- rs, h <- applyRule g $ ruleLookup r ds] of
        [] -> [Failure rc]
        hs -> [GS h (rc+1) | h <- hs]
```

Here max is the rule application bound, ds is a list of the rule and procedure declarations in the GP 2 program, rs is a list of rules, and GS g rc is the current graph state. GS is the GraphState constructor, g is the working host graph, and rc is the number of rules that have been applied to g. The case-subject list comprehension can be read as, "for all rules r in rs, apply r to g and produce the list of all output graphs h." Each individual rule application may produce multiple output graphs; the list comprehension gathers every possible output into a single lazily-computed list. If the computed list is empty, then no rule in rs was applicable, and the list containing the single GraphState Failure is returned. Otherwise, the output graphs are placed into a fresh list of GraphStates, each with an incremented rule-application count.

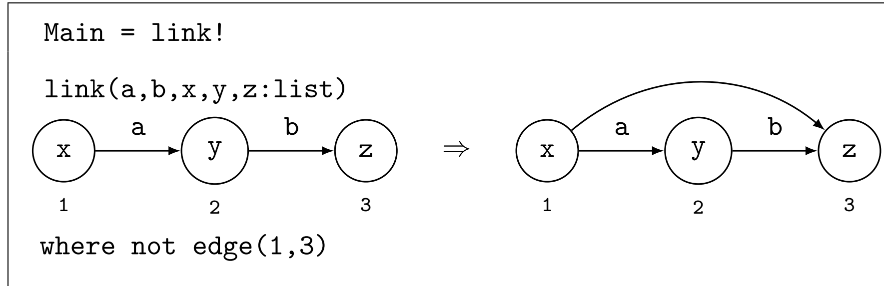


Figure 5.4.: A GP 2 program for transitive closure

# nodes	Single Result (s)	All Results (s)
5	0.01	0.44
10	0.04	>5m
20	1.67	-
30	14.39	-
40	66.31	-
50	>5m	-

Table 5.1.: Reference interpreter results for the transitive closure program

5.3.3. Performance Evaluation and Conclusions

Though not tuned for speed, the interpreter must run fast enough to allow its use as a practical tool. While we wish to illustrate the practicability of the interpreter, its performance is not a significant part of this thesis. Therefore, we present only a sample of the results and supplementary discussion from the paper [Bak+15]. The results presented here concern the transitive closure program of Figure 5.4. The program is very simple, highly nondeterministic, and the output graphs produced by the program are easy to verify. We used directed acyclic paths, or linear graphs, to test the interpreter.

We compiled the interpreter using the Glasgow Haskell Compiler[Tea] version 7.6.3 with optimisations and profiling support enabled:

```
$ ghc -O2 -prof -fprof-auto -rtsopts -o gp2 Main.hs
```

All figures reported were obtained using a quad-core Intel i7 clocked at 3.4GHz, with 8GB RAM, running 64-bit Ubuntu 14.04 LTS with kernel 3.13.0. The number of processor cores should not have a significant effect on the measured performance of the single-threaded GP 2 interpreter. We ran benchmarks using the following command

```
$ timeout --foreground 5m time \
      gp2 +RTS -p -sgc.prof -RTS $GPOPT $PROG $GRAPH 10000
```

Table 5.1 summarises the results for the reference interpreter on the transitive closure program. The timings presented are the sum of user and system time reported by the UNIX `time` command. The extra costs of evaluating a program in all-result mode go beyond those of generating all possible output graphs; the

interpreter must also test them for isomorphism. Unsurprisingly, execution time increases sharply with increasing size of host graph, putting many of the computations that completed in single-result mode beyond our five-minute execution-time limit. This is especially potent for highly non-deterministic programs such as transitive closure. In fact, there is a factorial growth in complexity as the host graphs get larger: the linear graph with 5 nodes computes 866 output graphs before isomorphism checking.

To our knowledge, none of the existing graph transformation systems has a published implementation in the same spirit as our reference interpreter, making this tool a novelty in the field. Indeed, it is quite striking that we have managed to implement a graph transformation language in around 1,000 lines of code using the lazy functional language Haskell. We have taken every opportunity to use a Haskell strength — lazy list-processing, and in particular list comprehensions for generate-and-test style definitions — to achieve this conciseness. However, despite our observations about error reports and traces, we concede that our current interpreter provides only a bare minimum in this respect. When working with the interpreter, we have had some unexpected results. For instance, the implementation raised the question of how root nodes ought to be treated in subtle cases, such as an interface node being rooted on one side of the rule and being unrooted on the other side. Occasionally, the practical consequences of a crisp semantic definition may be surprising to programmers, or it may pose challenges for an efficient implementation. We have found that our reference interpreter can shed helpful light in such instances.

As the results show, the interpreter is efficient enough for practical use in testing, both by GP 2 programmers and by the developers of other GP 2 implementations. Our main reservation here concerns all-results mode. Used in this mode, the interpreter can require very long execution times and all the memory our machines have available. One remedy might be to check for isomorphism or other equivalences between intermediate graphs, compacting the state-space. However, the extra machinery would complicate the interpreter, and it could demand even more space in some cases. Instead, our likely solution will be to build up a standard set of test programs. We can first run each test (for several days, if necessary) on a powerful machine to produce the set of all possible output graphs up to isomorphism. Our isomorphism checker, though simple, is efficient enough for rapid subsequent checking of single results produced by another implementation.

5.4. Experimental Environment

All experiments reported after this section were conducted on a quad-core Intel i5-2300 clocked at 2.8GHz with 8GB of RAM. The operating system is 64-bit Ubuntu 14.04 with Linux kernel 3.13.0.

The C code executed to obtain running times and memory use was compiled by the GNU Compiler Collection [Sta01] version 4.8.4 with the following optimisation and warning flags:

```
gcc -O2 -fomit-frame-pointer -Wall -Wextra
```

Reported running times are the sum of the user time and the system time from the UNIX `time` command. Maximum heap usage and total heap usage are obtained from Valgrind [NS07] version 3.10.0, executed by the command `valgrind --tool=exp-dhat` followed by the name of the executable.

5.5. GP 2 System Architecture

A broad picture of the GP 2 system architecture for a compiled implementation is given in Figure 5.5. We give an overview of the components before covering the core of the system, the GP 2 Compiler and the GP 2 Library, in the subsequent sections.

GP 2 Editor. The graphical editor is the interface between the user and the compiler. Users can construct graph programs graphically, using the mouse to construct graphs and typing the program text, labels and conditions. A prototype editor for GP 2 has been implemented as part of a Master's project at the University of York [Ell13]. It was designed with usability in mind; it features a tutorial to introduce the user to the tool and to GP 2 programming. Furthermore, it uses the Open Graph Drawing Framework (OGDF) C++ library for host graph visualisation. The editor communicates with the compiler via the textual format for programs and host graphs. At the time of writing the implementation is incomplete: the editor has been partially integrated with the GP 2 compiler, but there remain some bugs in the code base.

GP 2 Compiler. The compiler receives the text files specifying the GP 2 program and the host graph from the editor. It is responsible for syntax checking and semantic checking these files and generating C code to execute the program on the host graph. The lexical analysis and parsing is conducted by a Bison and Flex generated parser [Lev09]. A parser generator was used for ease of development and maintainability. Performance is not a significant consideration at compile time, but nevertheless Bison-generated parsers should be faster than a handcoded parser barring serious optimisations. The code generation phase is the most significant part of the compiler and will be deconstructed in the remainder of the chapter.

GP 2 Library. The data structures and operations used by the generated code are collectively referred to as the GP 2 Library. It is a collection of C modules containing data structures and functions described in the next section. The host graph parser, a Bison/Flex-generated parser used to read the host graph file at runtime, is also a part of the library.

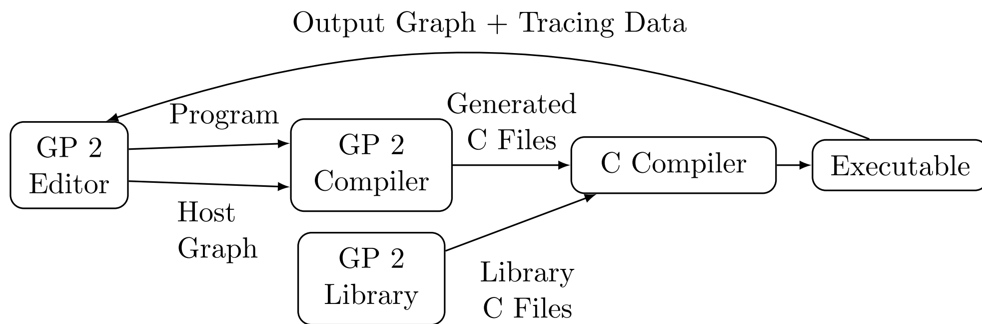


Figure 5.5.: GP 2 system architecture

5.6. Data Structures

5.6.1. Host Graphs

A graph structure stores node and edge structures in handcoded dynamic arrays. The initial array sizes are computed at compile time from the number of nodes and edges in the host graph. For large host graphs this is the least power of 2 greater than the number of nodes or edges. There is a minimum size to reduce overhead in resizing the array when executing graph programs that start with a small or empty host graph since graph programs could build a potentially large host graph from a relatively small input. Free lists are used to prevent fragmentation of the arrays. Nodes and edges are uniquely identified by their indices in these arrays. The graph structure also stores the node count, the edge count, and a linked list of root node identifiers for fast access to the root nodes in the host graph.

A node structure contains the node's identifier, a root flag, a matched flag, its label, its degrees, and references to its inedges and outedges. Each node structure contains four integers for storing two inedges and two outedges. Additional incident edges are placed in a dynamic array. These arrays are not supported by free lists. The motivation behind this choice of incident edge storage is to limit memory allocation overhead for host graph construction and modification: many common graph classes such as grids, binary trees, and cycles consist mainly of nodes with a small number of outgoing or incoming edges. While this increases the base size of node structures, it is not especially wasteful because in practice, host graphs contain very few isolated nodes. An edge structure contains the edge's identifier, its label, the identifiers of its source and target, and a matched flag. The matched flag of nodes and edges, initially false, is set during matching when a host graph item is paired with a rule graph item. It is used to check if candidate host items have already been matched.

Although the data structure is optimised in some respects, there is nothing that is tailored towards querying host graphs for matching information beyond the bare minimum. GP 1's graph data structure supported complex queries by, for example, storing lists of nodes and edges by label. One could query host graphs to return a list of edges with a specific label outgoing from a specific node. As a consequence, host graph updating becomes slower, but this is significantly

```c
typedef struct HostLabel {
    MarkType mark;
    int length;
    struct HostList *list;
} HostLabel;
```

Figure 5.6.: C data structure for host graph labels.

```c
typedef struct HostAtom {
    char type;
    union {
        int num;
        string str;
    };
} HostAtom;
```

Figure 5.7.: C data structure for host graph labels and atoms. A HostList is a doubly-linked list.

outweighed by the reduction in search time for matching rules. The underlying philosophy is that in graph transformation, a graph is queried more often than it is updated. The current graph data structure could be improved by supporting similar querying operations. This is achievable by auxiliary data structures that store nodes and edges by their labels.

5.6.2. Host Graph Labels

The definitions of the data structures for host labels and host atoms can be seen in Figure 5.6 and Figure 5.7. A label structure contains an enumerated type for marks (`MarkType`), the length of the list, and a `HostList`. The `HostList` type is a doubly-linked list of `HostAtoms` in order to implement the constant time list matching algorithm from the previous chapter. A `HostAtom` is a union of integers and C strings, equivalent to GP 2's atom type. Storage of lists at runtime may have an impact on performance when manipulating large labelled host graphs. We describe and empirically evaluate two implementations in Section 5.7.

5.6.3. Morphisms

The morphism data structure not only needs to capture the node-to-node and edge-to-edge functions that define a graph morphism (see Definition 3), but also the assignments mapping variables to their values. Thus the data structure used to represent morphisms contains the following four substructures: (1) an array of host node identifiers, (2) an array of host edge identifiers, (3) an array of assignments, and (4) a stack of variable identifiers. The first three items correspond to the mapping functions and the assignment. The assignment's array entries are a pair of a character type ((n)o value, (i)nteger, (s)tring, (l)ist) and a value. The purpose of the stack will be explained shortly.

The library defines three functions to add variable-value assignments. One of these is `addIntegerAssignment`, which takes a morphism, an integer identifier i and an integer k. It adds the assignment $i \to k$ if it is compatible with the existing assignment. This is checked by inspecting the ith index of the assignment array. If no assignment to i exists, signified by the type 'n', then the function updates

```
static unsigned hashHostList(HostAtom *list, int length) {
   unsigned hash = 0;
   int index;
   for(index = 0; index < length; index++) {
      HostAtom atom = list[index];
      int value = atom.type == 'i' ?
                  atom.num : hashString(atom.str);
      hash = ((hash << 5) + hash) + value;
   }
   return hash % LIST_TABLE_SIZE;
}
```

Figure 5.8.: GP 2's list hashing function

the morphism by setting the array entry to (‘i’, k) and returning 1. Otherwise, the morphism contains some assignment $i \to k$. It returns 0 if $k = k'$ and -1 if $k \neq k'$.

Some care is required to properly manage the assignments. The order of variable indices in the assignment array is determined by the order of variable declarations in the rule. There is no guarantee that the variables encountered at runtime follow this order. This causes a complication when match backtracking: if a rule graph item fails to match, only the variables in the label of that item should be removed from the assignment. Variables assigned in the matching of previous items should remain untouched. The stack is used to record assignment indices in the order in which the variables are assigned values. To support this, each node or edge array entry in the morphism contains the number of variables associated with that rule node or rule edge. In this way, backtracking a step is achieved by examining the number of variables associated with the current item, popping that number of items from the stack, and nulling each corresponding assignment entry.

5.7. Host List Storage

The first implementation of host label management allocated memory for host lists on an individual basis, where each host graph item stored its own list. Initial code profiling showed this to be a source of overhead in cases where the same list was being allocated to a large number of items. The second implementation uses a hash table as a central reference point to store lists. Host graph items contain a reference to a hash table bucket instead of a pointer to its own block of memory. In this way common lists are shared, reducing overhead caused by heap management and list copying. However, it is unclear if this is a significant performance gain in general, and whether extensive list sharing could be costly in some situations. We tested both list storage implementations with two programs: Sierpinski triangle generation, and Euler cycle generation. Both programs perform extensive label manipulation (see subsection 5.7.1 and subsection 5.7.2).

A good hash function for GP 2 lists should avoid collisions for typical GP 2 host

graphs. The labelled GP 2 programs in this thesis operate on integer-labelled graphs. At a bare minimum, single integers should hash to unique hash table entries up to the size of the hash table. Ideally, the hash function should consider all elements of the list and demonstrate a good distribution for strings. The GP 2 list hashing function is shown in Figure 5.8. It is an adaptation of Dan Bernstein's djb2 string hashing algorithm [Ber]. The original algorithm starts with a hash value $h = 5381$ and iterates over each character of the string, performing $h = h * 33 + c$ at each step, where c is the character's ASCII value. These constants were established through experimenting on hashing typical English strings which are not evenly distributed over the entire character set and do not have a small length. Instead of iterating over the characters of a string, we iterate over the elements of the list. The code `((hash << 5) + hash)` bit shifts the value of *hash* five places to the left, multiplying it by 32 (2^5). Adding *hash* to the result gives the desired product of 33. The value c is dependent on the atom's type: for integers it is simply the integer value, and for strings it is the djb2 hash of the string (computed with the `hashString` function in the code fragment). We use an initial hash value of 0. It immediately follows that the single integer lists map to unique hash table slots modulo the size of the hash table. The hash table stores 100,003 buckets: the first prime number over 10^5. This number is arbitrary; we choose it because the host graphs we use for testing typically peak at 10^5 nodes.

5.7.1. Case Study: Generation of Sierpinski triangles

The Sierpinski triangle is a triangle-shaped fractal [PJS04], a self-similar geometric structure. An algorithm for constructing increasingly close approximations to the Sierpinski triangle is described below.

1. Start with an equilateral triangle.
2. Draw a line between the midpoints of each of its sides to form four smaller congruent triangles. Remove the middle triangle.
3. Repeat the previous step with each remaining triangle.

Generating Sierpinski triangles was the subject of a case study for graph transformation tools [Tae+08]. Figure 5.9 demonstrates that the translation to graphs is straightforward. The initial graph is an equilateral triangle, represented by three nodes and three edges. Each *Sierpinski step* is represented by embedding a new three-node triangle in the centre of a previous triangle.

The problem exhibits exponential growth, making it a good performance benchmark for graph transformation implementations. It is accessible to graph transformation tools regardless of their application domain because the criterion for a valid solution is purely structural. Finally, the problem can be expressed as a straightforward and short algorithm which should admit relatively small solutions with simple graph transformation rules. The challenging part of the problem is forcing the Sierpinski step to match in the right place. The GP 2 solution in

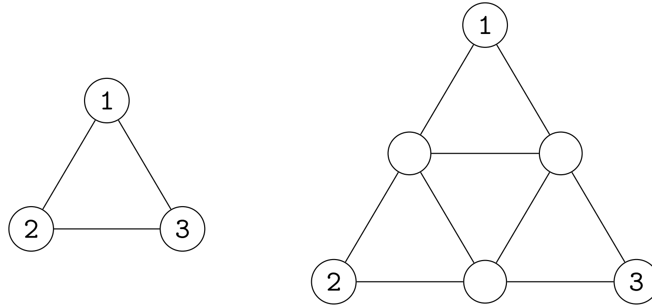


Figure 5.9.: Initial triangle and first generation of the Sierpinski triangle

Figure 5.10 relies on integer labels to force `expand` to be matched in the correct places. This makes the program relevant to the context of this section: as the generation size increases, so do the number of nodes and edges with the same integer label. Although a single size-1 list uses a very small amount of memory, the total space occupied by labels will grow exponentially with the generation number if one list is stored per host graph item. In contrast, if the lists are stored centrally, the space occupied by the lists is constant.

The program `sierpinski` assumes a host graph containing a single root node whose label is the generation number k of the desired output triangle. The rule `init` creates the initial triangle and appends 0 to the label of the root node. Each iteration of the outer loop increments the second integer of the root node and performs a Sierpinski step with the rule `expand` as much as possible. Termination is controlled by the condition of `inc`: when `inc` has been applied k times, the two integers will be equal, and the condition fails. Each application of `expand` performs a Sierpinski step: the small triangle on the left is expanded into the large triangle on the right. Note that the corners of the small triangle are also the corners of the large triangle as denoted by the node identifiers. Hence the expansion takes place inside the original triangle. The matching location is controlled by the integer `y`: the matched triangles are those whose top node is labelled with the current generation number. The expansion step labels the top three nodes of the large triangle with the next generation number for the following loop iteration. The GP 2 solution is identical to the GP 1 solution presented in [Tae+08] with the exception that the control node is rooted and the difference in notation for list concatenation. Figure 5.11 shows a third generation Sierpinski triangle from the GP 1 GUI.

Table 5.2 and Figure 5.12 show the experimental results. For both implementations, the runtime grows exponentially, but the higher generations reveal that the list hashing is faster by a constant factor, a significant amount of time for generations 11 and 12. The effects on heap usage are more interesting. One thing to note is the difference between the total and maximum memory use of the different implementations. Using a hash table to store the lists manages the memory very efficiently, evidenced by the constant difference between maximum heap usage and total heap usage over all generations. In contrast, the gap between total heap and maximum heap grows with the generation when the hash

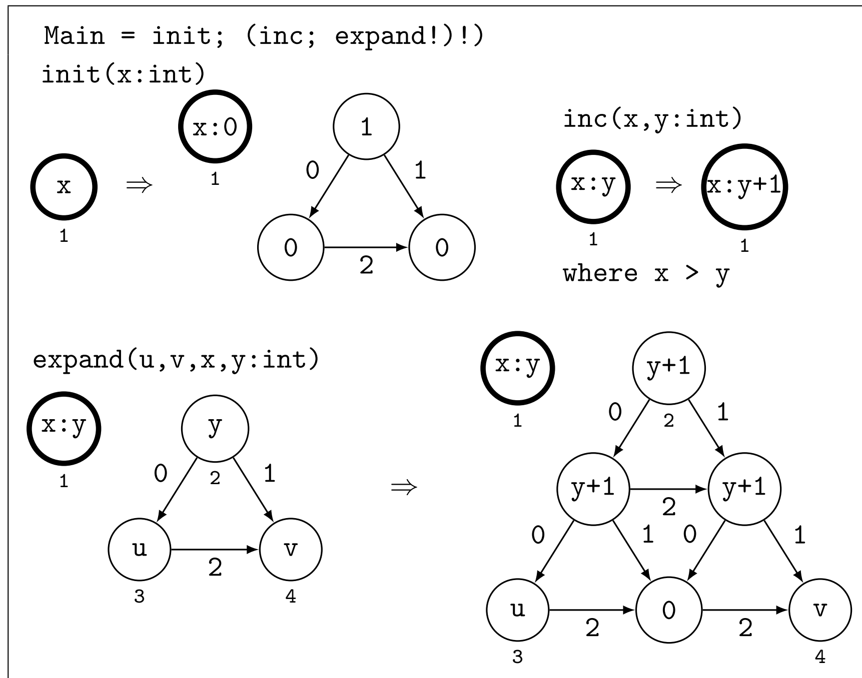


Figure 5.10.: The program sierpinski

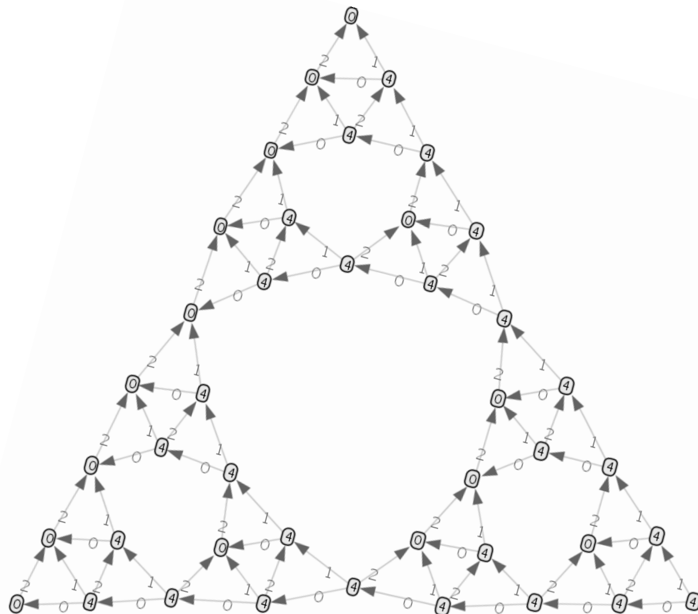


Figure 5.11.: Third generation Sierpinski triangle

	No Hashing			List Hashing		
Gen	Runtime	Max	Total	Runtime	Max	Total
5	0.003	0.17	0.198	0.004	0.909	0.911
6	0.006	0.558	0.641	0.006	1.174	1.177
7	0.017	1.293	1.539	0.017	1.542	1.544
8	0.062	4.529	5.266	0.048	3.676	3.679
9	0.402	10.798	13.005	0.318	6.639	6.412
10	5.253	37.864	44.479	2.568	23.784	23.787
11	83.208	-	-	24.959	-	-
12	793.208	-	-	244.627	-	-

Table 5.2.: Experimental results of `sierpinski` using two list storing implementations. Runtime is given in seconds. Maximum and total heap use is given in megabytes

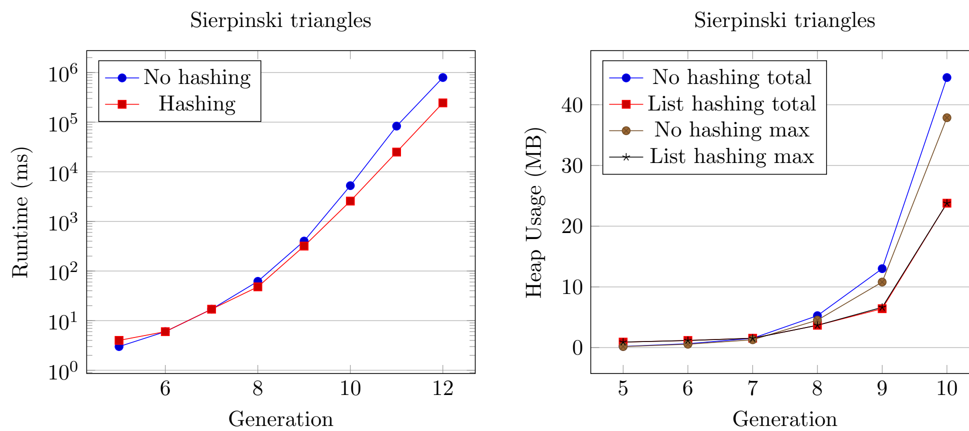


Figure 5.12.: Plots of the `sierpinski` experimental results

table is not used. This is because lists are discarded whenever a single node or edge is relabelled, whereas hash table entries are only discarded when the last node or edge with a particular label is deleted or relabelled. The hash table implementation uses significantly more memory for smaller generations because the constant size of the hash table outweighs the memory used by node and edge labels for small Sierpinski triangles. On the other hand, the growth in space is slower when the hash table is used. By generation 8, the memory used by the non-hashing implementation exceeds that of the hashing implementation. To put this into perspective, the generation 8 triangle contains 9,843 nodes and 19,683 edges, not an exceptionally large graph, but the list hashing approach uses about a third less memory.

5.7.2. Case Study: Computing Euler Cycles

An *Euler cycle* is a directed cycle that visits all edges of a graph exactly once. A graph is *Eulerian* if it contains an Euler cycle. The GP 2 program `euler` [Plu12] of Figure 5.13 takes an non-empty atomic-labelled Eulerian host graph as input and outputs an Euler cycle, represented by edge labels. Of course, few

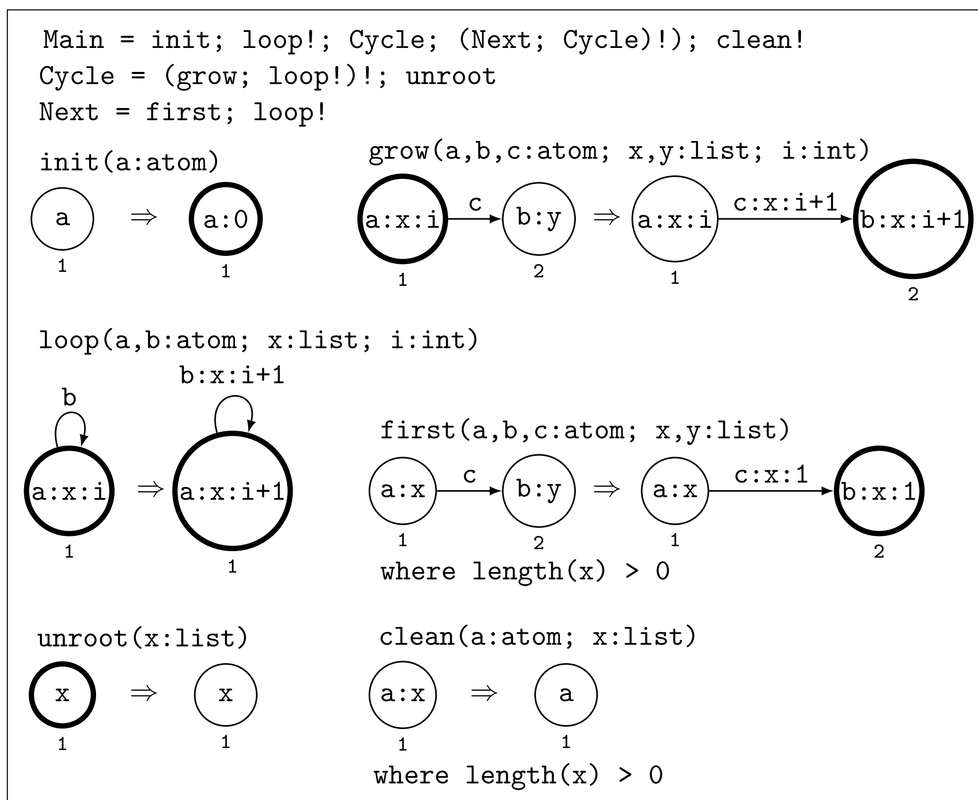


Figure 5.13.: The GP 2 program `euler`

host graphs are Eulerian, but one can use the property that a graph is Eulerian if and only if it is connected and every node has the same indegree and outdegree [BJG08] to test a host graph with a GP 2 routine before running `euler`. We do not present such a routine here because it is not relevant to the current discussion.

The Euler cycle in the output graph is represented by integers appended to the original edge label in the order of the cycle. The Euler cycle is computed by walking the host graph with the rules `grow` and `loop`. The source node of an edge stores the current number in the cycle which is used to label its outgoing edge with the correct integer. Nondeterminism of rule application means that there is no guarantee that the first execution of `Cycle` will traverse all the edges in the graph. In this case, the rule `first` attempts to find an unvisited edge. If one exists, then the subsequent cycle is inserted into the current cycle by extending the length of the list.

The example of Figure 5.14 [Plu12] illustrates the algorithm. The left graph is the host graph after the application of `init`. The first execution of `Cycle` applies `grow` to the edges $1 \to 2$, $2 \to 4$, and $4 \to 1$ and unroots the root node, giving the middle graph. The procedure `Next` searches for unvisited edges sourced at a visited node by the rule `first`. $2 \to 3$ is such an edge. The list appended to the new cycle edge is the "tail" of the source node (1) followed by 1. The inserted cycle is formed by the edges labelled $1 : 1 : x$, where x specifies the order of the edges. The program terminates with the right graph. Reading the edge labels, we see that the program has computed the Euler cycle $1 \to 2 \to 3 \to 4 \to 2 \to 4 \to 1$.

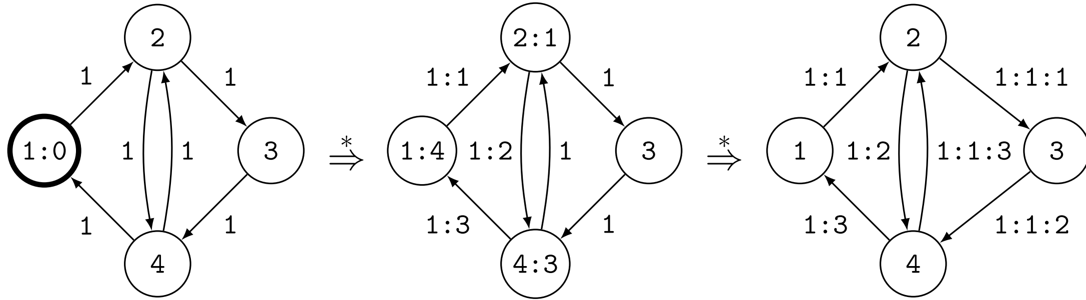


Figure 5.14.: Example run of `euler`

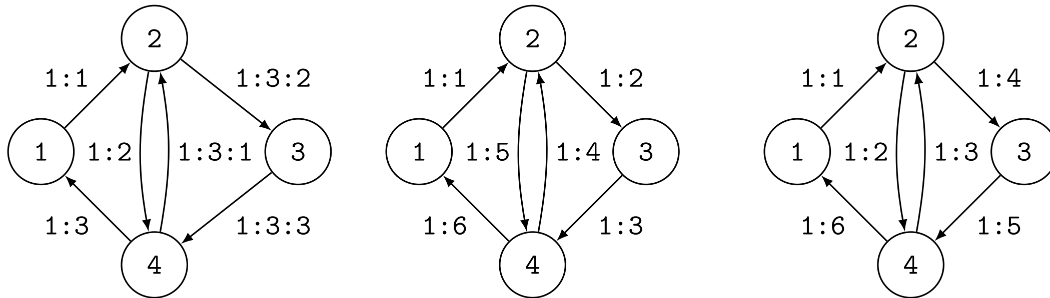


Figure 5.15.: Alternate output graphs of `euler`

The program could have generated three other Euler cycles from the same start node. Note that `first` could have matched the edge $4 \to 2$, resulting in the cycle $1 \to 2 \to 4 \to 2 \to 3 \to 4 \to 1$. As shown in the left graph of Figure 5.15, the edges in the inserted cycle are labelled $1 : 3 : x$. The other two graphs illustrate the Euler cycles computed if the edges are chosen in such a way that all edges are traversed in the first execution of `Cycle`.

Star cycles were used to benchmark the euler cycle program. A star cycle $SC_{k,n}$ is a collection of k cycles of size n, with $k-1$ cycles connected to a central cycle by a pair of edges. All edges are labelled 1, and nodes in each cycle are labelled from 1 to n. Star cycles were chosen because they are non-trivial euler cycles that are relatively easy to generate with a GP 2 program. We also wanted the force the program to make some cycle insertions for a greater variety of list lengths. This can be seen in Figure 5.16, the output graph obtained after executing `euler` on the host graph $SC_{4,4}$. The program and host graphs are interesting with respect to list storage because of the label distribution of the host graph over time. Initially, all edge labels are equal and there are at most n node labels. As the Sierpinski case study demonstrated, significant list sharing benefits the use of a hash table to store lists. However, the lists change over time. In particular, each edge label is unique in the output graphs which is equally memory-demanding for both implementations. During program execution the nodes are assigned intermediate lists which are likely to be distinct, but they are relabelled to their original values before the program terminates. Because of the dynamic relabelling behaviour, it is hard to predict the disparities in runtime and

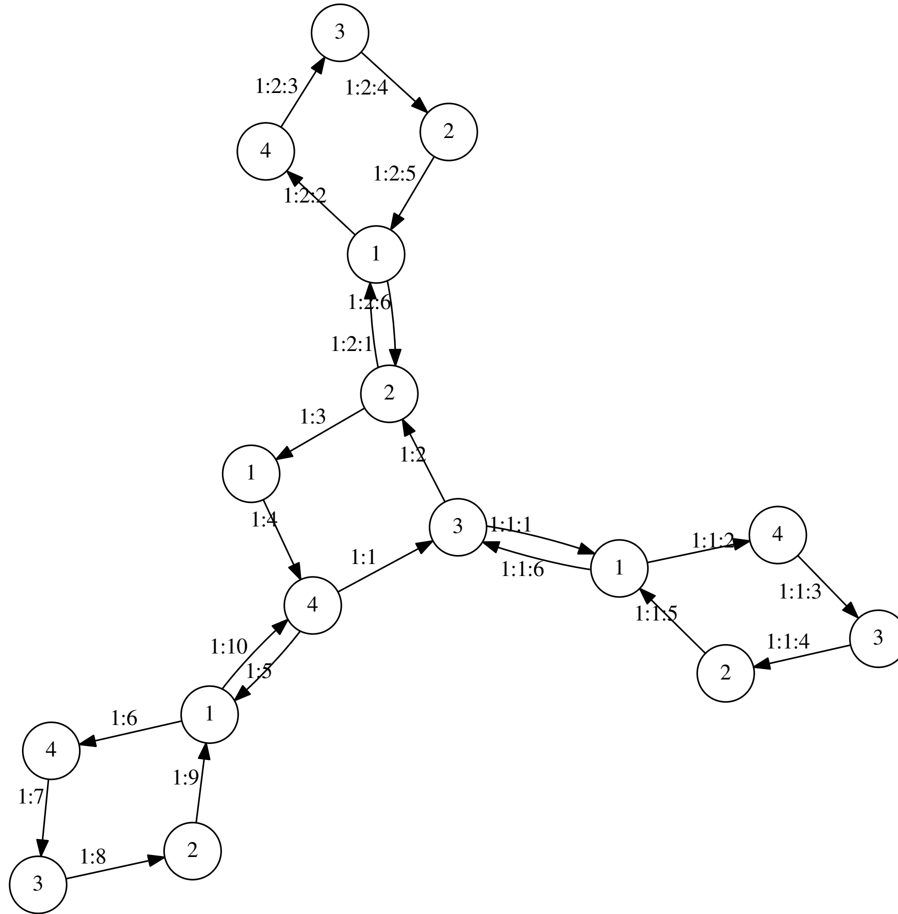


Figure 5.16.: The euler cycle computed by `euler` on the star cycle $SC_{4,4}$.

space consumption between the two implementations.

We executed `euler` on the star graphs $SC_{k,1000}$ where k ranges from 1 to 10. The experimental results are in Figure 5.3 and Figure 5.17. We do not display a plot of the runtimes because they are almost identical. The jumps in the plot arise from the doubling in size of the graph's node and edge stores as the host graph size increases. The memory management of the two implementations are noticeably different. Examining the maximum heap figures reveals that the space used in the list hashing implementation grows slightly faster than that of the non-hashing implementation. This is because hash table entries store not only the list, but an auxiliary linked-list data structure to support chaining. This becomes a factor in this case because both implementations store approximately the same amount of lists due to unique edge labels and a large variety of node labels. However, the hash table is less wasteful as evidenced by the difference in growth and values between the total heap use of both implementations. This arises because the implementation without list hashing has to frequently create new lists and discard old lists to perform applications of relabelling rules and to

	No Hashing			List Hashing		
$SC_{k,1000}$	Runtime	Max	Total	Runtime	Max	Total
$k = 1$	0.033	0.326	0.499	0.033	1.19	1.37
$k = 2$	0.106	0.633	0.978	0.107	1.561	1.746
$k = 3$	0.226	1.414	1.933	0.231	2.379	2.484
$k = 4$	0.393	1.652	2.454	0.394	2.683	2.789
$k = 5$	0.605	2.417	3.516	0.61	3.521	3.634
$k = 6$	0.869	2.657	4.037	0.875	3.816	3.938
$k = 7$	1.172	2.897	4.558	1.187	4.12	4.242
$k = 8$	1.522	3.137	5.079	1.542	4.424	4.567
$k = 9$	1.969	4.426	6.682	1.944	5.777	5.932
$k = 10$	2.375	4.666	7.203	2.396	6.081	6.236

Table 5.3.: Experimental results of `euler` using two list storing implementations. Runtime is given in seconds. Maximum and total heap use is given in megabytes

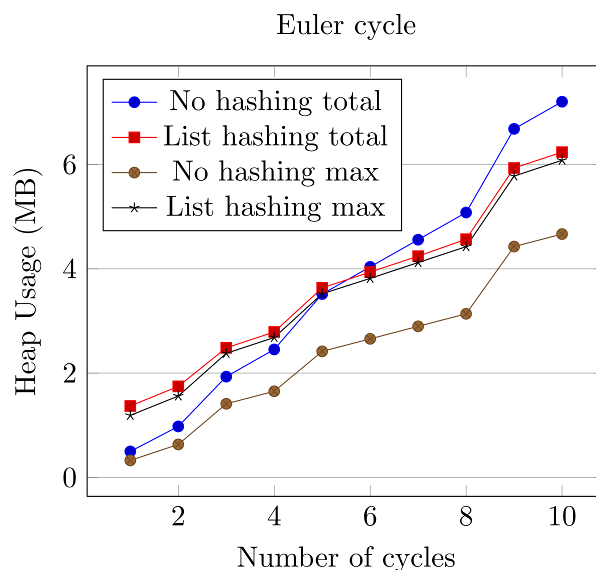


Figure 5.17.: Plot of `euler`'s memory use

support graph backtracking [2], discussed in detail in the next section. Overall, we conclude that for large host graphs with very few repeated lists, not using a hash table is more efficient as it occupies less heap at any one time.

5.7.3. Analysis

In the Sierpinski triangle generation program, where a small set of labels is spread over a large collection of nodes and edges, the hash table proves its worth, outperforming the alternative in both time and space efficiency. In the Euler cycle program, which features a mixture of heavy list sharing and storage of distinct labels in the order of the size of the host graph, the non-hashing implementation

[2] In some cases, old labels are kept in memory in case a previous version of the host graph needs to be restored

is more memory efficient, but not to the same extent as in the Sierpinski benchmark. It is clear from the results that the overhead of maintaining the hash table has a negligible effect on running time. The main drawback of hash tables in general is a severe reduction in performance when there are frequent collisions. There is currently no evidence to suggest that this is a factor for practical GP 2 programs. Indeed, with the current hash function, it is hard to imagine a GP 2 program that would cause a significant amount of collisions. The only advantage we see when not using a hash table is the constant difference in memory use for small host graphs. However, the size of the hash table is about 800KB, an insignificant amount of memory on modern devices. On the other hand, the difference in memory use for large host graphs is unbounded.

Another benefit of list hashing is that it could be used to implement fine-grained graph querying operations. For example, with the presence of a node store indexed by list hash values, the host graph can be queried for nodes with a specific label, trimming the search space significantly for rules with constant values in their left-hand side. The approach could even be extended to query items whose labels contain variables that have already been instantiated.

For the reasons discussed, we fixed the list storage implementation as the hash table. This is the implementation used for the experiments reported in the remainder of the thesis.

5.8. Host Graph Backtracking

The semantics of GP 2's conditional branches and loops require the host graph to be backtracked to a previous state in certain circumstances. The `if-then-else` statement executes its `then` and `else` branches on the graph reached before executing the condition. The `try-then-else` statement only executes its `else` branch on the graph reached before executing the condition. If a failure occurs in a loop body, the semantic rule states that computation resumes with the graph reached before starting the most recent loop iteration. Subprograms that may require means to facilitate graph backtracking during their execution are called *critical subprograms*, namely conditions in conditional branches and loop bodies. The structures of critical subprograms are analysed to identify where graph backtracking support is needed. The goal of the static analysis is to minimise the runtime overhead of supporting graph backtracking, which is dependent on both the critical subprogram and its context.

Remark 9. In the rest of this chapter we distinguish different kinds of rule. An *empty rule* is a rule with an empty left-hand side. A *predicate rule* is a rule whose left-hand side and right-hand side are equal. Predicate rules are typically used to test the existence of a property of the host graph without changing the graph. We sometimes refer to a non-empty and non-predicate rule as a *standard rule*.

5.8.1. Static Analysis

Before we present the formal static analysis, we show some illustrative examples. In the following, G is the graph state before entering the program fragment, $r_{1,2,3}$ are standard rules, p is a predicate rule, and $e_{1,2}$ are empty rules.

1. `if` r `then P else Q` does not require graph backtracking. If a match exists for r in G, it does not need to be applied. Instead, immediately execute `P` on G.

2. `if` p`;` r `then P else Q` does not require graph backtracking. If p fails, the current graph is G, so no backtracking is required before taking the else branch. If p succeeds, G does not change after application of p, and the remainder of the condition is an instance of example (1).

3. `if` r`;` p `then P else Q` requires graph backtracking. If r succeeds, and p fails, the changes made by r must be undone.

4. `if` p`;` r`! then P else Q` requires graph backtracking. If p succeeds, then r could be applied multiple times, requiring the changes to be undone before executing `P`.

5. `try` p`;` r`! then P else Q` does not require graph backtracking. If p succeeds, the `then` branch is guaranteed to be taken (assuming termination) because a loop never fails. The changes made by $r!$ are kept because of the semantics of `try-then-else`.

6. `(`r`;` e_1`;` e_2`)!` does not require graph backtracking. If r fails to match, the rule is not applied and the loop breaks, retaining the graph state before entering the loop. If r suceeds, the end of the iteration will be reached without failure because e_1 and e_2 are guaranteed to match.

7. `(if` r_1 `then` r_2 `else` r_3`)!` does not require graph backtracking. The effect of the if statement's condition is not "visible" to the loop because the `if-then-else` always executes the branch subprograms on the original graph. Thus the loop body can be seen as a single rule application of either r_2 or r_3 to G.

8. `(try` r_1 `then` r_2 `else` r_3`)!` requires graph backtracking. Using the logic of the previous example, the loop body can be seen as an application of either $r_1; r_2$ or r_3 to G. The first sequence could fail on r_2, requiring the changes made by r_1 to be undone.

The examples raise several points that influence the formalisation and implementation of a static analysis for graph backtracking. First, predicate rules can be ignored when they occur at the start of a critical subprogram. Second, the first command in the critical subprogram of a `try-then-else` statement or a loop body can be ignored if all subsequent commands always succeed. Third, the semantics of the two conditional branching statements can result in different

outcomes with respect to graph backtracking on programs that are similar on the surface.

The formalisations that follow take all these points into account and generalise them. We define *simplification* of commands. Commands simplify to themselves except for the following: a procedure call simplifies to its command sequence, `if C then P else Q` simplifies to `P or Q`, and `try C then P else Q` simplifies to `(C; P) or Q`. We emphasise that this simplification is not intended to be a semantics-preserving transformation: it is a way for the static analyser to break down complex programs and reason about them with respect to graph backtracking.

Commands that cannot fail are defined recursively. The basic commands that cannot fail are `skip`, `break`, a rule call of an empty rule, and a rule set call containing only empty rules. We extend this to compound commands with the following rules:

- A command sequence cannot fail if all of its commands cannot fail.
- A looped program cannot fail.
- `if/try C then P else Q` cannot fail if its simplified command cannot fail.
- `P or Q` cannot fail if both `P` and `Q` cannot fail.

We similarly define commands that do not change the host graph, or *null commands*: `skip`, `break`, and `fail` are null commands. Predicate rule calls and rule set calls containing only predicate rules are null commands. The extension to compound commands is the same as non-failing commands except for loops. A looped command does not terminate if all of its rules match the working graph.

- A command sequence is null if all of its commands are null.
- A looped program is null if the loop body is null.
- `if/try C then P else Q` is null if its simplified command is null.
- `P or Q` is null if both `P` and `Q` are null.

We are now ready to define the programs that require graph backtracking. Let $C = C_1; \ldots; C_n$ be a critical subprogram. Simplify all commands to get $C' = C'_1; \ldots; C'_{n'}$. Remove any null commands from the start of the sequence to get $C'' = C''_1; \ldots; C''_{n''}$. Then, if C'' is the condition of an `if-then-else` statement, backtracking is required if C'' contains a loop or if $n'' > 1$. If C'' is the condition of a `try-then-else` statement or a loop body, graph backtracking is required if either $n'' = 1$ and $C'' = P$ `or` Q where at least one of P and Q requires backtracking, or if any C_i can fail for $2 \leq i \leq n''$.

5.8.2. Implementation

The abstract syntax tree of the program text is inspected by a C function that performs the static analysis described in the previous section. It calls a recursive boolean function f on subtrees representing critical subprograms. If graph backtracking is necessary, the function returns true, which causes a flag to be set in the AST node representing the parent of the critical subprogram (either a conditional branch statement or a loop). This instructs the compiler to generate code for graph backtracking while the critical subprogram is being executed. The function does not explicitly transform command sequences as described above. Instead, it simulates the transformation. For example, calling f on (an internal representation of) `if C then P else Q` returns $f(\texttt{P}) \vee f(\texttt{Q})$.

The implementation supports graph backtracking in two ways. The first is to copy the entire host graph to memory before entering a critical subprogram. The time complexity of this operation is linear in the size of the host graph because the copying function iterates over all nodes and edges to duplicate their labels and the incident edge arrays of nodes. The second is to record the individual changes made to the host graph during execution of the critical subprogram in such a way that they can be reversed to retrieve the correct graph. This is achieved by a *graph change stack*, a concept taken from the YAM [MP08b]. Each stack frame contains the necessary data to undo a single graph modification. A dynamic array is used to implement the graph change stack because the number of stack frames required is not known at compile time.

We investigated the performance of these graph backtracking methods. Cases can be identified where recording graph changes is more efficient. For example, consider the GP 2 program fragment `if (r1; r2) then P else Q`, where `r1` and `r2` are standard rules. The host graph changes made by `r1` have to be reversed before executing `P` or `Q`. It is clear that recording graph changes for at most two rule applications will be significantly more space efficient and, at the very least, not significantly less time efficient than copying the potentially large host graph. Another case in favour of graph recording is nested critical subprograms. For example, in the program `if (if (r1; r2) then (r3; r4) else (r5; r6)) then P else Q`, graph backtracking is required for both if statements. The host graph would be copied twice, effectively doubling the memory overhead, while the graph change stack could span multiple "restoration points", allowing any intermediate host graph to be restored. Thus the graph change stack would cost no more memory regardless of the nesting depth.

Is the converse true? Is there a class of graph programs in which copying the host graph is more efficient than recording graph changes? Consider a critical subprogram containing a loop. The number of loop iterations, and hence the number of graph changes made in the loop, is not known at compile time. This occurs in a common GP 2 program structure called a *reduction test*: `if C! then P else Q`. For such a program, it is unclear whether graph recording would be more efficient than copying the host graph. As the number of graph changes grows larger, more items are pushed to the graph change stack, increasing runtime

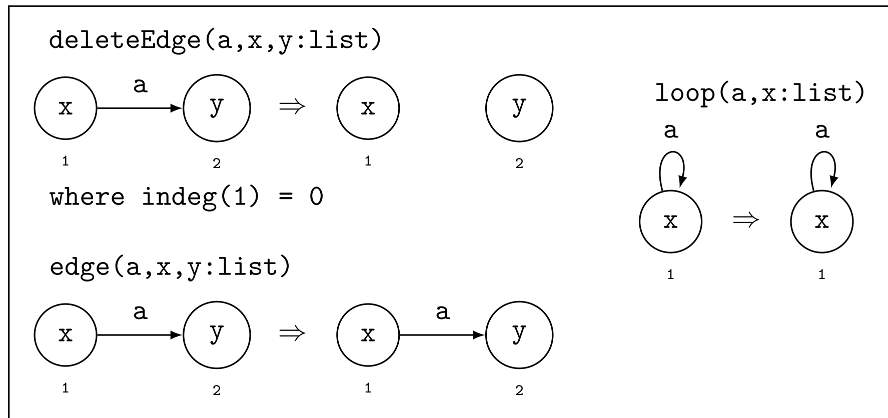


Figure 5.18.: Rules for acyclicity testing

overhead. Furthermore, restoring the old graph state is a very fast reassignment of the host graph pointer. In contrast, the changes on the graph change stack have to be inverted on the host graph, incurring a further overhead penalty proportional to the number of graph changes made in the critical subprogram.

From this discussion arises the following hypothesis: *Consider a graph program that makes k modifications to the graph during execution of the critical subprogram. As k increases, the time taken to execute the program with graph recording is significantly slower than the time taken to execute the same program with a single graph copy.*

If the hypothesis is true, it is worth extending the static analysis on graph programs to establish where it is best to copy the graph and where it is best to record changes. Specifically, loops in a critical subprogram will notify the code generator to generate code that copies the graph before entering that critical subprogram to avoid a potential performance hit from recording and inverting a large number of graph changes. If the hypothesis is false, then graph recording can be used universally, as the number of changes does not have a significant impact on runtime performance compared to graph copying.

Two recognition graph programs are used to test the hypothesis. Both programs test if the host graph belongs to a certain graph class by applying *reduction rules* for as long as possible. A reduction rule matches a certain substructure of the host graph and removes it. The structure of the resulting graph is used to determine whether the host graph belongs to the graph class in question. The result is encoded in an isolated root node with the label "yes" or the label "no". The destructive nature of the reduction test means that graph backtracking is required if the user wishes to preserve the tested host graph.

5.8.3. Case Study: Cycle Checking

Recall that a graph is *acyclic* if it contains no cycles, including loops. The rules of a GP 2 program to test a graph for acyclicity are shown in Figure 5.18 [Plu12].

Using `deleteEdge`, the program removes all edges outgoing from nodes with no incoming edges. The rule preserves acyclicity, and it cannot remove a cyclic

	No Backtracking		Graph Copying		Graph Recording	
Graph Size	Runtime	Max	Runtime	Max	Runtime	Max
10	2	0.038	2	0.055	2	0.043
10^2	2	0.038	2	0.055	2	0.044
10^3	8	0.157	8	0.288	8	0.198
10^4	216	2.184	222	4.282	218	2.84
10^5	24,732	17.323	24,933	34.101	24,577	22.566

Table 5.4.: Experimental results of three versions of the acyclicity program. Graph size is given in number of nodes. Runtime is given in milliseconds. Maximum heap use is given in megabytes

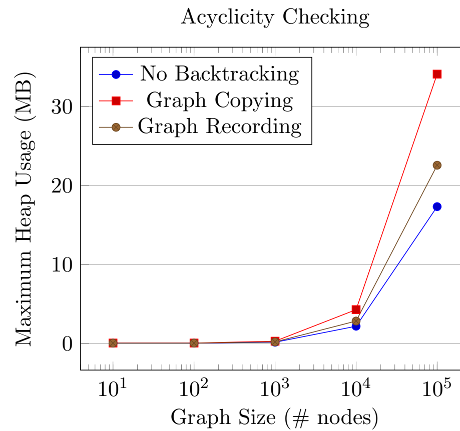


Figure 5.19.: Plot of the memory use of the acyclicity checking programs.

edge because all nodes in a cycle have an indegree of at least 1. Applying this rule for as long as possible on an acyclic graph results in a graph with no edges. Conversely, applying this rule for as long as possible on an cyclic graph leaves the cycles present in the host graph. The two rules `edge` and `loop` are used to test for the presence of an edge. If either rule matches, the host graph contains a cycle. Otherwise the host graph is acyclic.

The two versions of the acyclic testing program are below. Rules `no` and `yes` have the empty graph as their left-hand side and a single root node on the right-hand side labelled `"no"` and `"yes"` respectively.

```
1 Main = deleteEdge!; if {edge, loop} then no else yes
2 Main = if (deleteEdge!; {edge, loop}) then no else yes
```

The first version is used as a base for comparison. No graph backtracking is required because it performs the reduction step destructively, not restoring the host graph. The output is the original graph with all its non-cyclic edges removed, plus a single root node with the appropriate label. The second version performs the reduction step in the condition of an if statement which necessitates graph backtracking. The output is the host graph plus a single root node with the appropriate label.

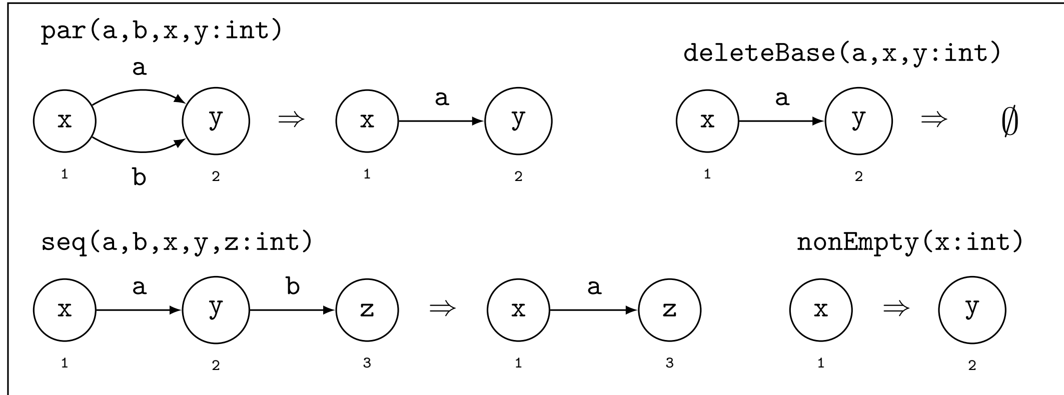


Figure 5.20.: Rules for series-parallel testing

The program was executed on broken cycles (a cycle with a single edge removed) of increasing size. All labels are empty. The results can be seen in Table 5.4 and Figure 5.19. There is little difference between the running times of the three programs. The heap use of graph copying is much greater than that of graph recording for large host graphs, although we note that the reduction rule is very simple: it removes a single edge, which requires only one entry on the graph change stack per loop iteration.

5.8.4. Case Study: Recognition of Series-Parallel Graphs

An important class of graphs, particularly for modelling circuits, are series-parallel graphs [BJG08]. Series-parallel graphs are defined as the class of graphs that reduce to a single edge between two nodes under the following transformations:

1. Replace two parallel edges with a single edge with the same source and target.
2. Replace a directed path of two edges connecting three nodes where the middle node has degree 2, with a single edge connecting the endpoints in the same direction.

We consider directed integer-labelled series-parallel graphs. The definition is directly translated to a GP 2 program using the rules in Figure 5.20 [Plu12].

The program executes the two reduction rules `par` and `seq` on the host graph for as long as possible. If the resulting graph is a single non-looping edge, then the host graph is by definition series-parallel. After the reduction, the test concludes with the removal of a non-looping edge with `deleteBase` and checking that the current graph is empty. The result is encoded, as in the acyclic test, in an isolated root node with the label "yes" or the label "no". Again, there are two versions of the program to control the presence of graph backtracking mechanisms at runtime.

	No Backtracking		Graph Copying		Graph Recording	
Graph Size	Runtime	Max	Runtime	Max	Runtime	Max
10	2	0.037	2	0.043	2	0.056
10^2	2	0.038	2	0.05	2	0.57
10^3	18	0.126	20	0.324	21	0.227
10^4	836	1.699	861	3.312	847	3.302
10^5	94,412	-	97,594	-	96,276	-

Table 5.5.: Experimental results of three versions of the series-parallel checking program. Graph size is given in number of edges. Runtime is given in milliseconds. Heap use is given in megabytes

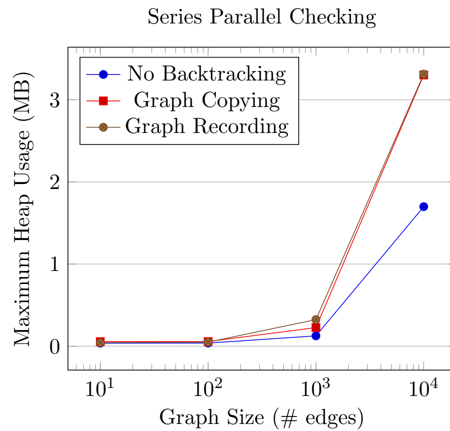


Figure 5.21.: Plot of the memory use of the series-parallel checking programs.

```
1 Main = {par, seq}!; deleteBase; if nonEmpty then no else yes
2 Main = if ({par, seq}!; deleteBase; nonEmpty) then no else yes
```

The host graphs used to test this program are machine-generated[3] series-parallel graphs, weighted so that the number of parallel edges and the number of "sequential edges" are approximately equal. All nodes and edges are labelled 1. On these host graphs, `par` performs one host graph modification: removing an edge, while `seq` performs four: two edge removals, a node removal, and an edge addition. Hence the average number of recorded graph changes per loop iteration is 2.5.

The results can be seen in Table 5.5 and Figure 5.21. Valgrind did not terminate in a reasonable time for the largest host graph. Like the acyclicity testing program, there is little difference in runtime between the three executions. However, the heap usage evens out for the two programs that perform graph backtracking. The greater number of graph changes per loop iteration requires a greater number of changes to be placed on the stack, hence more memory consumption. To take this a step further, we artificially added more graph changes to the reduction step as shown in Figure 5.22:

[3]Using Faulkner's GraphGEN ML program [Fau15]

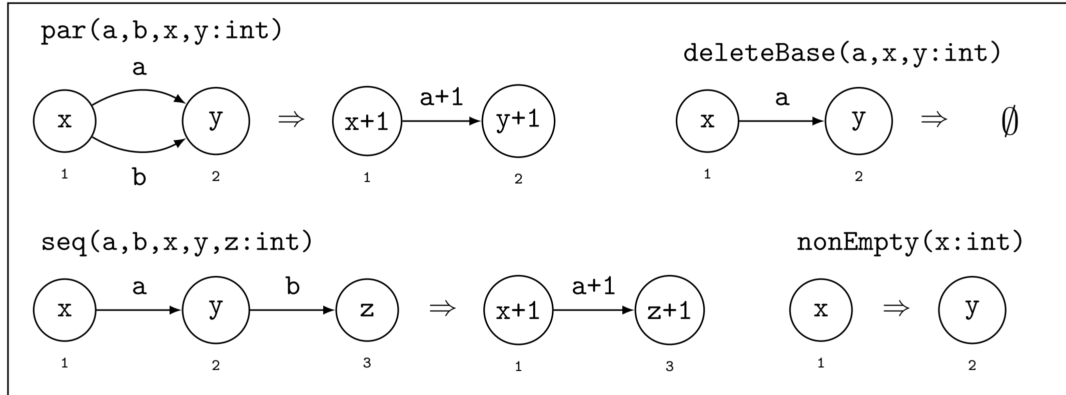


Figure 5.22.: Modified rules for series-parallel testing

In the modified program, `par` performs three relabelling operations in addition to the edge deletion and `seq` performs two relabelling operations in addition to the node deletion, two edge deletions, and edge addition. This averages out to 5 graph modifications per loop iteration. We ran the modified program on the same host graphs as in the previous experiment. The results in Figure 5.6 and Figure 5.23 show that the memory used by graph recording grows more rapidly than the memory used by graph copying.

5.8.5. Analysis

According to our experiments, the overhead introduced by both forms of graph backtracking has a negligible impact on runtime performance. This falsifies the hypothesis: an increase in the number of recorded graph changes does not decrease runtime performance. Therefore, for the purpose of optimising the speed of executing graph programs, there is no need to pick and choose one graph recording method over the other depending on the context of the program. However, for space efficiency, the results show that the difference in memory usage is dependent on the program structure and the complexity of the rules. Specifically, for reduction programs in which a large amount of computation needs to be recorded and reversed, either implicitly (by copying the host graph) or explicitly (by storing a representation of the computation), the complexity of the reduction step has a large impact on the relative memory use.

The optimal solution would be to extend the current static analysis to heuristically select the graph backtracking mechanism that is estimated to use the least memory with respect to the program structure. This would be a complex extension to the codebase: one would need to design, implement and test a heuristic function based at a minimum on the counts of the changes made by each rule in a critical subprogram. We chose not to take this route since the development time and effort was not worth the small and perhaps insignificant reduction in memory use for a subset of graph program-host graph pairs. Instead, we chose to use graph recording as the sole mechanism for graph backtracking at runtime. The program patterns that strongly favour graph recording, such as those performing graph backtracking for a small (and constant) number of graph changes

	No Backtracking		Graph Copying		Graph Recording	
Graph Size	Runtime	Max	Runtime	Max	Runtime	Max
10	2	0.037	2	0.056	2	0.043
10^2	2	0.038	2	0.57	2	0.63
10^3	20	0.126	19	0.227	20	0.521
10^4	859	1.699	862	3.302	853	4.885
10^5	98,789	-	96,230	-	97,608	-

Table 5.6.: Experimental results of three versions of the modified series-parallel program. Graph size is given in number of nodes. Runtime is given in milliseconds. Heap use is given in megabytes.

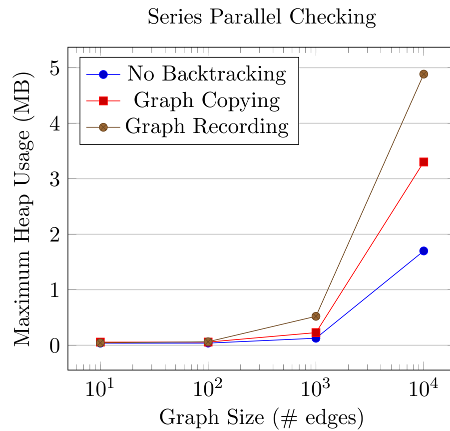


Figure 5.23.: Plot of the memory use of the modified series-parallel checking programs.

and those with nested critical subprograms, are more frequent than critical subprograms that perform a large number of graph changes with respect to the size of the host graph.

5.9. Code Generation

The structure of the GP 2 compiler is broken down in Figure 5.24. As mentioned in the previous section, a Bison-generated parser is used to parse the text files. This is a standard, well-established way to parse a context free grammar, so we only describe this phase in brief. The host graph file is parsed to check for syntax errors and to generate a count of nodes and edges to be later used as the arguments of the function responsible for initially allocating memory to the graph data structure. The program parser syntax checks the input program file and builds an abstract syntax tree (AST) of the program. The abstract syntax tree is built with Bison *action code*, arbitrary C code that is attached to Bison grammar productions to be executed when the production is reduced during parsing. Each piece of action code calls a function to build an AST node representing the piece of program structure that has been parsed. Bison generates a bottom-up parser, so child nodes are generated before their parents. The functions to build parent

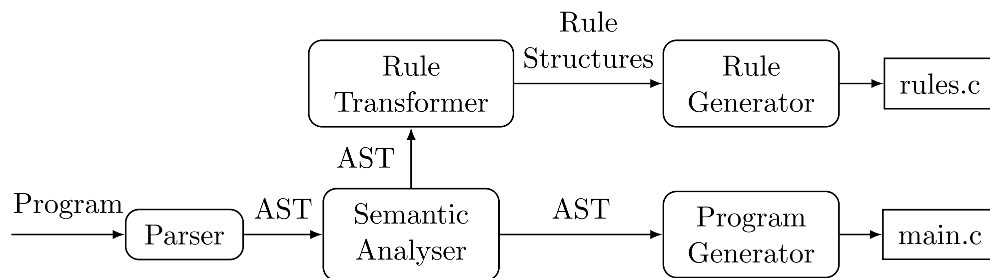


Figure 5.24.: GP 2 compiler architecture

nodes take as arguments pointers to any child nodes constructed during a previous parser reduction step and any necessary semantic information. Bison's syntax makes this form of tree generation quite straightforward and intuitive.

The semantic analyser is a collection of C functions that walk the program AST and check for any semantic errors, which are described in Section A.6. This phase also makes some modifications to the AST. It assigns each rule name a unique identifier: a concatenation of the rule's scope, which is either *Main* or the name of its declaring procedure, and the rule's name. Procedure names are globally unique, which guarantees uniqueness of rule identifiers even if rule names are shared across multiple procedure declarations (see Section 3.4 for a description of local rules and GP 2's scoping system). It points each rule call and procedure call in the program text to the AST node representing the declaration of the corresponding rule and procedure respectively. This eases the code generation process.

The focus of this section is on the generation of code to match and apply rule schemata. Rule application is the unit of computation in GP 2, and the theoretical bottleneck in performance, so it is important to generate concise and efficient C code. The code generator for rule schemata is the most complex part of the compiler. We describe its three main components using the artificial rule in Figure 5.25 as a running example: the matching code generation, the condition code generation and the rule application code generation. Combined, the output is a C module that exports two functions for standard rules: one to match the rule, and one to apply the rule. No matching function is generated for empty rules, and no application function is generated for predicate rules. The final part of code generation is combining these functions according to the control sequence of the GP 2 program. Therefore, to conclude the section, we present the code generation for the GP 2 control constructs with the intention of demonstrating (but not proving) that the compiler respects GP 2's semantics.

The first step is the transformation phase. The AST of each rule declaration is transformed into a complex rule data structure that captures all the information necessary for generation of correct and concise rule application code. The node and edge structures in particular contain far more information than those in the host graph. Nodes contain several flags to inform the code generator if it is a root node, how the rule changes the node (e.g. deletion, relabelling), and whether its indegree or outdegree is queried for a right-hand side label. They also contain

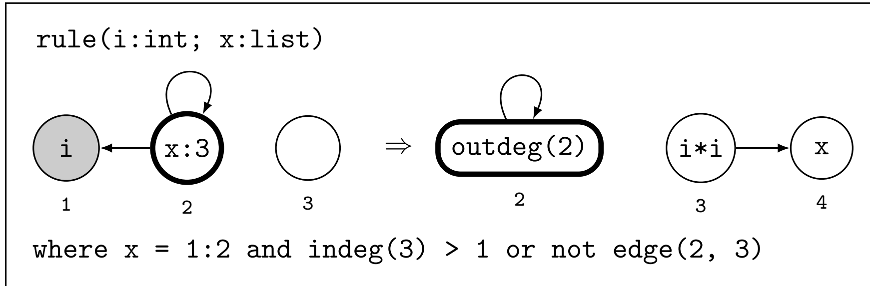


Figure 5.25.: The GP 2 rule schema `rule`.

pointers to their incident edges and any predicates they participate in. An edge structure contains similar flags and pointers to its source and target nodes. Nodes and edges also store an *interface pointer* whose purpose is to connect two interface nodes or edges. For example, the interface pointer of a left-hand side node points to its counterpart in the right-hand side, or *NULL* if the node is not in the interface. The implementation is faithful to the convention that interfaces do not contain edges. Preserved edges are inferred from the interface nodes: they are precisely the edges that connect the same interface node(s) in the same direction on both sides of the rule. In this example, there is one preserved edge, the loop on node 2. Variables point to any predicates they participate in and store a flag to indicate if they are used by the rule (specifically, if they exist in any right-hand side label).

The morphism data structure, as described in Section 5.6.3, contains three arrays. One integer array represents the mapping between rule nodes and host nodes. Another represents the mapping between rule edges and host edges. The third array, an array of assignments, represents the mapping between variables and values. Rules are known at compile time, so the runtime system has a complete knowledge of the runtime data structure for morphisms, in particular the size of the three arrays. The morphism is initialised with dummy values: -1 for the node and edge arrays, and ('n', *NULL*) for the assignments. Each node, edge and variable is assigned a unique identifier which acts as the index into the appropriate array component of the morphism. The identifier assignment for rule is $\{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 2\}$ for nodes, $\{2 \to 1 \mapsto 0, 2 \to 2 \mapsto 1\}$ for edges, and $\{i \mapsto 0, x \mapsto 1\}$ for variables. From now on we refer to rule nodes and edges as 'n' and 'e' respectively followed by their identifier as above in order to synchronise the prose with the upcoming code fragments. For example, rule node 3 is called $n2$.

5.9.1. Searchplans

Finding a match for a rule is the bottleneck in any implementation of graph transformation. The algorithm used to implement matching is, therefore, crucial to the performance of the entire implementation. A common algorithm for graph matching is the *searchplan* (see, for example, [Dör95]). A searchplan matching algorithm decomposes the matching of a rule's left-hand side into a sequence

of primitive matching operations and executes those operations in order to find a complete match within the host graph. We have already seen an abstract searchplan matching algorithm in Section 4.3.

What are the benefits of a searchplan matching algorithm? It decomposes a complex structural problem (subgraph isomorphism) into a number of simple "atomic" steps, which allows a modular implementation. For instance, one could implement a searchplan by writing a function for each step and composing the functions. Left-hand sides of rules are often connected and the node degrees in host graphs are bounded. In these cases, the complexity of matching a rule with a searchplan is bound by the complexity of the first matching operation. Another advantage of this approach is that it can easily be optimised and tailored to a specific application area. Indeed, there has been a lot of work put into dynamic searchplans, where properties of the host graph, or expected classes of host graphs, are used to generate the optimal sequence of matching operations according to a cost function [Zün96; HVV07; BKG07].

We inherit the static searchplan generation algorithm of GP 1 [MP08a]. The searchplan is fixed at compile time, and only takes into account the structure of the rule. A rule can be matched by many searchplans. Some will outperform others depending on the metrics of the host graph. A static searchplan generation algorithm is blind to the host graph, and therefore is not guaranteed to generate the most efficient searchplan. On the other hand, the complexity and overhead of the runtime system is reduced. In GP 1, in order to minimise branching and backtracking, primitive search operations are ordered by their determinism as follows:

1. Check predicates of the schema condition whose variables have been instantiated.

2. Find source and target nodes of matched edges.

3. Find an edge whose source and target nodes have both been matched.

4. Find an edge whose source or target node has been matched.

5. For conditions of the form `not edge(v,w)` where either v or w has been matched, find the other node.

6. Find a node.

GP 2's searchplan generation adopts this order but with some key differences. As described later, conditions are checked as soon as possible, but we do not consider condition evaluation to be an atomic searchplan operation. Furthermore, the GP 2 compiler does not perform operation (5). Considering a specific type of condition adds unnecessary complexity to the compiler and to the generated code. Furthermore it is not clear that this would improve matching efficiency in the current model. In fact, it increases the likelihood of conducting two expensive node matching operations which may be avoidable.

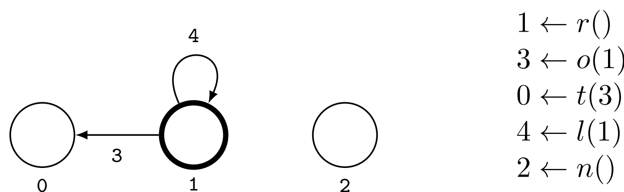


Figure 5.26.: The structure of `rule` and its associated searchplan.

Searchplans are constructed through an undirected depth-first search on left-hand sides of rules. When a node or edge is visited, the appropriate matching operation is appended to the searchplan. Each search starts at a root node if possible, otherwise an unvisited unrooted node is chosen. An example left-hand side and searchplan can be seen in Figure 5.26. The matching operations are represented by a single letter; their full specification is as follows:

- Match an unrooted node.
- Match a root node.
- Given a matched edge, match its target.
- Given a matched edge, match its source.
- Given a bidirectional edge, match one of its incident nodes.
- Match an edge.
- Given a matched node, match one of its outgoing edges.
- Given a matched node, match one of its incoming edges.
- Given a matched node, match one of its looping edges.

The searchplan is represented internally by a linked list of operation structures. A sequence of C functions, subsequently called *matchers*, is generated from each operation, each responsible for finding a match for a single rule item. The structure of the resulting code is a nested chain of matchers as shown in the pseudocode of Figure 5.27. The main rule matching function calls the first matcher, which in this case matches the root node `n1`. If a matcher succeeds in finding a compatible host graph item, it updates the morphism and calls the next matcher in the chain or returns true if it is the last matcher. If none of the candidate host items match, the matcher returns false, returning control to the previous matcher or to the main matching function.

Most of the work is done in testing whether the host item in question is a valid match for the rule item represented by a matcher. The most expensive of these tests is label matching. There are three simple tests that can rule out invalid host items before their labels needs to be considered. A host item h cannot take part in a match if:

```
bool match_rule {
  return match_n1;
}

bool match_n1 {
  for(root nodes N of the host graph) {
    if(N is not a valid match for n1) continue;
    else {
      flag N as matched;
      update morphism;
      if(match_e0) return true;
    }
  }
  return false;
}

bool match_e0 {
  for(outedges E of match(n1)) {
    if(E is not a valid match for e0) continue;
    else {
      flag E as matched;
      update morphism;
      if(match_n0) return true;
    }
  }
  return false;
}

...

bool match_n2 {
  for(nodes N of the host graph) {
    if(N is not a valid match for n2) continue;
    else {
      flag N as matched;
      update morphism;
      return true;
    }
  }
  return false;
}
```

Figure 5.27.: Skeleton of the rule matching code.

1. h is flagged as matched. Mapping distinct rule items to the same host graph item violates injective matching.

2. The rule item is not marked `any` and h's mark is not equal to the rule item's mark.

3. h is not structurally compatible with respect to the rule and the current partial morphism.

The third point requires elaboration, and differs depending on the searchplan operation and the rule's behaviour. Consider node `n0`. It has one incoming edge and no outgoing edges. Clearly the mapping $n0 \to v$ for some host node v cannot extend to a total morphism if v has no incoming edges. Therefore, when finding a match for a rule node n, there is condition on host nodes v: $\text{indeg}(v) \geq \text{indeg}(n) \wedge \text{outdeg}(v) \geq \text{outdeg}(n)$. If v is deleted by the rule, the condition is stricter because of the dangling condition. Again consider $n0 \to v$. Any outgoing edge e of v is not matched because $n0$ has no outgoing edges in the rule. Hence e will be a dangling edge after rule application. Similarly, if v has more than one incoming edge, at least one of these edges will be left dangling by the rule. Therefore the degree condition is: $\text{indeg}(v) = \text{indeg}(n) \wedge \text{outdeg}(v) = \text{outdeg}(n)$. Evaluating this condition is fast: the rule structure is known at compile time, so the generated code compares the degrees of host nodes against constants.

For loops in the rule, the structural check on the host edge is a simple test of whether it has the same source and target node. For non-looping edges, the source and target consistency needs to be checked. For example, edge `e0` is matched by an outgoing edge operation on the node `n1`. Host edges are retrieved from the outedge list of `n1`'s image. The structural check tests if the target of the host edge is matched. If so, the morphism is queried to verify that the target of the host edge is equal to image of `e0`'s target `n0`. If `n0` has been assigned a different host graph node, the structural morphism condition is violated, so the host edge in question cannot extend the partial morphism to a total morphism.

The label matching algorithm is essentially an implementation of the Update Assignment and Check procedures described in Section 4.4. To reiterate: the atoms either side of the list variable are matched first and the list variable is assigned the remaining sublist of the host list. In this way matching is a constant time operation, given that at most one list variable may occur in a left-hand side label. Two examples of the C code generated to match labels are in Figure 5.28. The local variable `match` stores the result of the list matching.

The empty list can be matched in one line of C code. Any other rule list generates much more verbose code using several local variables: `label` stores the label of the host item, `new_assignments` stores the number of assignments made within a single matcher, and `result` stores the result of a single variable-value assignment attempt. The list matching code is wrapped in a do-while loop so that matching can be aborted with a break statement as soon as an inconsistency between the rule label and the host label is detected. The code generator writes a preliminary length check based on the length of the rule list (line 6), then it

```
1  match = label.length == 0 ? true : false;
2
3  bool match = false;
4  int new_assignments = 0;
5  do {
6    if(label.length != 1) break;
7    HostListItem *item = label.list->first;
8    int result = -1;
9    if(item->atom.type != 'i') break;
10   result = addIntegerAssignment(morphism, 0, item->atom.num);
11   if(result >= 0) new_assignments += result;
12   else break;
13   match = true;
14 } while(false);
```

Figure 5.28.: Generated code for label matching. Line 1 is the code to match an empty list. The code in lines 3-14 matches the integer variable `i`

iterates through the rule list, at each step calling a function to generate code to match an individual atom. The function call to match the integer variable `i` writes the code in lines 8-12. After this, if control is still within the loop, then a break statement has not been called, which means that the match succeeded. Before exiting the do statement, the compiler writes a line to set `match` to true. The value of match is queried by the subsequent code as outlined in Figure 5.27.

We conclude this section by noting that behaviour of this algorithm strongly resembles the rooted rule matching algorithm presented in Section 4.4 as it incrementally builds a morphism with the addition of node-to-node, edge-to-edge and variable-to-value mappings to an internal data structure. The main difference between the two algorithms is that the algorithm generated by the GP 2 compiler searches for one match only in a depth-first way, while the other algorithm searches for all matches in parallel. Therefore the searchplan-based matching algorithm here can be seen as a practical realisation of the rooted rule matching algorithm.

5.9.2. Conditions

Generating code to evaluate conditions is not straightforward. The main reason is that conditions query variables, nodes and edges depending on their assigned host graph values. A naive and inefficient approach is to evaluate the condition at the end of rule matching once the total morphism is found. This simplifies code generation since the condition is only evaluated in one place, but it could result in large amounts of unnecessary match backtracking at runtime because it is often the case that the condition can be shown to be false by a single assignment to a condition variable. Therefore, the compiler generates code to evaluate the condition the moment a participatory item is assigned a value. In this way, failure is detected as soon as possible, and the backtracking is simplified: we only need to backtrack one step, as we do for any other form of matching failure. Generating code to perform "on-demand" condition evaluation within each

```
1  /* Global variables */
2  bool b0 = true, b1 = true, b2 = false;
3
4  static bool evaluateCondition(void) {
5     return ((b0 && b1) || !b2);
6  }
7
8  static void evaluatePredicate1(Morphism *morphism) {
9     int n2 = lookupNode(morphism, 2);
10    /* If the node is not yet matched by the morphism, return. */
11    if(n2 == -1) return;
12
13    if(getIndegree(host, n2) > 1) b1 = true;
14    else b1 = false;
15 }
16
17 static void evaluatePredicate2(Morphism *morphism) {
18    int n1 = lookupNode(morphism, 1);
19    if(n1 == -1) return;
20
21    int n2 = lookupNode(morphism, 2);
22    if(n2 == -1) return;
23
24    Node *source = getNode(host, n1);
25    bool edge_found = false;
26    int counter;
27    for(counter = 0; counter < source->out_edges.size + 2; counter++) {
28       Edge *edge = getNthOutEdge(host, source, counter);
29       if(edge != NULL && edge->target == n2) {
30          b2 = true;
31          edge_found = true;
32          break;
33       }
34    }
35    if(!edge_found) b2 = false;
36 }
```

Figure 5.29.: Generated C code for a condition

searchplan matching function, at both the level of label matching (for variables) and structural matching (for nodes and edges) is one of the most complex parts of the code generator.

We walk through the code generation for the condition of our running example, namely `where x = 1:2 and indeg(3) > 1 or not edge(2, 3)`. Some of the C code generated to evaluate this condition is shown in Figure 5.29. The function `evaluatePredicate1` evaluates `edge(2, 3)` and `evaluatePredicate2` evaluates `indeg(3) > 1`. First, the predicates are assigned identifiers. Explicitly, `x = 1:2` is predicate 0, `indeg(3) > 1` is predicate 1, and `not edge(2, 3)` is predicate 2. A global boolean variable is associated with each predicate at runtime. Boolean operators give the condition a tree structure, one that is generated by the parser. Three passes are made over the AST subtree representing the condition. The first pass generates the declarations and initialisations of the boolean variables (line 2). Variables representing negated predicates are initialised to false, and all others are set to true. This guarantees that the condition evaluator returns true under the default values, which is necessary when the condition is evaluated before all of its containing variables are instantiated. The second pass generates the function that evaluates the condition (lines 4–6). It performs a simple transformation from the tree structure to the boolean expression it represents. The third pass is responsible for generating the functions to evaluate each predicate (lines 8–36). These functions first check if all the relevant items in the predicate are instantiated. If so, the predicate is evaluated and its boolean variable is set. Otherwise, the function exits, and the boolean variables keep their default values.

As aforementioned, the condition is evaluated whenever one of its variables (a rule node or a rule variable) is assigned a value. Therefore, in the matching function for `n2`, *evaluatePredicate1* and *evaluatePredicate2* are called when a matching host node is located. Any calls to functions that evaluate predicates are immediately followed by a call to *evaluateCondition*. If *evaluateCondition* returns false, the matcher resets the boolean variables to their initial values and examines the next candidate host node (or returns false if none remain). This is illustrated in the pseudocode given in Figure 5.30. Variables are treated in the same way: predicates containing variables are evaluated immediately after any of its variables is assigned a value.

5.9.3. Rule Application

The code generator scans the right-hand side of a rule to generate the function to apply the rule. First, local variables are declared to store the values from the morphism needed for right-hand side labels, namely values of variables and the degrees of host nodes. The right-hand side may not contain all variables in the rule. As mentioned earlier, nodes and variables in the rule data structure are flagged if they participate in a right-hand side label. This informs the code generator to print code to extract the appropriate values from the morphism or from the host graph. This minimises the morphism and host graph querying at runtime. The values of integer variables, the values of string variables and

```
bool match_node {
  for(nodes N of the host graph) {
    if(N is not a valid match for node) continue;
    else {
      flag N as matched;
      call predicate evaluators;
      if(condition evaluates to true) return true;
      else reset boolean variables;
    }
  }
  return false;
}
```

Figure 5.30.: Generated code for matching a node that participates in the condition

the degrees of host nodes are stored in variables of the appropriate type. The value of a list variable l is not immediately extracted from the morphism because the type information from the assignment is necessary to create right-hand side labels involving l. In our running example, variables i, x, and the outdegree of node 2 are used in right-hand side labels, resulting in the following local variable initialisation code at the start of the rule application function.

```
void applyMain_rule(Morphism *morphism, bool record_changes) {
  int var_0 = getIntegerValue(morphism, 0);
  Assignment var_1 = getAssignment(morphism, 1);
  int node_index = lookupNode(morphism, 1);
  int outdegree1 = getOutdegree(host, node_index);
```

In addition to the morphism, the rule application function takes a boolean argument called *record_changes*. If set to true, the code will push a frame to the graph change stack when a change is made to the host graph. We omit that code for conciseness. Host graph modifications are performed in the following order to prevent conflicts and dangling edges: delete edges, relabel edges, delete nodes, relabel nodes, add nodes, add edges. It follows that the images of $e0$ and $n0$ are removed from the host graph first:

```
int host_edge_index = lookupEdge(morphism, 0);
if(record_changes) { ... }
removeEdge(host, host_edge_index);

int host_node_index = lookupNode(morphism, 0);
if(record_changes) { ... }
removeNode(host, host_node_index);
```

The next stage of rule application is to relabel `n1` and `n2`. The code to relabel `n1` is below.

```
host_node_index = lookupNode(morphism, 1);
HostLabel label_n1 = getNodeLabel(host, host_node_index);
HostLabel label;
int list_var_length0 = 0;
```

```
int list_length0 = list_var_length0 + 1;
HostAtom array0[list_length0];
int index0 = 0;
array0[index0].type = 'i';
array0[index0++].num = outdegree1;
HostList *list0 = makeHostList(array0, list_length0, false);
label = makeHostLabel(0, list_length0, list0);
if(equalHostLabels(label_n1, label)) removeHostList(label.list);
else {
  if(record_changes) { ... }
  relabelNode(host, host_node_index, label);
}
```

An array of `HostAtoms` called *array0*[4] is created to store the evaluated right-hand side label. The size of the array is the sum of the number of non-list variable atoms in the right-hand side label and the values of all list variables in the right-hand side label. The first value is worked out at compile time, while the second requires a runtime computation if any list variables exist in the label. In this case, the right-hand side label is `outdeg(2)`, a list of length 1 containing no list variables. The first element of the array is assigned the type '*i*' and the value of the local variable *outdegree*1. Before relabelling is performed, the function *makeHostList* adds the new label to the hash table and returns a pointer to the list. This pointer is passed to *makeHostLabel* with the mark from the rule label. A mark is represented as an enumerator. The code generator prints the integer value of the enumerator; here 0 represents the absence of a mark. Finally, the new label is compared with the current label of $n1$. If they are different, then the change is pushed to the graph change stack if necessary and the node is relabelled. Otherwise, *removeHostList* is called to decrement the reference count of the list in the hash table. The code to relabel the other node is similar, except the array value is set to *var_0 * var_0*, where *var_0* was earlier initialised to the value of variable *i*.

Finally, the code to add the new node and edge is below.

Populating the array from a list variable is more cumbersome than from an integer or a string variable because the way that the array is populated depends on the type of the assignment. Another complication is that adding an edge to the host graph requires knowledge of its source and target. Nodes incident to added edges are either interface nodes or nodes created by the rule. This information is available from the interface pointers in the structures for rule nodes. If the node is in the interface, the node identifier is found through the morphism. In the second case, the node identifier is found in the *rhs_node_map* array created before nodes are added to the host graph. In the running example, the new edge is incident to $n2$, an interface node, and the new node. The code queries the morphism to get $n2$'s image, and it queries *rhs_node_map* for the added node. The morphism is reset through a function call before function exit so that future matches of the same rule are not corrupted by the existing morphism values.

[4] The integer suffix is used to prevent variable name clashes.

```
/* Array of host node indices indexed by RHS node. */
int rhs_node_map[3];
int list_var_length2 = 0;
list_var_length2 += getAssignmentLength(var_1);
int list_length2 = list_var_length2 + 0;
HostAtom array2[list_length2];
int index2 = 0;
if(var_1.type == 'l' && var_1.list != NULL) {
  HostListItem *item2 = var_1.list->first;
  while(item2 != NULL) {
    array2[index2++] = item2->atom;
    item2 = item2->next;
  }
}
else if(var_1.type == 'i') {
  array2[index2].type = 'i';
  array2[index2++].num = var_1.num;
}
else if(var_1.type == 's') { ... }
HostList *list2 = makeHostList(array2, list_length2, false);
label = makeHostLabel(0, list_length2, list2);
host_node_index = addNode(host, 0, label);
rhs_node_map[0] = host_node_index;
if(record_changes) { ... }

int source, target;
source = rhs_node_map[0];
target = lookupNode(morphism, 2);
host_edge_index = addEdge(host, blank_label, source, target);
if(record_changes) { ... }
```

5.9.4. The Main Routine

The main function of the generated C program is responsible for calling the matching and application functions as designated by the command sequence of the GP 2 program. The program is a set of rule applications organised with an imperative syntax. The code generator writes a short code fragment for each rule call and translates each GP 2 control construct into the equivalent C control construct. In order to preserve the meaning of the program, the code must provide graph backtracking when appropriate. In the presented code, we assume that graph backtracking is implemented via a graph change stack as discussed in section 5.8.

The runtime code is supported by a number of global variables, including the host graphs and morphisms. The main function first calls a function to populate the host graph's data structure via the host graph parser, then it allocates memory for each morphism. Morphisms are passed to the matching and application functions, and they are reset to contain dummy values after the rule is applied or after the rule fails to match. A global boolean variable *success*, initialised to true, is used to store the outcome of a computation to support the control flow of the program.

<table>
<tr><th>Rule Type</th><th>Generated Code</th></tr>
<tr><td>Empty</td><td><pre>apply_R(b);
success = true;</pre></td></tr>
<tr><td>Predicate</td><td><pre>if(matchR(m)) success = true;
else <failure code></pre></td></tr>
<tr><td>Standard</td><td><pre>if(matchR(m)) {
  applyR(m, b);
  success = true;
}
else <failure code></pre></td></tr>
</table>

Figure 5.31.: Generated C code for rule calls

The code generator carries some information about the command it is currently processing to support generation of correct code. Some aspects of the generated code, such as failure handling, are dependent on the context of the command. For instance, failure in a loop body is treated differently from failure at the top level. Information about graph backtracking is also present, obtained from the static analysis described in subsection 5.8.1. Each rule application function takes a boolean argument that pushes changes onto the graph change stack if set to true.

Code generated from rule calls, the basic unit of a GP 2 program, is dependent on the rule's structure. Figure 5.31 shows the code generated for each rule type. The letters m and b are used to represent the morphism and boolean arguments to rule-related functions. Empty rules do not have a matching function and their application function does not take a morphism argument. Predicate rules do not have an application function. The failure code is shown at the end of the section.

Figure 5.32 summarises the translation of each GP 2 control construct to C. Command sequences and procedure calls are not shown. Command sequences are easily handled by generating the code for each command in the designated order. When a procedure call is encountered in the program text, the code generator inlines the command sequence of the procedure at the point of the call. The condition of a branching statement is generated in C's `do-while` loop: if the command sequence fails before the last command, C's break statement is called to exit the condition, where control is assumed by the then/else branch. GP 2's loop translates directly to a C while loop. One subtlety is the looped command sequence, where the line `if(!success) break;` is printed after the code for all commands except the last, the same mechanism as used in rule set calls. A second subtlety is that success is set to true after a loop exits because GP 2's semantics states that a loop cannot fail.

The non-deterministic constructs are handled in different ways. The rule set call `{R1, R2}` is tackled by applying the rules in textual order until one rule matches or they all fail. C's `do-while` loop is used to exit the sequence after a rule application: it would be incorrect to try and match `R2` if `R1` succeeds. In contrast,

Command	Generated Code

{R1, R2}	do {
	  if(matchR1(M_R1)) {
	    <success code>
	    break;
	  }
	  if(matchR2(M_R2)) <success code>
	  else <failure code>
	} while(false)

if C then P
else Q
	int restore_point = <current GCS frame>;
	do C while(false);
	undoChanges(host, restore_point);
	if(success) P else Q;

try C then P
else Q
	int restore_point = <top of GCS>;
	do C while(false);
	if(success) P
	else {
	  undoChanges(host, restore_point);
	  Q
	}

(P; Q)!	int restore_point = <top of GCS>;
	while(success) {
	  P
	  if(!success) break;
	  Q
	  if(success) discardChanges(restore_point);
	}
	success = true;

P or Q	int random = rand();
	if((random % 2) == 0) <program code for P>
	else <program code for Q>

Figure 5.32.: C code for GP 2 control constructs

C's pseudo-random number generator chooses between the two subprograms of the `or` statement `P or Q`. We chose to implement rule sets in a deterministic way because in our experience, rule sets are used to elegantly model deterministic behaviour. For example, the series-parallel recognition program loops a rule set to reduce a graph to a basic structure, behaviour that is globally deterministic despite the local non-determinism. Another common use case occurs in the acyclic checking program, in which a rule set is used to test two structural properties — the existence of a looping edge or a non-looping edge — in a single command. The rule set calls can be used practically even with the knowledge that the implementation is deterministic. On the other hand, implementing the `or` statement in the same way defeats the purpose of the construct. We acknowledge that users may wish to use rule sets in a genuinely nondeterministic way (one use case could be graph generation), and we aim to implement rule sets by pseudorandom choice

Context	Failure Code
Top Level	`print failure message;` `garbageCollect();` `return 0;`
Condition	`success = false;` `break;`
Loop	`success = false;` `<pop and undo graph changes>`

Figure 5.33.: Generated C code for failure.

in the future. Finally, we note that both approaches are sound with respect to the semantics.

Restore points are created and assigned to the top of the graph change stack before entering a critical subprogram requiring backtracking. The function *undoChanges* restores a previous host graph state by popping and undoing changes from the stack until the restore point is reached. The function *discardChanges* pops the changes but does not undo them. It is only called at the end of a successful loop iteration to prevent a failure in a future loop iteration from causing the host graph to roll back beyond the start of its preceding iteration. Each restore point has a unique identifier in the code to facilitate multiple graph backtracking points.

The failure code is context-sensitive as shown in Figure 5.33. If there is a failure at the top level, the program is terminated after reporting to the user and garbage collecting. The failure message either states the name of the rule that failed to match or that the `fail` statement was invoked. Failure in a condition sets the success flag to false so that the subsequent code takes the else branch of the conditional statement. Failure in a loop sets the success flag to false and calls `undoChanges` to restore the host graph to the state it was in at the start of the most recent loop iteration.

5.10. Comparison with Existing Implementations

We compare the graph transformation tools discussed in section 2.2 with the implementations of GP 2 and GP 1. We categorise the tools on several criteria answering the following questions: (1) What is the underlying theoretical framework for graph transformation rules? (2) Which algorithm is used to match left-hand sides of rules to host graphs? (3) Does the language have a complete formal semantics? (4) How is the graph transformation executed? Is it compiled into native code or interpreted by the tool? (5) Which language is used to implement the interpreter or compiler? Figure 5.34 collates the answers to these questions.

Most tools base their rules on the algebraic approach. The exceptions are PROGRES' Programmed Logic-based Structure Replacement [Sch97] and PORGY's

Tool	Rules	Sem.	Matching	Implementation	Language
PROGRES	PLSR	Y	Searchplan (D)	Int. and comp.	C
AGG	SPO	N	CSP	Interpreted	Java
GROOVE	SPO/DPO	N	Incremental	Interpreted	Java
GrGEN.NET	SPO/DPO	N	Searchplan (D)	Compiled	Java/C#
PORGY	Port Graphs	Y	Ullman [Ull76]	Interpreted	C++
GP 1	DPO	Y	Searchplan (S)	Interpreted	Haskell/C
GP 2	DPO	Y	Searchplan (S)	Compiled	C

Figure 5.34.: Features of graph transformation tools

Port Graphs [AK08], frameworks constructed specifically for their respective tools and application areas. Excluding those, and outside of GP, SPO is universally used as the default approach by those tools, although GROOVE and GrGEN.NET provide users with the option to write DPO rules. The algebraic approach is popular because it is well-founded by decades of mathematical research, and tool designers often wish to incorporate constructs and results from the theory in practice. AGG's critical pair analysis is a notable example [MTR05], and we further note that GP 2 is making steps towards the same goal in the DPO approach [HP15].

PROGRES [Sch91a], PORGY [AK08; FKP14] and GP 1 [Plu09] have a published formal semantics for the entire graph transformation system beyond the behaviour of rule application. The formal semantics for GP 2 are present in this thesis (Sections 3.3 and 3.5); the published semantics of GP 2 [Plu12] does not cover new language features such as the break statement. A semantics is useful for verifying the correctness of an implementation and for formal verification of graph programs. We believe that PROGRES' semantics is too extensive and complicated for this purpose, while the semantics of PORGY and GP should be amenable for these use cases because of the relative simplicity of the languages and the semantics. In GP 2's case, this has been demonstrated with the development of a Hoare-style proof system for the language [Pos13].

The theoretically expensive problem of matching the left-hand side of a rule in a potentially large host graph is addressed in various ways. The most common of these is matching with a searchplan, a composition of small matching operations. Both GP implementations generate searchplans statically, while PROGRES and GrGEN.NET use a searchplan that is dynamically computed using the metrics of the working graph. Roughly speaking, the aim of examining the host graph is to select a searchplan exhibiting the smallest branching factor with respect to the host graph. This concept was introduced by Dörr [Dör95]. Both PROGRES and GrGEN.NET use a cost function to heuristically select the optimal searchplan with a greedy algorithm. The main difference between the two approaches is that GrGEN.NET's searchplan generation algorithm uses data from the current graph [BKG07], while PROGRES makes assumptions about the structure of the host graph based on statistics of typical graphs in their application area [Zün96]. Another dynamic approach is *incremental pattern matching*. The basic idea is to

store all possible matches for the rules at runtime so that they can be accessed quickly when required. This allows very fast matching at the cost of an expensive initialisation phase and the runtime overhead of managing the requisite data structures. The GROOVE engine matches rules in this way using the RETE network technique for graph grammars [BGT91]. A RETE network is a graph representing the incremental construction of all the left-hand sides in the rule set, starting at labelled nodes at the bottom layer and finishing at complete left-hand sides at the top layer. Subgraphs common to multiple left-hand sides are shared in the network. The host graph items are passed through this network at runtime to populate the set of matches which is dynamically updated as the host graph changes. GROOVE extends the original idea with support for its own language features such as quantifiers, NACs and regular path expressions [GJR10; JGR12]. AGG represents subgraph matching as a constraint satisfaction problem (CSP) [Rud98] in order to use the wealth of research into optimised CSP solvers. Finally, PORGY's matching algorithm is based on Ullman's subgraph isomorphism algorithm [PMD12]. The original algorithm is a depth-first search with a look-ahead based search space refinement at each step of the search [Ull76].

The PROGRES compiler generates bytecode which can be executed directly with an interpreter, intended for interactive validation, or used to generate C or Modula-2 source code for rapid prototyping and for the final executable. Both the AGG and GROOVE tools use a Java codebase that interpretively executes graph transformation rules. GrGEN.NET generates executable code (.NET assemblies) from the graph models and rewrite rules to be executed with the support of the system's runtime libraries. The PORGY is tool is implemented in C++. Their underlying hierarchical graph data structure stores a representation of the complete state space along with the graph transformation rules, suggesting an interpretive execution. GP 1 executes graph programs using the York Abstract Machine (YAM), a C program that interprets YAM bytecode generated by a compiler written in Haskell [MP08a].

The GP 2 language has a compiled implementation, which means that less work needs to be done at runtime in comparison to the tools that interpret graph transformation rules at runtime. However, this does not mean a compiled execution is guaranteed or even likely to outperform an interpretive execution. More significant is the matching algorithm. There have been many implementation efforts in optimising subgraph matching with dynamic algorithms [Zün96; BKG07; Hor+10; GJR10], based on the belief that the time gained in examining less of the search space compared to a static algorithm is worth the runtime overhead. However, we are not aware of any direct comparisons between static and dynamic matching algorithms. The closest comparison may be [GJR10], which runs an incremental matching algorithm against searchplan algorithms, although it is unclear how these searchplans are generated. Nevertheless, we do not expect GP 2 to outperform current tools outside of rooted graph programs without a more sophisticated rule matching algorithm.

5.11. Summary and Discussion

In this chapter we have presented two novel implementations of the graph programming language GP 2. The first is a reference interpreter written in Haskell. The codebase consists of about 1,000 lines of Haskell source code, a remarkable feat for a complete implementation of graph transformation even given the typically elegant and concise programming style afforded by Haskell. The performance of the tool is adequate for our purposes, which is a nice outcome given that the tool was programmed with conciseness and simplicity over performance.

The second is a compiler and runtime library that executes high-level graph programs by generating C code from a textual specification of the program and input graph. Some difficult design choices led us to empirically compare distinct implementations of particular GP 2 features in order to make an informed choice as to which approach would be most efficient in general. We documented the code generation process, covering structural rule matching, label matching, condition evaluation, rule application, and control constructs, in order to convince the reader that the compiler makes an effort to be as efficient as possible and that the translation from source code to target code is sound with respect to GP 2's semantics.

There is still much work to be done for the GP 2 implementation to achieve its maximum potential. There are clear areas of improvement for the current compiler, including some "quick wins", but we also identify a dynamic matching algorithm as a more involved optimisation area for the future. However, the case studies presented in this chapter and in the next demonstrate that the generated code is capable of performing demanding computations on large host graphs reasonably efficiently. Chapter 6 combines theory and practice by using the GP 2 compiler to execute rooted graph programs.

6. Case Studies in Rooted Graph Programs

6.1. Introduction

The purpose of this chapter is to put the theory of rooted graph transformation into practice and experimentally determine the efficiency of the implementation described in the previous chapter. We demonstrate that graph programs with fast rule schemata can be used to construct instructive and meaningful solutions to established graph algorithms that perform in the same order of magnitude as tailored implementations in C. In this way, users get the best of both worlds: they can write visual, high-level graph programs with the performance of a relatively low-level language.

6.2. Graph Traversing

Two of the most fundamental graph algorithms, depth-first search (DFS) and breadth-first search (BFS), are based on exploring the entire input graph. Traversing a graph is useful in several ways. First, a graph search can perform a computation on each node upon reaching it. Second, a graph search can reveal interesting and desirable properties about the structure of a graph [THCRS09], a simple example being connectivity. Third, many complicated graph algorithms have graph traversing at their core. Finally, from the point of view of rooted graph programs, graph traversing algorithms provide a way to explore a graph and perform a global computation with the use of fast rule schemata, which in some cases matches the theoretical complexity of standard algorithms in spite of the overhead of applying graph transformation rules.

Graph traversing algorithms are applicable to both directed and undirected graphs. We consider only directed graphs because GP 2 does not support undirected graphs. We note that the programs and results of this section are also applicable to direction-independent traversal: the abstract algorithms are easily adaptable to undirected graphs, while the GP 2 programs can simulate direction-independent traversal by using bidirectional edges in rule schemata.

We give a conceptual description of both types of graph traversal before presenting GP 2 programs that perform these traversals. The terminology, programming patterns and results of this section form the basis for the two case studies introduced later in the chapter.

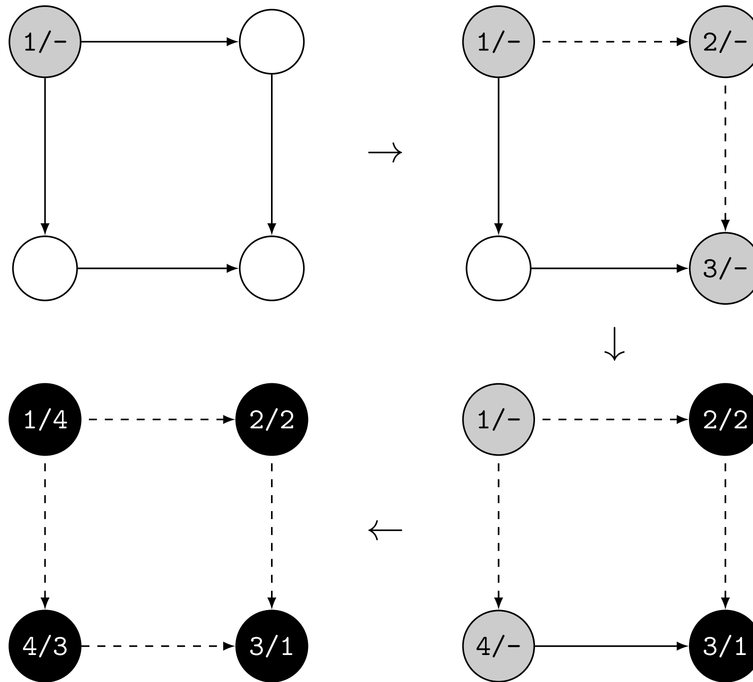


Figure 6.1.: Illustration of a depth-first search

6.2.1. Depth-first Search

A depth first-search of a graph starts by visiting an arbitrary node. Each step of the search visits the target of an unexplored outgoing edge from the most recently-visited node v. If no such edge exists, then v is *finished*, and the outedges of the next most recently-visited node are examined. This process repeats until all outedges of all visited nodes have been explored. Search continues by visiting an arbitrary unvisited node. Search terminates when all nodes in the graph have been visited.

For a graph G, a *preordering* is a list of the nodes in G, where v occurs before w if v is visited before w during a DFS. A *postordering* is a list of nodes in G, where v occurs before w if v is finished before w during a DFS. A *reverse postordering* is the reverse of a postordering, which is in general not equal to the preordering given by the same search. An important property of a reverse postordering is that it is a topological ordering of the nodes of G [THCRS09], a property which we shall use in Section 6.4.

These concepts are illustrated in Figure 6.1, the DFS of a square graph. Colours represent the state of each node: unvisited nodes are white, visited nodes are grey, and finished nodes are black. Each node is labelled with *pre*/*post*, where *pre* and *post* are the positions of the node in the graph's preordering and postordering respectively. Explored edges are dashed. We refer to the nodes by their position in the grid. For example, the top left node is TL which is the first node visited by the algorithm. Then the DFS explores the edges in the order $TL \to TR$, $TR \to BR$, $TL \to BL$, $BL \to BR$. This search order produces the preorder *TL, TR, BR, BL*, the postorder *BR, TR, BL, TL*, and the reverse postorder *TL, BL, TR,*

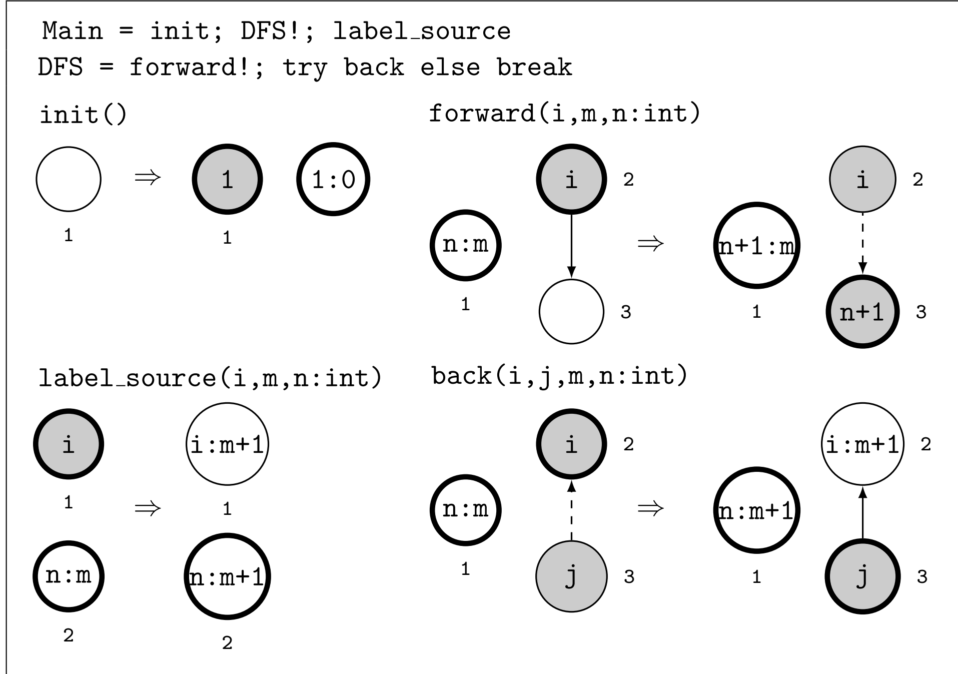


Figure 6.2.: The GP2 program `dfs`

BR. Note that the reverse postorder is different from the preorder: *BL* was the last node to be visited, but it was not the last node to be fully explored. We note that DFS may also be used to categorise edges, which reveals interesting properties about the graph [THCRS09], but these properties are not relevant to the programs in this chapter.

Remark 10. In the context of graph-traversing GP 2 programs, we say a host graph node is *visited* when it first participates in the match of a successful rule application.

Remark 11. We use the term *blank* throughout this chapter to refer specifically to unmarked nodes and edges labelled with the empty list. We use the term *blank graph* to refer to a graph whose nodes and edges are blank.

The GP 2 program `dfs`, shown in Figure 6.2, is a concrete realisation of the DFS algorithm. It performs a directed depth-first search on the blank host graph starting at an arbitrary node v. Nodes not reachable from v are not visited by the search. The output graph is the host graph with two changes:

1. The nodes visited in the DFS are labelled $pre : post$ as illustrated in Figure 6.1. During the computation, visited nodes are only labelled with *pre* until they are finished, at which point *post* is appended to the label.

2. An additional root node stores two counts of the number of nodes visited in the DFS (obtained through the preorder and postorder labelling). This illustrates that graph traversal can perform a global computation on host graphs.

The program maintains two root nodes. The grey root node in the host graph is used to navigate the graph in a depth-first manner. The second unmarked root

node, called the *counter*, is created by the program. It stores the current preorder count and the current postorder count. It assigns its preorder count to the root node after it is moved forward, and its postorder count before it is backtracked. When the program terminates, these counts will both be equal to the number of nodes in the host graph that are reachable from the root node.

The rule `init` prepares the search by matching an arbitrary host graph node called the *source*. The source is rooted and coloured grey which marks it as visited. It is also labelled with its preorder position (1). The rule also creates the counter. The procedure `DFS` is applied as long as possible. A loop iteration has two steps. First, it moves forward along a path of unmarked edges passing through blank nodes for as long as possible, then moves back one step when forward movement is no longer possible. Each rule application moves the root node along the path, greying blank nodes and labelling them with the next preorder number from the counter. Explored edges are dashed. Unlike in the previous example, this is not a permanent mark. Instead, it acts as a trail of breadcrumbs to facilitate backtracking once a node is finished. Visited nodes are greyed and labelled, so they cannot be matched as the third node in `forward`'s left-hand side. At some point, `forward` is no longer applicable, either when the root node has no outgoing edges, or when the targets of all of its outgoing edges have been visited. In either case, the root node is finished. The rule `back` appends the current postorder count to its label, moves the root node back one step along the path of dashed edges, and unmarks it. After a single application of `back` the next loop iteration starts, which searches for an unexplored outnode from the current root node. In this way, all outedges of visited nodes are explored, and every node reachable from the source node is reached. `DFS!` terminates when `back` is no longer applicable, at which point the root node is the node matched by `init`. The construct `try back else break` is used to exit the loop when `break` fails without reverting the graph to the state before entering the current loop iteration. Finally, `label_source` appends the postorder count to the source because it is not the subject of a `back` rule application.

Figure 6.3 shows an example run of `dfs` on the same graph as in the previous example. The top left graph is the state after applying `init`. The top right graph is the state after two applications of `forward`. Observe that the nodes are labelled in the order in which they are visited. The bottom right graph is the state after two applications of `back` and one application of `forward`. The rightmost nodes have been labelled with their postorder positions, but the top left node has not since the top left node is not yet finished: the search has continued on its second outgoing edge $1 \to 4$. The bottom left graph is the output graph of the program. The counter is labelled with two integers, both equal to the number of nodes in the host graph.

Remark 12. In the following proofs, we use *root node* to refer only to the root node that was part of the original host graph, i.e. not the counter node.

Lemma 4. Let G be a blank graph with source v. The following property is an invariant of the loop `DFS!`: the root node is reachable from v through a path of

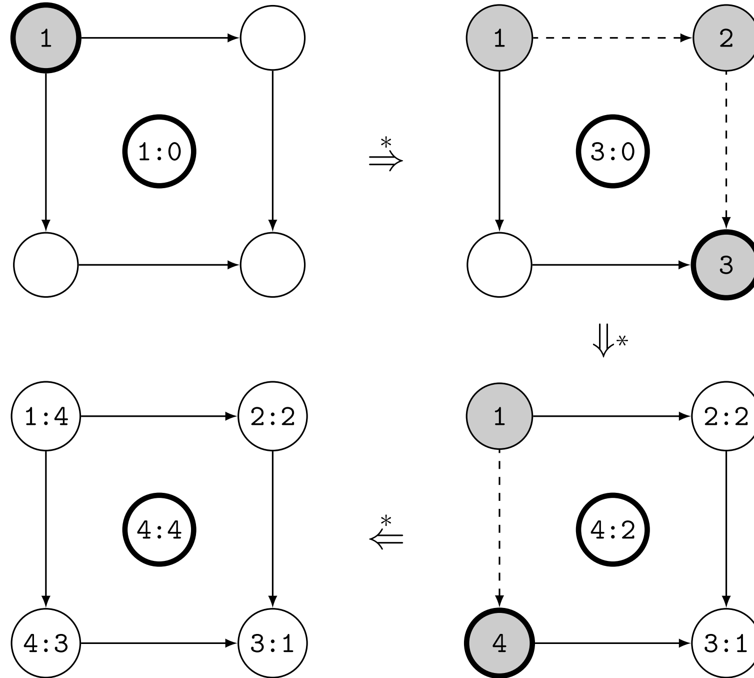


Figure 6.3.: Example execution of `dfs`

dashed edges. Every node in this path is grey. There are no other dashed edges.

Proof. The property trivially holds immediately after the application of `init`. Let w be the root node. Assume there is a path, possibly empty, of dashed edges from v to w consisting of only grey nodes, with no other marked edges in the graph. After an application of `forward`, this path has been extended by a single marked edge connecting w and one of its outgoing neighbours w'. w' is the new root node, w' is grey, and the rule creates no additional marked edges. Therefore the property still holds. A similar argument shows that `back` also preserves the invariant. □

Lemma 5. Let G be a blank graph. The program `dfs` terminates when run on G.

Proof. We only need to prove termination of the loop `DFS!`; the rest of the program consists of single rule applications. Let $>$ be the following lexicographic ordering on graphs: $G > H$ if G contains more blank nodes than H, or if G and H contain the same number of blank nodes and G contains more dashed edges than H. If `forward` is applied to G to give H, then $G > H$ because `forward` marks and labels a blank node. In addition, `back` is applied to G to give H, then $G > H$ because `back` undashes an edge and does not change the number of blank nodes in the graph. It follows that `DFS!` terminates because there are a finite number of graphs less than the host graph with respect to the given ordering. □

Lemma 6. Let G be a blank graph, and let v be the source of the DFS. The program `dfs` visits all nodes reachable from v.

Proof. We give a proof by contradiction. Assume that there exists a node w reachable from v that is unvisited when the loop `DFS!` terminates. w is blank because it has not been visited. w is not the root node, since neither `forward` nor `back` makes a blank node the root. w is not the target of an edge outgoing from a visited node w': since w' is visited, it must have been the root at some stage in the computation. If w were the target of an outgoing edge of w', then `forward` would have matched for $w' \to w$, either when w' was first made the root or immediately after it was made the root from an application of `back`. Therefore, the source of any edge whose target is w is an unvisited node. We can inductively extend this argument to conclude that all nodes from which w can be reached are unvisited. However, v is visited by `init`, contradicting the assumption that w is reachable from v. Therefore `dfs` visits all nodes reachable from v. □

Proposition 3 (Correctness of `dfs`)**.** Given a blank input graph G, `dfs` chooses a source node v and labels all nodes w reachable from v with a two-element list, where the first element is the preorder position of w, and the second element is the postorder position of w.

Proof. We refer to the first and second elements of the counter's list by *pre* and *post* respectively. First, `init` matches a node v and labels it 1, which is clearly v's preorder position. The rule also sets *pre* to 1 and *post* to 0. A node w is visited when it is matched by `forward`, which labels w with $pre + 1$ and increments *pre*. No other rule modifies the value of *pre*, so the node labelling respects the definition of preorder. The rule `forward` is looped, so an application of `back` is only attempted when the current root node does not have an outgoing edge whose target is unvisited, precisely when that node is finished. An application of `back` labels a finished node with $post + 1$ and increments *post*. No other rule modifies the value of *post*, so the node labelling respects the definition of the postorder. Lemma 5, `dfs` terminates, which guarantees a valid output graph. Finally, Lemma 6 ensures that all nodes reachable from v are visited by `dfs`. □

Proposition 4 (Complexity of `dfs`)**.** The program `dfs` runs in linear time on host graphs of bounded degree, and in quadratic time on host graphs of unbounded degree.

Proof. We assume that the host graph is blank because this provides the worst case complexity. The rule `init` matches in constant time because all nodes in the host graph are valid matches. All other rules are fast rule schemata. By Theorem 2 (see Section 4.5.1), they are applied in constant time on host graphs of bounded degree and they are applied in linear time on host graphs of unbounded degree. The rule `label_source` is applied once, while the rules `forward` and `back` are applied a linear number of times in the node size of the graph. It follows that the program runs in linear time on host graphs of bounded degree, and quadratic time otherwise. □

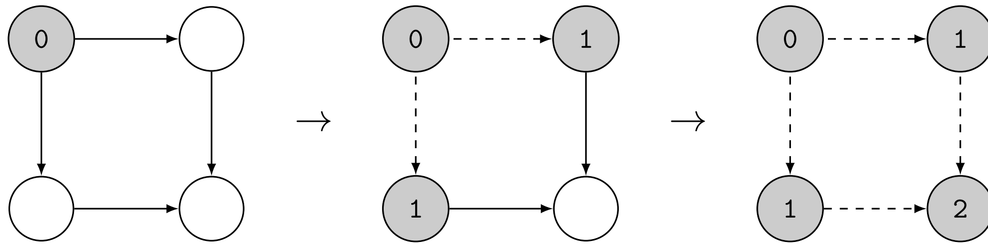


Figure 6.4.: Illustration of a breadth-first search.

6.2.2. Breadth-first Search

Like DFS, a breadth first-search of a graph starts by visiting an arbitrary node v. Each step of the search visits the target of an unexplored outgoing edge from the *least* recently-visited node v. In this way, all edges from a single node are explored before the outgoing edges of any other node. Again, this process repeats until all outedges of all visited nodes have been explored. Search continues by visiting an arbitrary unvisited node. Search terminates when all nodes in the graph have been visited.

The order of nodes visited by BFS makes it a natural way to measure the least number of edges it takes to get from the start node to any other reachable node in the graph. Formally, the *distance* of a node w from a node v is the number of edges on the path from v to w containing the fewest edges. Breadth-first search (BFS) can be used to compute the distance from a source node to all nodes reachable from the source as illustrated in Figure 6.4. As in the DFS example, search commences at the top left node. Visited nodes are grey and explored edges are dashed. The nodes are labelled with their distance from the top left node.

Figure 6.5 shows a GP 2 program `bfs` that performs a directed BFS on a singly-rooted graph. The rule `unroot_blue` is not shown, which is the same as `unroot_blue` except the left-hand node is blue. The output is the host graph with the reachable nodes labelled with their distance from the 0-labelled node.

Like `dfs`, a root node is created to keep a record of the node count. An arbitrary node from the host graph, which we again call the source, is coloured red and labelled 0 by `init`. Each loop iteration starts with an extension phase, where all unmarked nodes outgoing from a the root node with distance d from the source are marked with the contrasting colour and labelled with $d + 1$. This phase is repeated for all nodes at distance d before switching to the next layer by rooting an arbitrary node at distance $d + 1$. Marks are used to control these phases. When the final iteration is complete, all nodes are unmarked except a single root node marked either red or blue, which is cleaned up by the final rule application.

Figure 6.6 shows an example run of `bfs` on the same graph as before. The top left graph is the state after applying `init`. The top middle graph is the state in the first outer loop iteration immediately after `Extend!` is executed. The top right and bottom left nodes have been matched by `extend_red`. They are blue, rooted, and labelled with their distance from the top left node. There

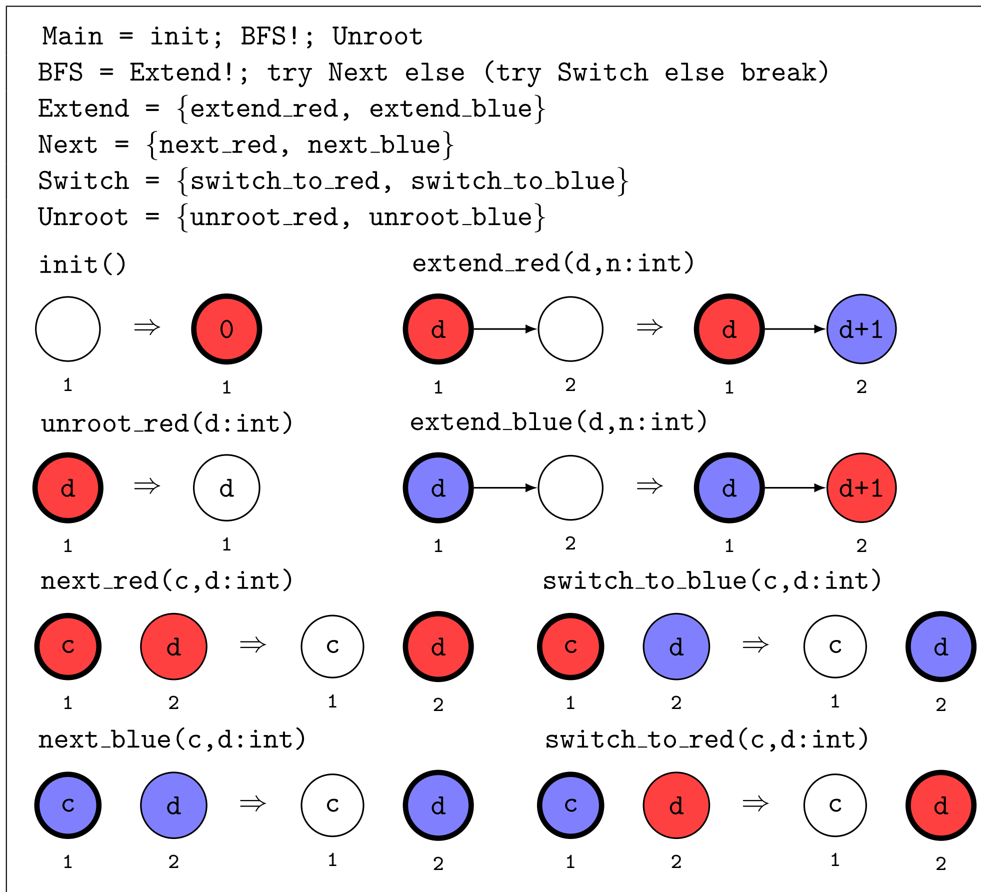


Figure 6.5.: The program bfs

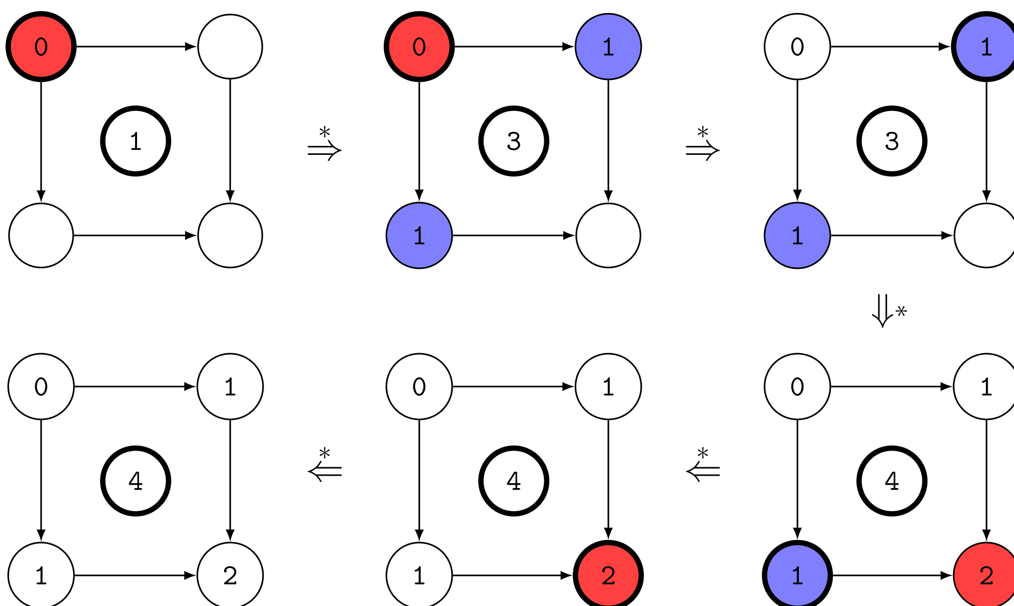


Figure 6.6.: Example execution of bfs

are no other red nodes, so `next_red` fails, which triggers execution of the inner `try-then-else` statement. The application of `switch_to_blue` uncolours the top left node and nondeterministically roots the top right node. At the start of the second outer loop iteration, the state is the graph at the top right. An application of `extend_blue` and `next_blue` gives the bottom right graph. No extension rules are applied, and `next_blue` fails, so the root node is switched to a red node before the next iteration. No rules are applicable in this iteration, so the outer loop breaks by the `break` statement in the inner `try-then-else`, and `unroot_red` is applied to give the output graph at the bottom left of the figure.

Lemma 7. Let G be a blank graph with source v. When `bfs` is applied to G, the following property is invariant in the loop `BFS!`: the root node r is marked either red or blue and is labelled with its distance d from v, and all other nodes at distance d from v are unrooted, marked with the same colour as r and labelled d.

Proof. The invariant trivially holds immediately after the application of `init` because the root v is coloured red and labelled 0 by the rule. Consider the first iteration of the loop. First, `extend_red` is applied as long as possible. Each rule application seeks a blank target node w of one of v's outgoing edges. There are two cases.

Case 1. *A node w exists.* Then $w \neq v$ because of injective matching, so w's distance from v is 1. This node is marked blue and labelled 1 by `extend_red`. After the execution of `Extend!`, all targets of edges outgoing from v are blue and labelled 1. The program enters the `try-then-else` statement. The procedure `Next` fails because v is the only red node in the graph, and control proceeds to the inner `try-then-else` statement. `switch_to_blue` succeeds, without loss of generality matching v and w. v is unrooted and unmarked, while w remains blue and becomes a root node. Control reaches the end of the current iteration. The invariant holds: the root node w is blue and labelled 1, its distance from v. All other nodes at distance 1 from v are blue.

Case 2. *No such node w exists.* `Extend!` exits after zero rule applications. The program enters the `try-then-else` statement. `Next` and `Switch` fail because v is the only marked node in the graph. `BFS!` exits by the `break` statement, after which `unroot_red` uncolours and unroots v. The invariant trivially holds because there are no root nodes in the graph.

Now assume that the invariant holds after the kth loop iteration. Without loss of generality, let $k + 1$ be even. Then the single root node w is blue and labelled with its distance k from v. The $k + 1$th execution of the loop body first executes `Extend!` which finds all targets of outgoing edges of w, marks them red, and labels them with $k + 1$. If another blue node w' exists, it is made the root node by `next_blue`, which also unroots and unmarks w. Another execution of `Extend!` repeats the process for another set of nodes at distance $k + 1$ from v. When the last blue node w'' is processed in this way, `Next` fails. w'' is unmarked and unrooted by either `switch_to_red` or `unroot_blue`. In the former case, the

red node matched by `switch_to_red` is rooted, and all other nodes at distance $k + 1$ are red and labelled $k + 1$. In the latter case, no root nodes exist. In both cases the invariant holds. □

Lemma 8 (Termination of `bfs`). Given a blank input graph G with a single root node v, the program `bfs` terminates.

Proof. Clearly `init` and `Unroot` terminate, so we only need to prove termination of the loop `BFS!`. Let $>$ be the following lexicographic ordering on graphs: $G > H$ if G contains more blank nodes than H, or if G and H contain the same number of blank nodes and G contains more unmarked nodes than H. Then, if `extend_red` or `extend_blue` is applied to G to give a new graph H, then $G > H$ as the extension rules mark and label a blank node. In addition, applying a `next` rule or a `switch` rule to G to give the graph H implies $G > H$ since those rules preserve the number of blank nodes and unmarks one node. It follows that `BFS!` terminates because there are a finite number of graphs less than the host graph with respect to the given ordering. □

Lemma 9. Let G be a blank graph, and let v be the source of the BFS. The program `bfs` visits all nodes reachable from v.

Proof. We give a proof by contradiction. Assume that there exists a node w reachable from v that is unvisited when the loop `BFS!` terminates. w is blank because it has not been visited, therefore w is not the root node, since the root node is always marked. It follows that w is not the target of an edge outgoing from a visited node w': since w' is visited, it must have been the root at some stage in the computation. If w were the target of an outgoing edge of w', then w would have been marked red or blue by one of the `extend` rules and later made the root node by either a `next` rule or a `switch` rule. Therefore, the source of any edge whose target is w is an unvisited node. We can inductively extend this argument to conclude that all nodes from which w can be reached are unvisited. However, v is visited by `init`, contradicting the assumption that w is reachable from v. Therefore `bfs` visits all nodes reachable from v. □

Proposition 5 (Correctness of `bfs`). Given a blank input graph G, `bfs` chooses a source node v and labels all nodes w reachable from v with its distance from v.

Proof. Follows directly from Lemmas 7, 8, and 9. □

Proposition 6 (Complexity of `bfs`). The program `bfs` runs in quadratic time.

Proof. We assume that the host graph is blank because this provides the worst case complexity. The rules `init`, `unroot_red` and `unroot_blue` are applied at most once and match in constant time. The extending rules are fast rule schemata applied a linear number of times in the size of the host graph. By Theorem 2 (see subsection 4.5.1), they are applied in the worst case in linear time. The `next` and `switch` rules are not fast rule schemata. Their complexity is linear because they search for one non-root node, and they are applied a linear number of times. It follows that the program runs in quadratic time. □

Outside of graph programming, the computational complexity of both DFS and BFS is linear in the size of the graph. Imperative implementations can achieve linear time with the use of auxiliary data structures to aid the search. DFS implementations use a stack to store the branching points, namely the list of nodes to be visited next, because the algorithm continues the search from the most recently encountered branch. In contrast, BFS implementations use a queue because the next branch to explore is the least recently encountered one. The analogous mechanism for graph programs is root nodes. Each root node represents a branching point. GP 2 can naturally perform DFS with fast rule schemata while maintaining only one root node since the graph traversal can efficiently search for the most recent branching point while backtracking, a step that has to be taken in any case. On the other hand, this cannot be achieved so easily when programming BFS. Instead we use a single marked root node to model the current branching point, representing other nodes in the "queue" with the same mark. The consequence is that we do not have a program consisting entirely of fast rule schemata, which gives a greater complexity than DFS for host graphs with bounded degree, although the general worst case complexity is the same. We note that it should be possible to program BFS using only fast rule schemata by explicitly simulating a queue. The idea is that an external root node acts as a global pointer to a node at the current depth, and auxiliary edges connect nodes at the same depth. Initial attempts to write such a program have revealed that it is very difficult to achieve using only rules that are rooted and connected.

6.3. Case Study: 2-Colouring

Vertex colouring has many applications [Ski08] and is among the most frequently considered graph problems. We focus on 2-colourability: a graph is *2-colourable*, or *bipartite*, if one of two colours can be assigned to each node such that the source and target of each non-loop edge have different colours. We first give a general result that enables us to show correctness of the graph programs that follow.

Lemma 10. Consider the algorithm that labels nodes of a connected, undirected graph G by assigning each node a colour from the set {red, blue} as follows: first assign an arbitrary node the colour red, then repeat the following procedure until all nodes have a colour: nondeterministically find an uncoloured node connected to an coloured node v and label it with the contrasting colour to that of v. Then the following statement holds: G is not bipartite if and only if, at any point in the algorithm, two connected nodes have the same colour.

Proof. If G is not bipartite then, by definition, there is no way to assign integers to nodes without labelling two connected nodes with the same integer.

We prove the other direction by the contrapositive: we assume G is bipartite, and we show that the algorithm never assigns the same label to two connected nodes. Let v be the initial node of the algorithm. v is coloured red. We prove, by

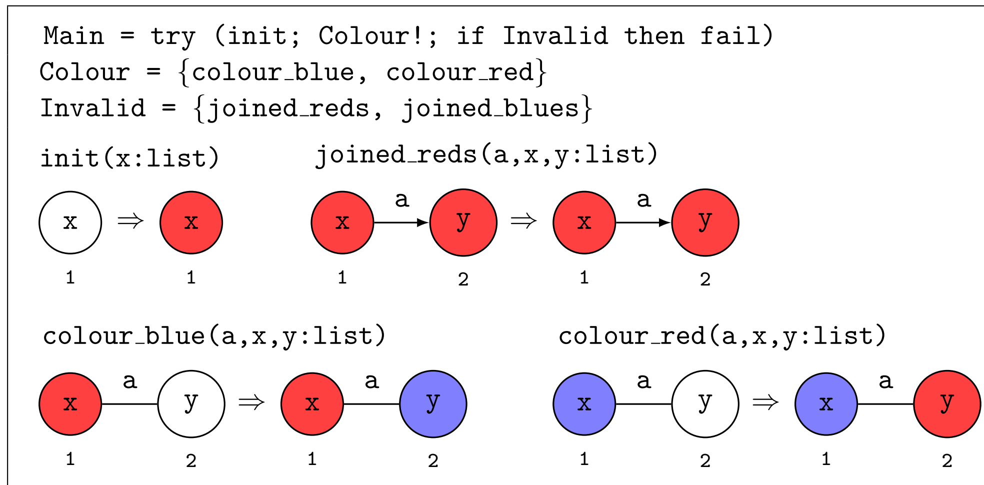


Figure 6.7.: The program `2colouring`

induction, that no neighbours of v can be coloured red. If v has one neighbour, clearly the algorithm must colour that neighbour blue.

Assume that if v has fewer than n neighbours, the algorithm cannot colour any of them red. Let v have n neighbours $w_1, \ldots, w_n$. By the induction hypothesis, without loss of generality, none of $w_1, \ldots, w_{n-1}$ are red, so we assume they are blue. If w_n is not reachable from any of $w_1, \ldots, w_{n-1}$ except through v, then w_n must be coloured blue because it is only colourable via the edge $v \to w_n$. So assume there is a path P from one of v's neighbours, say w_1, to w_n, that does not contain v. w_1 is blue, so for w_n to be coloured red, P must contain an odd number of edges. It follows that G contains a cycle of odd-length consisting of the two-edge path w_n, v, w_1 and the odd-length path P from w_1 to w_n. Therefore G is not bipartite, a contradiction. By induction, no neighbours of v can be assigned the same colour as v.

This argument can be extended to every other node in the graph. Therefore a node can never be assigned the same colour as one of its neighbours in a bipartite graph. □

The following sections present GP 2 programs that find a 2-colouring of a graph with the algorithm described above (adapted to GP 2's directed graphs). The input to these programs is a connected, unmarked and unrooted graph G. If G is bipartite, the output is a valid 2-colouring of G. Otherwise, the output is G. The first program contains no roots in its rules. The other two programs are rooted: one colours the graph using a depth-first traversal, while the other uses a breadth-first traversal. The colouring rules of all three programs use bidirectional edges to match host graph edges independently of their direction.

6.3.1. Non-rooted 2-colouring

Figure 6.7 shows the non-rooted GP 2 program `2colouring`. `joined_blues` is omitted: it is the same as `joined_reds`, except its nodes are blue.

Lemma 11. Given a connected graph G, the program `2colouring` returns a 2-colouring of G if G is bipartite, otherwise it returns G.

Proof. Using the rule `init`, the program first marks an arbitrary node of G and colours it red. Then the loop `Colour!` nondeterministically finds uncoloured nodes that are connected to coloured nodes and colours them with the contrasting colour. The rules in `colour` decrease the number of uncoloured nodes, so `Colour!` will terminate precisely when each node has a colour.

By Lemma 10, after `Colour!` terminates, if G is bipartite, then the current graph is a valid 2-colouring of G. Otherwise, it contains a non-looping edge connecting two nodes of the same colour. The `if-then-else` statement uses `Invalid` to check for such an edge. If one exists, `Invalid` succeeds. The **`then`** branch triggers a fail which, by the semantics of `try-then-else`, causes the host graph G to be returned. Otherwise, the `else` branch is taken which does nothing. After that, the **`then`** branch of the `try-then-else` statement is taken which retains the current graph. □

6.3.2. Rooted 2-colouring

Figure 6.8 shows a rooted 2-colouring GP 2 program that colours the graph during a depth-first traversal. The rules `colour_red` and `joined_blues` are omitted, which are the "inverted" versions of the rules `colour_blue` and `joined_reds` with respect to the node marks. In particular, the right-hand side of `joined_blues` also has a grey root node.

A glance at the program text reveals that `dfs-2colouring` is more complicated than its non-rooted counterpart. At its core, it is an undirected depth-first search in which the source node is chosen nondeterministically by the program. The colouring rules and back rules correspond to the `forward` and `back` rules of `dfs` respectively. Each visited node is coloured with the contrasting colour to the previous node in the traversal. Unlike `dfs`, backtracking does not undo the marking performed by the traversal because the global colouring is preserved for the output.

The most significant departure from `2colouring` is the placement and behaviour of `Invalid`. In `dfs-2colouring`, this check is performed immediately after a node is coloured. If the 2-colourability is violated, the root node is marked grey, which acts as a flag for non-bipartiteness. The check is performed by fast rule schemata, which does not improve the worst case complexity compared to the global check at the end of `2colouring`, because in both cases the check is performed on the neighbourhood of every node during program execution. The advantage to performing a local check at each step is that a host graph can be proven to be non-bipartite before it has been completely coloured. In addition, since every rule schema is fast, all matching is performed in constant time on host graphs of bounded degree.

Figure 6.9 shows the execution of `dfs-2colouring` on the host graph in the upper-left of the diagram. This graph is clearly not 2-colourable. The rule `init` colours node 1 red. The rule `colour_blue` nondeterministically matches the edge

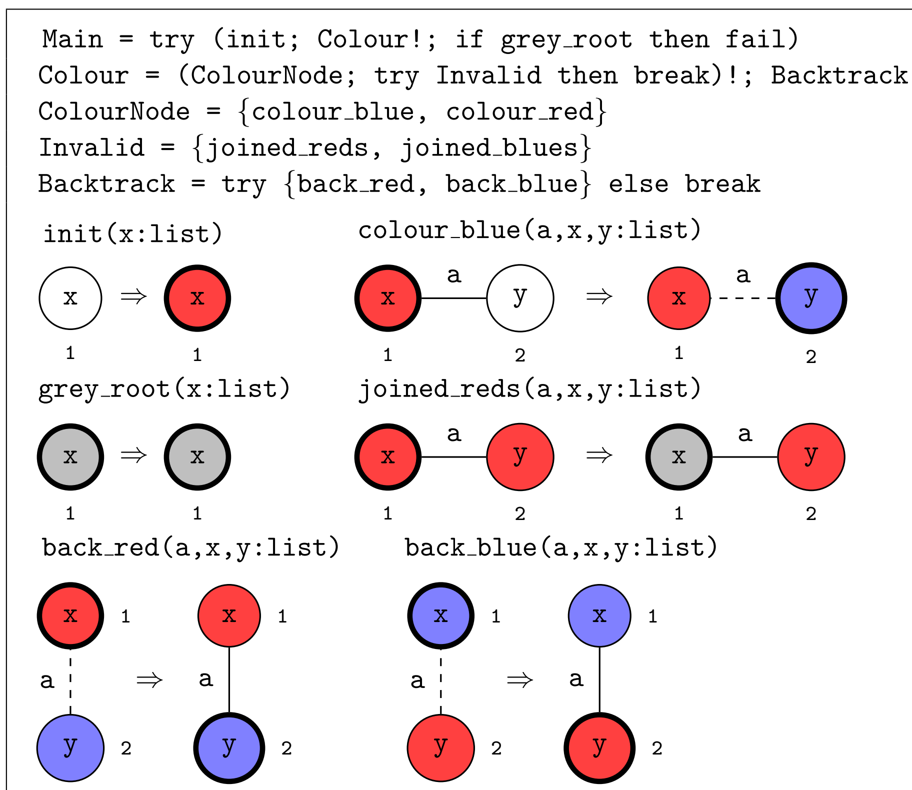


Figure 6.8.: The program `dfs-2colouring`

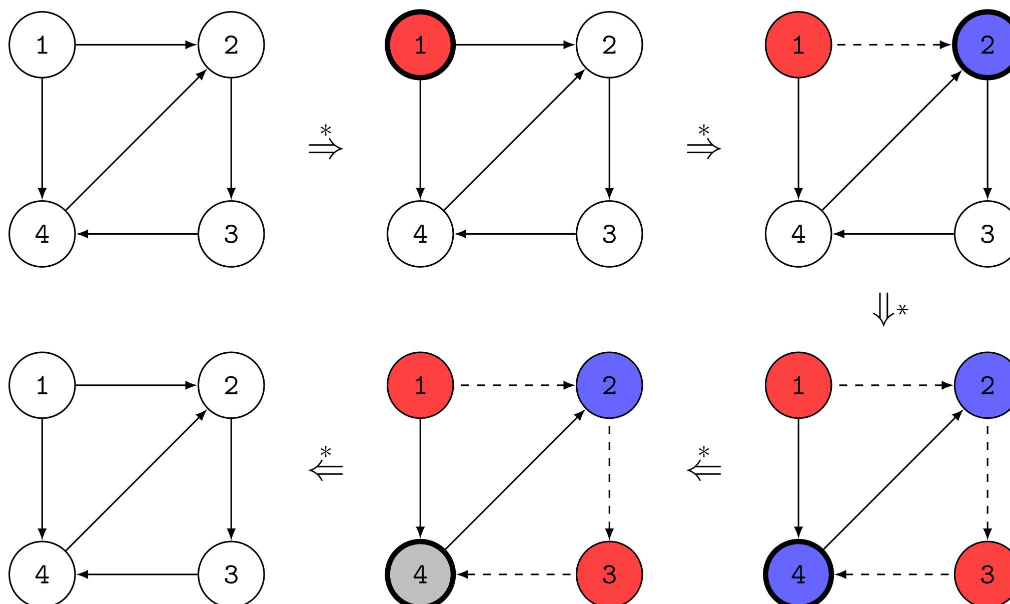


Figure 6.9.: Example run of `dfs-2colouring`

$1 \to 2$. It roots node 2, colours it blue and dashes the edge. The colouring rules are applied twice more to give the lower-right graph. At this point the rule `joined_blues` matches the edge $4 \to 2$. This colours the root node grey. The inner loop breaks, and control passes to `Backtrack`. Both back rules fail because neither match a grey root node. This causes the outer loop to break. Finally, `grey_root` succeeds, causing the `try` statement to fail and return the original graph.

The following two results formally establish the correctness of `dfs-2colouring`.

Lemma 12. Let v be the unique root node of a connected and unmarked graph G. Upon executing `dfs-2colouring` on G, after the application of the rule `init`, the following property is an invariant of the program: the current root node is reachable from v via a path of dashed edges. Every node in this path is either blue or red. There are no other marked edges.

Proof. Follows from Lemma 4. □

Proposition 7 (Correctness of `dfs-2colouring`)**.** Given a connected, unmarked and unrooted host graph G, the program `dfs-2colouring` returns a 2-colouring of G if G is bipartite, otherwise it returns G.

Proof. First, we prove that the program terminates. If we simplify the program to preserve the loop structure, removing only statements containing single rule applications and control statements, and inlining the `Colour` procedure, the program text reads `try (ColourNode!; Backtrack)!`. An almost identical argument to that of Lemma 5 can be used to prove that this loop always terminates. We split the remainder of the proof into two cases.

Case 1. *G is bipartite.* We can discard the `try Invalid then break` clause because, by Lemma 10, `Invalid` is never successful. The `Colour` procedure reduces to `(ColourNode!; Backtrack)!`. Since the input G is connected, we use the argument of Lemma 6, easily adaptable to undirected paths, to prove that every node in G is visited. By Lemma 12, these nodes are marked either red or blue by the loop. By Lemma 10, these nodes are marked in a way that does not violate 2-colourability. Only the rules in `Invalid` colour the root node grey, so the `if-then-else` statement in the body of `Main` fails, and the `try-then-else` statement suceeds, terminating the program with the 2-coloured graph.

Case 2. *G is not bipartite.* By Lemma 10, at some point during program execution, the procedure `Invalid` succeeds. This colours the root node grey and breaks the loop in `Colour`. Neither `back_red` nor `back_blue` match a grey root node, so `Backtrack` will break the `Colour!` loop. The if statement in the body of `Main` succeeds, and the try statement fails, terminating the program with G. □

It is not so straightforward to convert `bfs` to a BFS-based 2-colouring program. In order to output a 2-colouring, the `next` and `switch` rules should unmark the root node. However, this causes the program to loop forever on non-trivial host

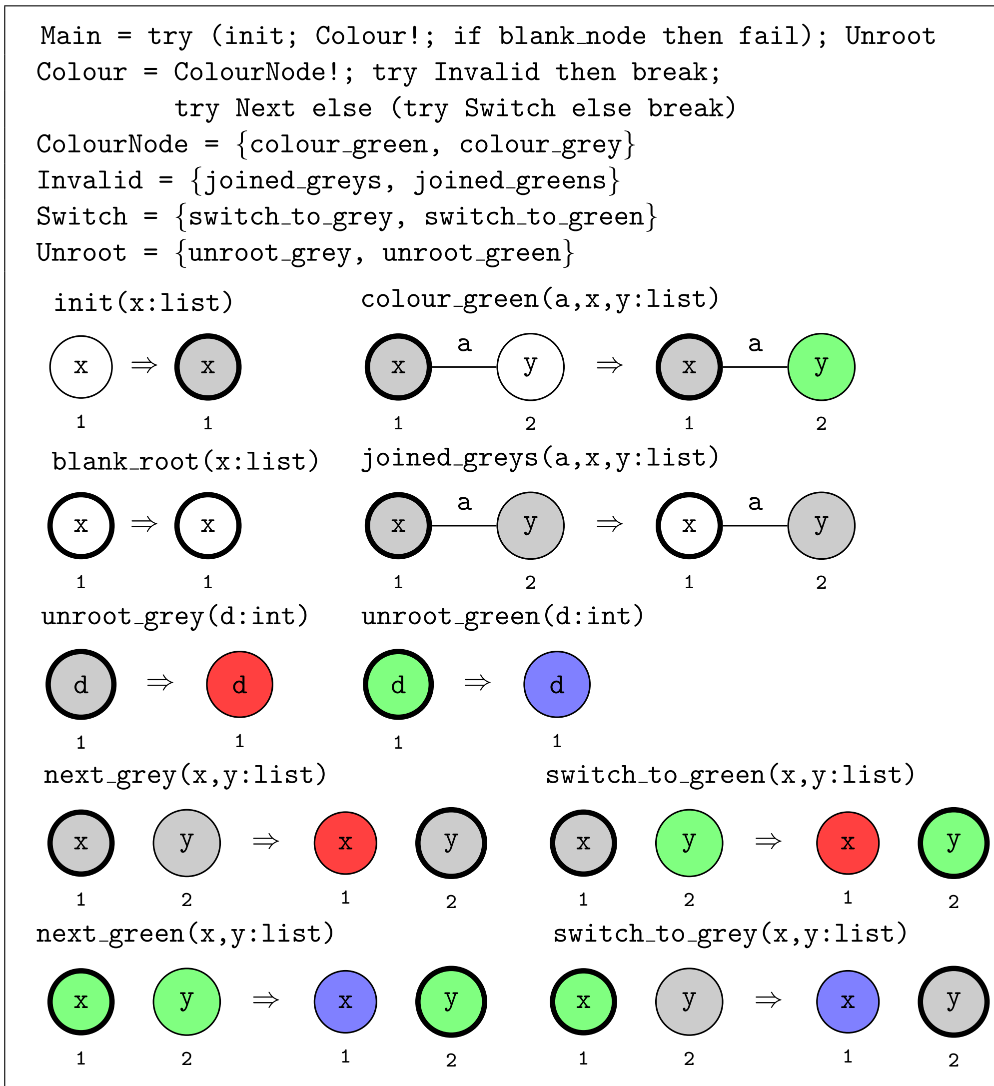


Figure 6.10.: The GP2 program `bfs-2colouring`

graphs, because they can match the same pair of nodes repeatedly. In terms of Lemma 8, these rules do not create a smaller graph with respect to the graph ordering in the proof. We address this issue in the program `bfs-2colouring` by using four marks as shown in in Figure 6.10. Grey and green are used as the initial 2-colouring colours. When grey and green nodes are finished, they are remarked to red and blue respectively by one of the `next`, `switch` and `unroot` rules. In this way we can guarantee termination.

Generally speaking, `bfs-2colouring` combines the `bfs` algorithm with the local invalid edge checking of `dfs-2colouring`. In this case, an unmarked root node is used as the non-bipartite flag. The rules play the same roles as they did in `bfs` (the `expand` rules have been renamed to `colour_green` and `colour_grey`). Two rules are not displayed: `colour_grey` and `joined_greens` which are the inverted versions of `colour_green` and `joined_greys` respectively. Again, the source node is chosen nondeterministically by the program. The graph traversal grows from the source akin to `bfs`. The program needs to check for invalid edges

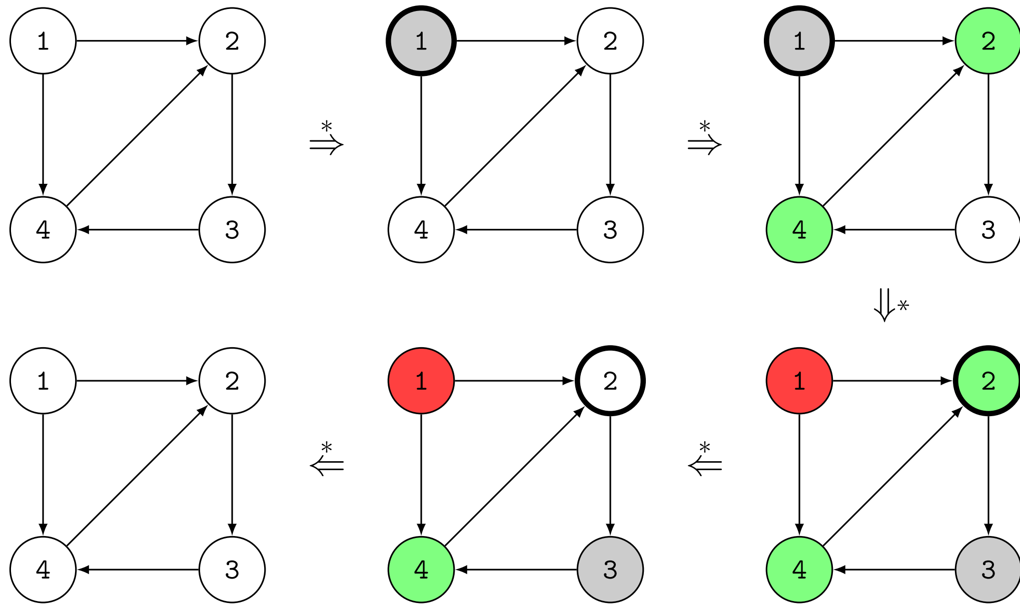


Figure 6.11.: Example run of `bfs-2colouring`

on each node during its time as a root. This is achieved by placing the familiar `try Invalid then break` clause directly after the `ColourNode!` loop. In this way, the current root node is checked before it is unrooted.

Figure 6.11 shows an execution of `bfs-2colouring` on a non-bipartite graph. The top row proceeds as `bfs`. The top right graph contains an edge whose end-points are the same colour, but this is not immediately detected by the program because neither of these nodes are rooted. The transition from the top right graph to the bottom right graph is done by `switch_to_green`, which matches the top left and top right nodes, and `colour_grey` (the start of a new loop iteration), which matches $2 \to 3$. When `Invalid!` is executed, `joined_greens` matches which unmarks the root node and breaks the loop. In the `Main` procedure, `blank_node` succeeds which causes its containing `try-then-else` statement to fail, returning the original graph.

Proposition 8 (Correctness of `bfs-2colouring`)**.** Given a connected, unmarked and unrooted host graph G, the program `bfs-2colouring` returns a 2-colouring of G if G is bipartite, otherwise it returns G.

Proof. Ignoring the clearly terminating `try Invalid then break`, we get the same program structure as `bfs`. Termination is proved by the same argument as in the proof of Lemma 8 with $>$ defined as follows: $G > H$ if G contains more unmarked nodes than H, or if G and H contain the same number of unmarked nodes and G contains more grey nodes or more green nodes than H. Correctness of the output graph follows from a case analysis as in the proof of Proposition 7, using Lemma 10. □

6.3.3. Complexity Comparison

The complexity of `2colouring` is at least quadratic in the size of the host graph. Each colouring rule is applied a linear number of times and involves a linear-time node search and an edge search bounded by the degree of the host graph. The equivalent rooted programs were constructed with the goal of using fast rule schemata as much as possible in order to improve the theoretical complexity. We formalise the complexity of both rooted 2-colouring programs.

Proposition 9 (Time complexity of `dfs-2colouring`)**.** On unmarked input graphs, the running time of `dfs-2colouring` is linear in the size of graphs with bounded node degree, and quadratic otherwise.

Proof. First, `init` is applied in unit time because every node in the host graph is a valid match of the left-hand side. All other rule schemata are fast. By Corollary 1, we know that each rule schema takes only constant time on rooted graphs of bounded degree. Moreover, none of the rule schemata increase the degree of any node or increase the number of roots. Therefore repeated rule schema applications in program runs preserve the assumptions of Corollary 1,

To show that the running time of `dfs-2colouring` is linear in the size of the input graph, we demonstrate that the maximal number of rule schema applications is linear. The rules `init` and `grey_root` are applied at most once in a program run. Next, notice that `colour` reduces the number of unmarked nodes and `back` does not increase this number. Hence `colour` is applied at most n times, where n is the node size of the host graph. The procedures `Invalid` and `Backtrack` are executed at most once for each application of a colouring rule. Therefore the total number of rule applications is $O(n)$.

Now we consider host graphs of unbounded degree. Observe that no left-hand sides in the program contain more than one edge. Therefore, by Figure 4.8 and its analysis, matching a single rule is no worse than linear. Since there are a linear number of rule applications, the overall time complexity is quadratic. □

To illustrate the second part of the proposition with a concrete example, consider the execution of `dfs-2colouring` on a blank star graph G with n edges. Assume that `init` matches the central node. This is a constant time match. The program then iterates the following sequence of rule applications n times: (1) Apply `colour_blue` to one of the uncoloured nodes branching out from the central node. (2) Check for violation of 2-colourability with `Invalid`. This always fails. (3) Try to colour a node adjacent to a leaf node. This always fails. (4) Apply `back_blue`. Each application of `colour_blue` takes $O(n)$ time because in the worst case, all n outgoing edges need to be examined for a valid match. The other rules are applied in constant time since each leaf node has only one incident edge. This gives a total running time of $n^2 + 3n + 1 = n + (n+1)^2 = O((n+1)^2) = O(|V_G|^2)$.

Proposition 10 (Time complexity of `bfs-2colouring`)**.** On unmarked input graphs, the running time of `bfs-2colouring` is quadratic in the size of the host graph.

Proof. Follows directly from Proposition 6. □

6.3.4. Experimental Results

To experimentally validate the theoretical complexity of the 2-colouring programs, we executed them using the GP 2 implementation. For the first experiment, the programs were executed on *square grid graphs* (grids). The reasons are threefold.

1. Grids are 2-colourable. This guarantees that all three programs perform the same computation, namely matching and colouring every node in the graph. For non-2-colourable graphs, the rooted programs may detect non-2-colourability before all the nodes are matched.

2. Grids have bounded node degree, which in particular tests the theoretical linear complexity of `dfs-2colouring`.

3. Grids have a simple structure that admits relatively simple generation of large host graphs.

A concrete example of the structure of the grid graphs we use for testing is in Figure 6.12.

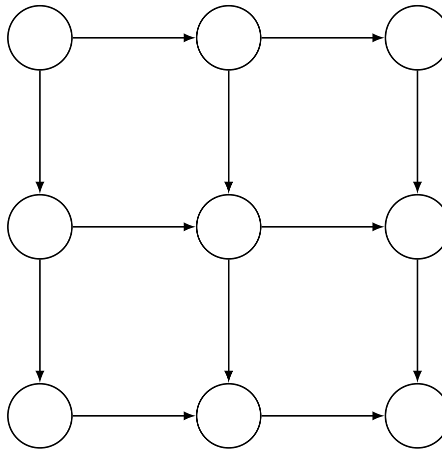


Figure 6.12.: An example grid graph

The results are given in Table 6.1 and Figure 6.13. Both rooted programs greatly outperform the non-rooted program. `dfs-2colouring` runs faster than `bfs-2colouring`, although the program based on breadth-first search performs a lot better than its theoretical complexity. The plots confirms the theoretical results that `dfs-2colouring` performs linearly with respect to the size of the host graph with bounded degree and that `bfs-2colouring` performs in quadratic time. An interesting observation is that `2colouring`'s runtime grows at approximately the same rate as `bfs-2colouring`, but the latter is an order of magnitude faster. In both programs, the matching of unrooted nodes a linear number of causes is the cause of the quadratic complexity. The striking gap in performance arises because `2colouring` matches unrooted nodes for every rule application, whereas

Grid Size	2colouring	bfs-2colouring	dfs-2colouring
10,000	3.808	0.212	0.041
20,164	15.451	0.768	0.69
30,276	35.084	1.671	0.097
40,000	62.054	2.863	0.127
50,176	99.19	4.459	0.154
60,025	145.219	6.392	0.181
70,225	221.102	9.049	0.213
80,089	246.248	12.675	0.237
90,000	334.893	17.163	0.267
102,400	438.987	23.997	0.311

Table 6.1.: Experimental results of three 2-colouring programs. Grid size is given by the number of nodes, and runtime is given in seconds

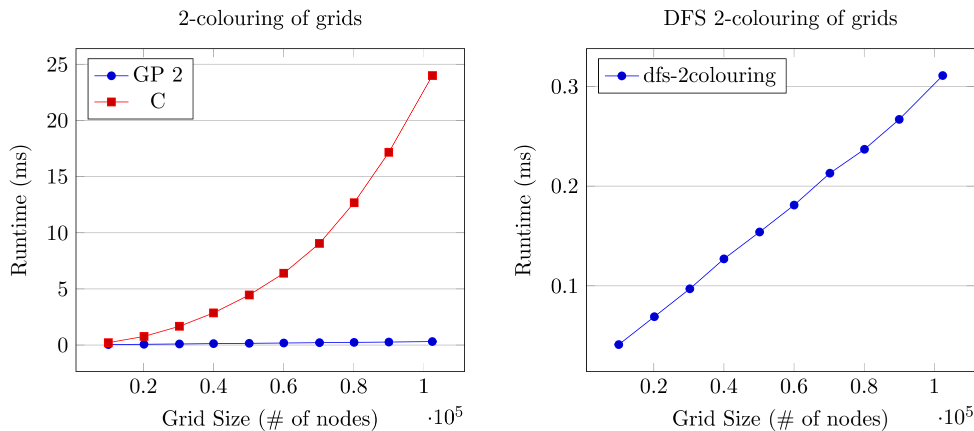


Figure 6.13.: Plots of the runtimes of the rooted 2-colouring programs on grids

`bfs-2colouring` only matches (isolated) unrooted nodes for a subset of its rule set. The take home point is that rooted rules can boost performance speed even if there exist unrooted (or non-fast) rules in the program.

To test the theoretical complexity on graphs of unbounded degree, we ran the 2-colouring programs on `star graphs`. A star graph consists of a central node with k outgoing edges. The targets of these outgoing edges themselves have a single outgoing edge. The test graphs are star graphs ranging from 10^4 to 10^5 edges in increments of 10^4.

Table 6.2 and Figure 6.14 show the runtimes and their plots. None of the curves are linear, but the two rooted programs again substantially outperform the unrooted program. These results match the theoretical expectations that both versions of rooted 2-colouring run in quadratic time on host graphs of unbounded node degree. However, `dfs-colouring` outperforms `bfs-2colouring` by about a factor of 2.

Star Size	`2colouring`	`bfs-2colouring`	`dfs-2colouring`
$0.1 \cdot 10^5$	2.342	0.599	0.276
$0.2 \cdot 10^5$	9.362	2.367	1.035
$0.3 \cdot 10^5$	20.899	5.263	2.307
$0.4 \cdot 10^5$	37.438	9.292	4.081
$0.5 \cdot 10^5$	59.929	14.804	6.61
$0.6 \cdot 10^5$	91.141	21.678	9.91
$0.7 \cdot 10^5$	128.285	30.263	13.874
$0.8 \cdot 10^5$	169.603	40.915	18.651
$0.9 \cdot 10^5$	216.78	53.338	24.094
$1 \cdot 10^5$	268.12	67.375	32.41

Table 6.2.: Experimental results of three 2-colouring programs. Star size is given by the number of edges, and runtime is given in seconds

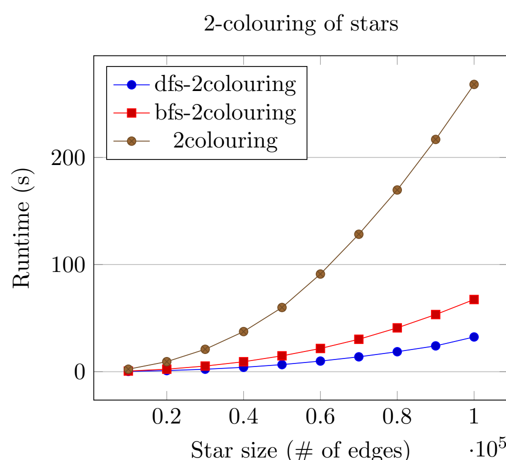


Figure 6.14.: Plot of the runtimes of the rooted 2-colouring programs on stars

6.4. Case Study: Topological Sorting

Topological sorting is a common graph algorithm. It has applications any scenario where a set of tasks or jobs needs to be ordered with respect to dependencies between the jobs such as scheduling tasks in a distributed system or computing the dependencies of the build system of a large software project. The standard specification is as follows: given a directed acyclic graph G, return a list of the nodes such that for all edges e, $s_G(e)$ occurs before $t_G(e)$ in the output list. We refer to this list of nodes as a *topological order*. In the context of graph programs, we will use a different output convention: assign a positive integer to each node such that for each edge $e, \texttt{number}(s_G(e)) < \texttt{number}(t_G(e))$. These integers are the positions of the nodes in the topological order.

There are two established linear-time algorithms for computing a topological sorting of a directed acyclic graph. The first algorithm works by choosing nodes from the graph in their topological order by always selecting nodes with no incoming edges and deleting the outgoing edges of selected nodes [Kah62]. The second is a depth-first traversal that computes the reverse postorder, which is a

```
1: Function topsort(G: graph)
2: L ← ∅
3: Queue Q ← ∅
4: for each n ∈ G where indeg(n) = 0 do
5:     enqueue(Q, n)
6: end for
7: while Q ≠ ∅ do
8:     Node n = dequeue(Q)
9:     append(L, n)
10:    for each outedge e of n do
11:        if indeg(t(e)) = 0 then
12:            enqueue(Q, n)
13:        removeEdge(G, e)
14:    end for
15: end while
16: return L
```

Figure 6.15.: The function `topsort`

topological ordering of the graph [THCRS09].

6.4.1. Standard Sorting

The pseudocode for the first algorithm is presented in Figure 6.15. A queue is used to store all nodes of indegree 0. When a node is removed from the queue, it is appended to the output list, its outedges are deleted, and new nodes with no incoming edges are added to the queue. The use of a queue gives the algorithm the flavour of BFS: all nodes at distance k from a source node are considered before the first node at distance $k+1$. In comparison to the single-sourced BFS algorithms discussed earlier in the chapter, this algorithm performs a "parallel" BFS starting from all the nodes in G with no incoming edges.

This algorithm is destructive: it removes edges in order to find the ordering as quickly as possible. At first glance, the translation to GP 2 appears straightforward. We have already encountered reduction-based GP 2 programs in subsection 5.8.3 and subsection 5.8.4. However, the GP 2 specification for topological sorting requires the preservation of the host graph. Wrapping the computation in a branching statement will not suffice because we wish to keep some of the graph changes, namely the node labels. We therefore simulate edge removal by dashing edges, representing the "non-dashed indegree" of a node by an integer prepended to its label. This value is replaced by the node's position in the topological ordering when all its inedges have been dashed. The GP 2 program is shown with an example execution in Figures 6.16 and 6.17.

In the example, the upper-left graph is the host graph. The node labels are strings; quotation marks are omitted for clarity. We use these labels to uniquely identify the nodes. There are three topological sortings for this graph: $ABCD$, $ACBD$, and $BACD$. The program can generate two of these sortings due to nondeterminism, and it cannot label the nodes in any other way. The second

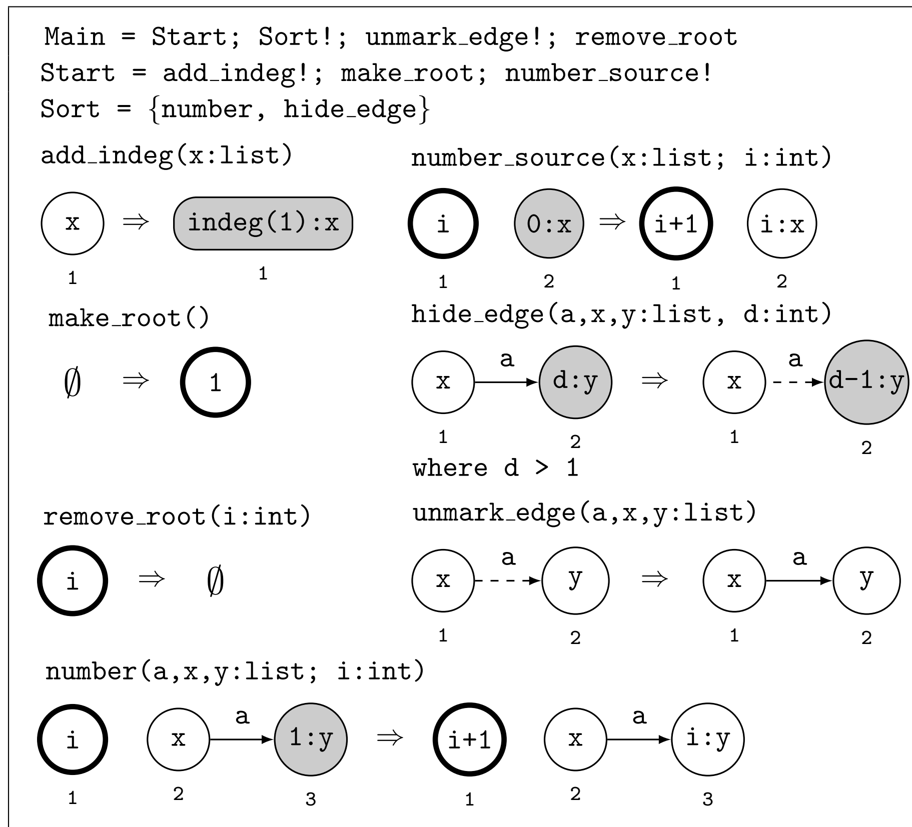


Figure 6.16.: The program `topsort`

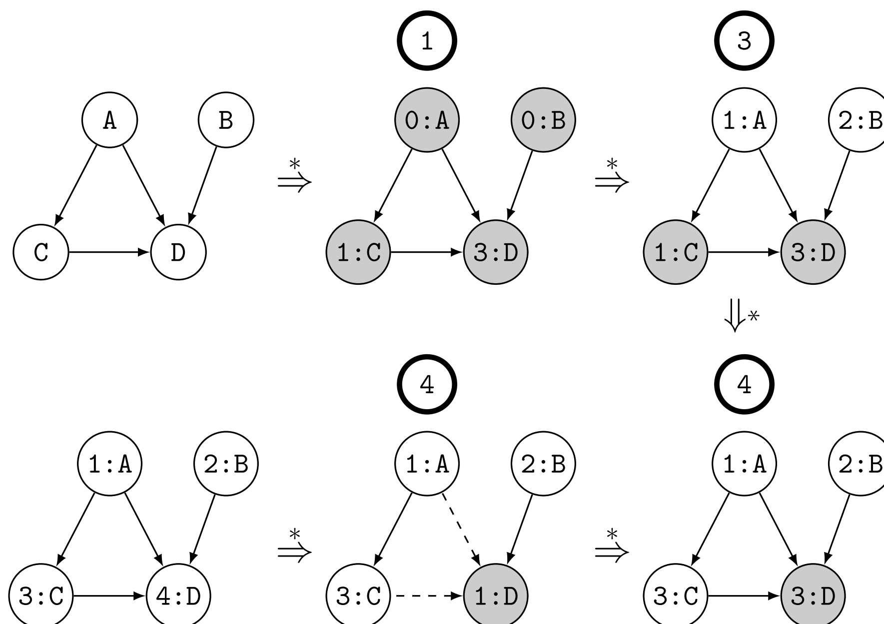


Figure 6.17.: Example run of `topsort`

sorting is not computable by the program because the nodes with no incoming edges (A and B) are always labelled before any other nodes. Therefore the program is sound but not complete.

The procedure `Start` is a preprocessing phase. The loop `add_indeg!` shades each node and prepends their indegrees to their labels. The rule `make_root` creates a root node with the label 1. This root acts as a global variable, storing the next unassigned integer in the topological ordering. The top middle graph is the state after `make_root` is applied. The loop `number_source!` starts the topological ordering by nondeterministically appending integers to labels of nodes with indegree zero and unshading them. The root node's label is incremented after each assignment to ensure each node is assigned a unique integer. The upper-right graph is the state after the completion of `Start`.

Once the zero-indegree nodes have been assigned integers, the program enters the `Sort` procedure, which sorts the rest of the host graph by nondeterministically applying `number` and `hide_edge` until neither can be applied. Applications of `hide_edge` effectively remove an edge from an ordered node to a marked (yet to be ordered) node. It is correct to assign a shaded node the next number in the topological order if there are no incoming marked edges. The rule `number` makes this assignment one step earlier: if the first element of a shaded node's label is 1, then there is no need to dash its remaining incoming edge; it suffices to unshade and label the node while keeping the matched edge unmarked.

The edge $A \to C$ of the top right graph is matched by `number` to give the bottom right graph. Note that $A \to D$ could not be matched by the same rule. This would be incorrect behaviour since it would cause D to be ordered before one of its incoming nodes. The rule `number` has no matches in the bottom right graph. The rule `hide_edge` is applied twice on edges $A \to D$ and $C \to D$, at which point `number` is applicable on edge $B \to D$, assigning the final position in the ordering to node D. No shaded nodes exist in the current graph, so neither `number` nor `hide_edge` are applicable. The program performs a cleanup phase `unmark_edge!; remove_root` to produce the bottom left graph: the host graph with its topological ordering.

Proposition 11. For a node v, define its *value*, denoted by $val(v)$, to be the first atom in its label. Given an unrooted acyclic graph G, the program `topsort` returns G with its nodes relabelled such that for each edge $e, val(s_G(e)) < val(t_G(e))$.

Proof. The proof consists of three parts. We show that the following properties are invariant of the loop `Sort!`: (1) The root node's value is greater than the value of any other unmarked node. (2) For all edges e connecting two unmarked nodes, $val(s_G(e)) < val(t_G(e))$. Finally, we prove: (3) After `Sort!` terminates, all nodes in the graph are unmarked. Together, these results demonstrate the desired behaviour of the program.

1. The root is created with initial value 1 by `make_root`. The loop `add_indeg!` marks all nodes. Each application of `number_source` unmarks a node v and

updates node values in the following way: $val(v) \leftarrow val(root), val(root) \leftarrow val(root)+1$. Therefore, at the start of `Sort!`, the invariant holds. The rule `hide_edge` preserves the invariant because it does not unmark any nodes and it does not modify any values. The rule `number` also preserves the invariant as it performs the same computations with respect to marking and updating node values as `number_source`.

2. When `Sort!` is first entered, the invariant holds because there are no edges with an unmarked target: up to this point, only nodes with no incoming edges are unmarked. The rule `hide_edge` preserves the invariant because it creates no new edges between unmarked nodes. The rule `number` also preserves the invariant. It matches an edge $e : v \to w$ where v is unmarked and w is marked. w is unmarked by the rule, so e is now subject to the invariant. The invariant is not violated because w is assigned the root's value, which is greater than the value of any other unmarked node by (1).

3. Assume that after `Sort!` terminates there exists a marked node v. v has at least one incoming edge, or it would have been unmarked by `number_source`. Let $w_1, \ldots, w_k$ be the sources of edges with target v. At least one of these nodes, say w_1, is marked, otherwise $k-1$ applications of `hide_edge` and 1 application of `number` would have unmarked v. We can apply the same argument to w_1 to infer that at least one of its incoming nodes is shaded. Continuing in this way, since G is acyclic, we conclude that there exists a node w' with indegree 0 that is marked after `Sort!` terminates. This is a contradiction because all indegree 0 nodes are unmarked by `number_source`. Therefore there does not exist a marked node after `Sort!` terminates.

□

The complexity of `topsort` is at least quadratic in the size of the host graph. The program contains rooted rules, but the root node is disconnected from the rest of the graph at all times. The looped rules are applied a linear number of times in the size of the host graph. None of the these rules are fast rule schemata, but they have no more than one edge, so their complexity is linear for host graphs with bounded degree and quadratic otherwise.

6.4.2. Depth-first Sorting

The algorithm specified in Figure 6.18 is a recursive depth-first traversal that explicitly produces a reverse postorder of its nodes by adding nodes to the head of the output list when they are finished.

Using the techniques discussed so far, this algorithm is comparatively easier to translate to GP 2 than the other sorting algorithm. We already have a GP 2 program `dfs` to label each node with its reverse postorder position. However, this would not be sufficient for the stated specification. The resulting node labels would satisfy $l_G(s_G(e)) > m_G(t_G(e))$ for each edge e, the inverse of the desired

condition. Our solution in Figure 6.19 performs two depth-first searches. The first pass traverses the graph to count the nodes. The node count N is stored in an isolated root node. The second pass relabels each node with $N - postorder(n) + 1$. This "inverts" the postorder, so that the output graph is labelled with the correct topological sort as defined by the specification. An example run is shown in Figure 6.20.

The procedure `DFS` traverses the graph and counts the nodes with a root node in the same way as `dfs`. `DFS` is executed for each indegree 0 node in the host graph. These nodes cannot be reached from one another, even if they exist in the same connected component. Each visited node is coloured red; the second DFS phase conducted by `Sort!` is responsible for unmarking the nodes. Before each DFS terminates, the indegree 0 node is unrooted so that at most one root node (excluding the counter) is maintained at all times. The top left graph of Figure 6.20 is the graph after `make_root` is applied. The top middle graph is the graph after `DFS!` terminates.

The top right graph is the state after one application of `init_sort` and two applications of `sort_forward`. The grey root node stores the list $n : m$, where n is the number of nodes in the host graph. m, currently 0, acts as a counter during the second DFS conducted by `Sort!`. When a node is finished, it is assigned the value $n - m$, and m is incremented. As the bottom right graph demonstrates, the first finished node is assigned the greatest integer in the ordering by `sort_back`, namely $n - 0 = n$. As m increases, the assigned integers decrease. Two further applications of `sort_back` and an application of `unroot` give the bottom middle graph. An application of `init_sort` and one more application of `unroot` on B give the bottom left graph, labelled with a valid topological sorting of the host graph. Observe that the postorder of the DFS is $DCAB$, and the topological ordering computed by the program is $BACD$, the reverse of this postoder.

Proposition 12. For a node v, define its *value*, denoted by $val(v)$, to be the first atom in its label. Given an unrooted acyclic graph G, the program `topsort` returns G with its nodes relabelled such that for each edge e, $val(s(e)) < val(t(e))$.

1: **Function topsort-DFS***(G: graph)*
2: **while** there exist unmarked nodes **do**
3: choose an unmarked node n
4: explore(n)
5: **end while**
6:
7: **Procedure explore***(n: node)*
8: mark n
9: **for** each outedge e of n **do**
10: explore($t(e)$)
11: **end for**
12: prepend(L, n)

Figure 6.18.: A function to compute topological sorting using DFS

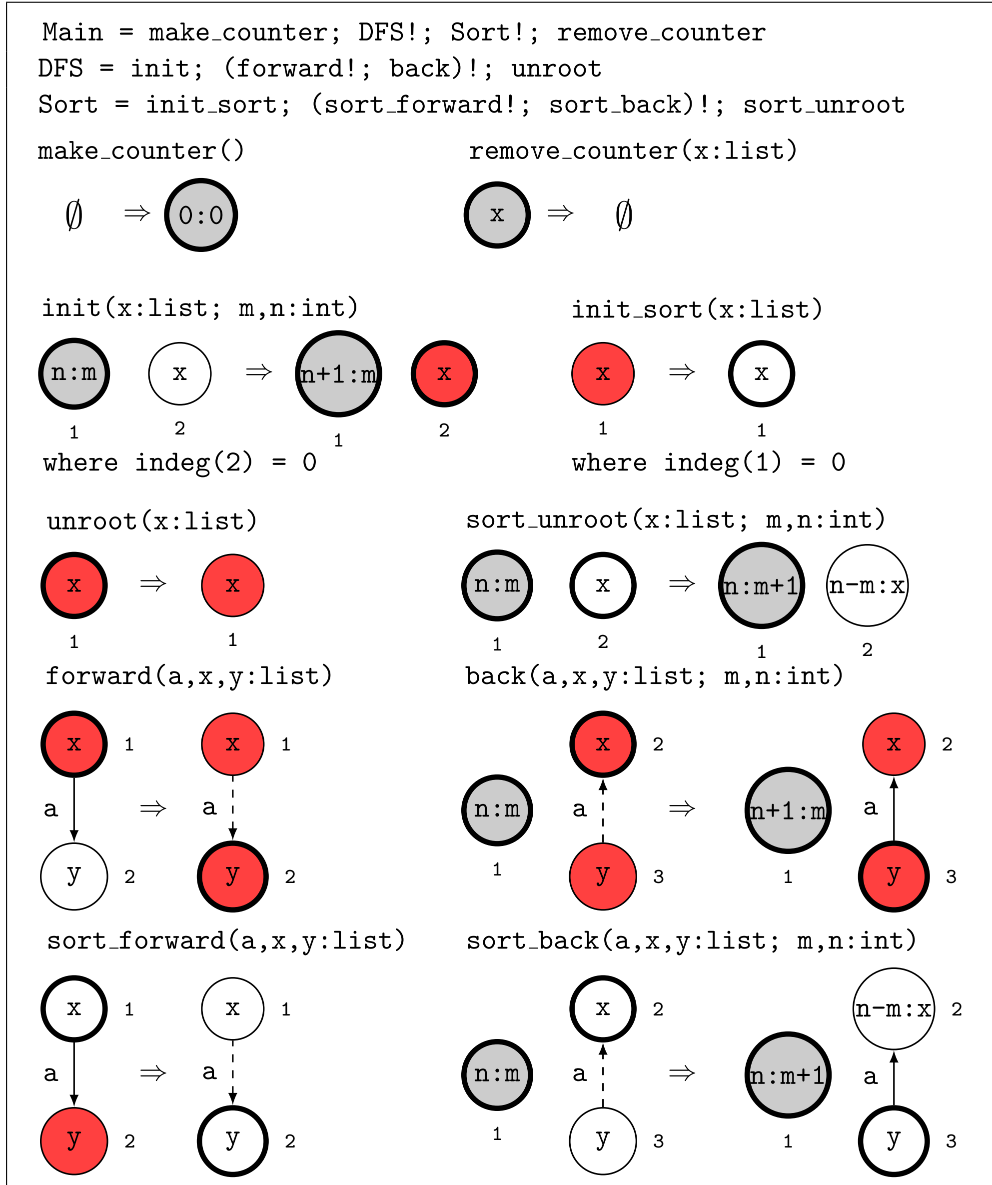


Figure 6.19.: The program `dfs-topsort`

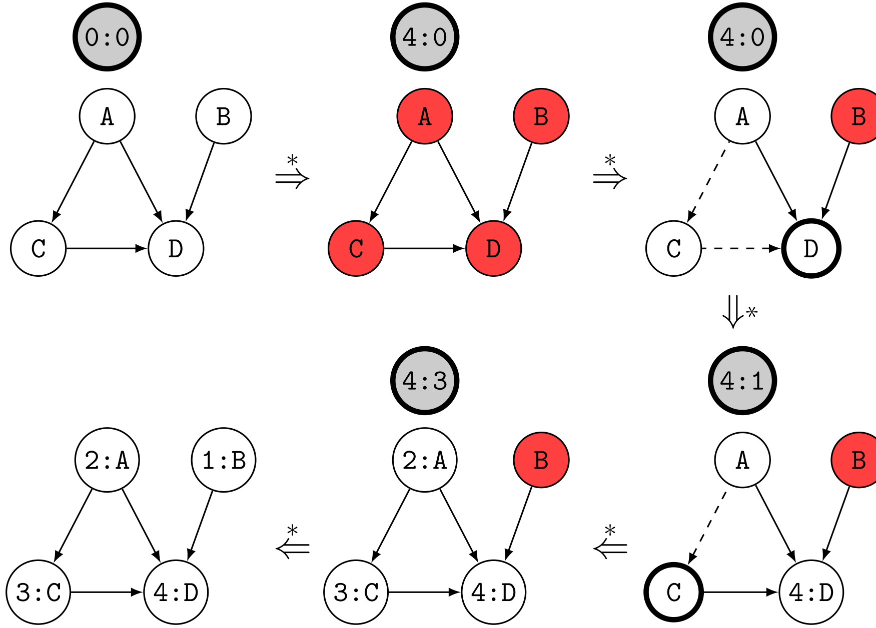


Figure 6.20.: Example run of program `dfs-topsort`

Proof. Let $V = \{v_1, \dots, v_k\}$ be the set of nodes in G with indegree 0. By Lemma 6, each iteration of `DFS!` visits some v_i and all nodes directly reachable from it. Moreover, all of these nodes are marked red, and v_i is unrooted by `unroot`. Since `DFS` marks an indegree 0 node, and `init` matches an unmarked node with indegree 0, `DFS` is applied once for each indegree 0 node in G. Therefore all nodes in the graph are visited after termination of `DFS!`. Moreover, they are red and unrooted.

The grey root node stores the list $|V_G| : 0$. These two numbers are used to assign the positions in the topological ordering to the label of each node during the second depth-first traversal conducted by `Sort!`. Each iteration of `Sort!` starts its depth-first traversals from a red node with indegree 0. The traversal moves through the red nodes of the working graph, unmarking a node when it is visited. When a node is finished, it is assigned the value $n - m$, where n and m are taken from the grey root's label $n : m$. As aforementioned, $n = |V_G|$ remains fixed. Another application of Lemma 6 means that all nodes are visited in this phase. m stores the number of finished nodes: it is initialised at 0, and it is incremented whenever a node is finished, recognised by an application of `sort_back` or `unroot`. For any edge $v \to w$, we have $val(v) = |V_G| - m', val(w) = |V_G| - m''$, and $m' > m''$. The third inequality is true because a node is always finished before any of its incoming nodes. It follows that $val(v) < val(w)$. □

Proposition 13. The program `gp2-dfs-topsort` runs in quadratic time on host graphs of bounded degree, and cubic time otherwise.

Proof. The complexity of the program is determined by the complexity of the procedures `DFS` and `Sort`. Both procedures traverse the graph in the same way as `dfs`. We cannot use the result of Proposition 4 directly, as these searches could

be invoked multiple times. Instead, we consider the complexity of the entire loops `DFS!` and `Sort!`. It is clear that both procedures have the same structure, so we only need to analyse one of these loops to establish the complexity of the whole program. Without loss of generality, consider `DFS!`.

Case 1. *G has bounded degree.* First note that `DFS` iterates at most $|V_G|$ times, since each iteration marks at least one node with `init`. Each iteration performs a DFS sourced at an indegree-0 node v. This node is found by the rule `init` which is not a fast rule schema, and so takes linear time to find the single unrooted node in the left-hand side. In the worst case, G has no edges, causing `init` to be matched $|V_G|$ times, giving an overall quadratic complexity.

Case 2. *G has unbounded degree.* Consider a single iteration of `DFS!`. By Proposition 6, the loop iteration visits all nodes reachable from the node marked by `init`. Note that when a node is visited, it cannot be matched by `forward` or by `back` in this iteration or any future iteration because visited nodes are marked red. Therefore, both `forward` and `back` are applied a linear number of times each in the entire loop `DFS!`. The complexity of matching these rules is quadratic, giving an overall cubic complexity. □

6.4.3. Experimental Results

Both topological sorting programs were executed on forests. The test graphs are generated by a GP 2 program, which takes as input a single node labelled with the number of nodes in the desired tree. The program creates three isolated root nodes, then nondeterministically grows branches from leaf nodes until the node size is reached. Each rule extends a leaf node with either one, two, or three incident edges. The structure of the program gives a 50% weighting to two branches, and 25% weight to one branch and to three branches. Some examples are shown below. We run the experiments using two types of forests: *outgrowing*, in which all edges point away from the root nodes; and *ingrowing*, in which all edges point towards the root nodes. The number of indegree-0 nodes in outgrowing forests are fixed, while the number of indegree-0 nodes in ingrowing forests increases as the size of the forest increases.

Figure 6.21 shows two outputs of the forest generator when instructed to build a forest containing 15 nodes. Figure 6.22 shows the output of both topological sorting programs when executed on the left forest. The different traversal strategies of the algorithms are clearly visible in the node labellings of the graphs.

The running times of the two topological sorting programs on forests ranging from 10^4 nodes to 10^5 nodes are presented in Table 6.3 and Figure 6.23. The left plot shows that the rooted topological sorting program clearly outperforms the standard program for both classes of forest. A closer inspection of the time growth of the rooted programs shows that `dfs-rooted` exhibits linear performance on outgrowing trees, but not on ingrowing trees. The linearity in particular is made clear in the right plot. The ingrowing trees add a degree of complexity because the number of indegree 0 nodes is unbounded. In the rooted programs, these are searched for (by `init` and `init_sort`) without the benefit of rootedness. Even

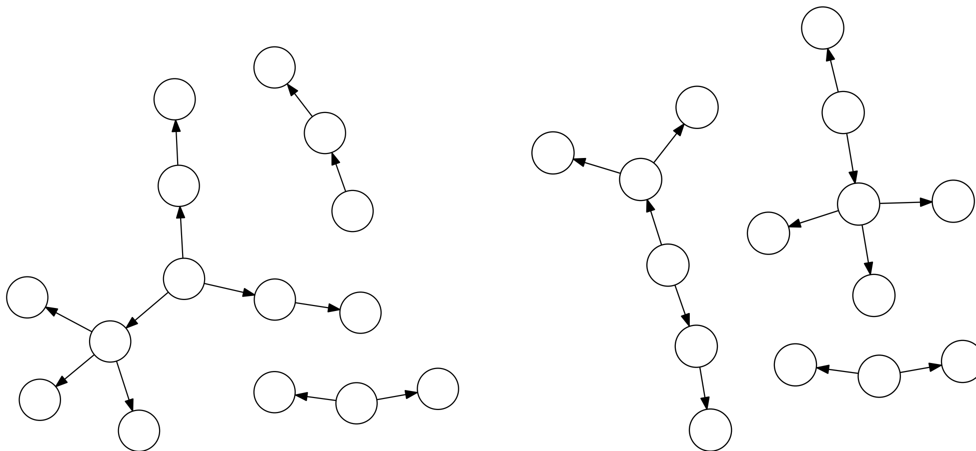


Figure 6.21.: Two outgrowing forests of 15 nodes generated by GP 2

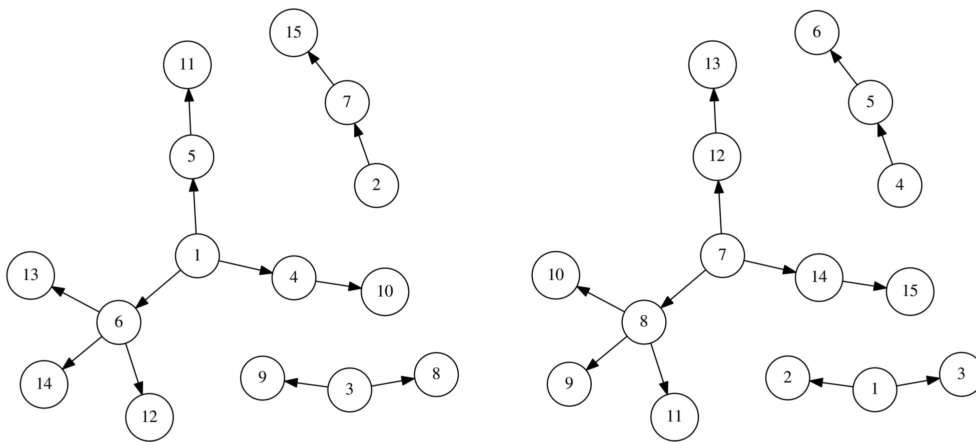


Figure 6.22.: Topological sortings of a forest. The sorting on the left was produced by `topsort`. The sorting on the right was produced by `dfs-topsort`

	Outgrowing		Ingrowing	
Forest Size	`topsort`	`dfs-topsort`	`topsort`	`dfs-topsort`
$0.1 \cdot 10^5$	2.469	0.052	5.13	0.884
$0.2 \cdot 10^5$	9.729	0.089	20.673	3.394
$0.3 \cdot 10^5$	22.093	0.128	48.022	7.625
$0.4 \cdot 10^5$	39.622	0.177	90.653	13.363
$0.5 \cdot 10^5$	62.205	0.212	151.009	21.025
$0.6 \cdot 10^5$	91.047	0.261	225.149	30.531
$0.7 \cdot 10^5$	126.087	0.304	321.1	42.075
$0.8 \cdot 10^5$	165.586	0.346	435.599	57.561
$0.9 \cdot 10^5$	221.033	0.387	574.812	76.096
$1 \cdot 10^5$	261.53	0.426	742.715	97.211

Table 6.3.: Experimental results of two topological sorting programs. Forest size is given by the number of nodes, and runtime is given in seconds.

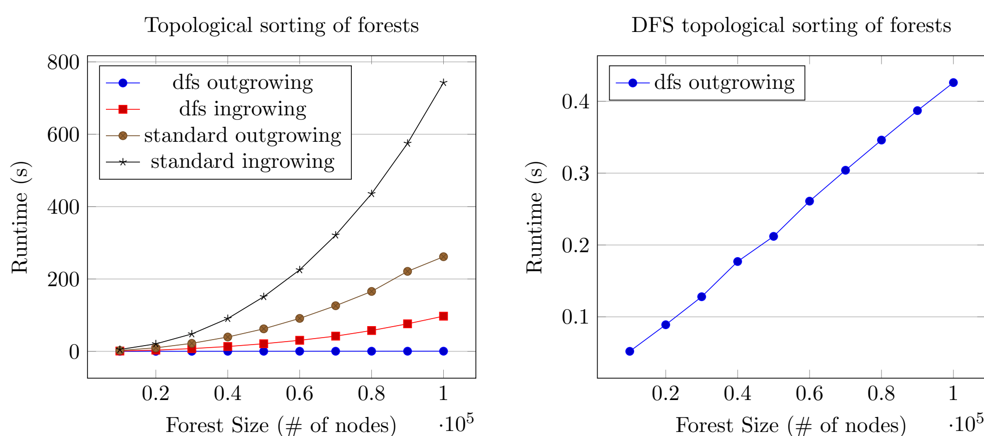


Figure 6.23.: Plots of the runtimes of the topological sorting programs on stars

with this in mind, it is striking that the rooted program's performance slows down by two orders of magnitude when the input changes from outgrowing forests to ingrowing forests.

6.5. Comparison with C Programs

Generally speaking, we cannot expect GP 2 programs, even rooted ones, to compete with a low-level implementation tailored to solve the problem at hand. However, some of the experimental results of the previous sections demonstrate impressive runtimes. For certain classes of test graphs, the rooted GP 2 programs were able to process graphs containing tens of thousands of nodes in well under a second. It would be interesting to see how these cases perform in comparison to a "bespoke" implementation. This section describes C implementations of 2-colouring and topological sorting, based on the code in Sedgewick's *Algorithms in C* [Sed02] and provides experimental results.

The goal is to write the most efficient C code to solve the two graph algorithms.

Hence the choice of graph data structure is an important one to make. Broadly speaking, the two most common representations of directed simple graphs (graphs without loops or parallel edges) are the *adjacency matrix* and the *adjacency list structure*. For a graph with n nodes, an adjacency matrix is an $n \times n$ array M of 0's and 1's (we avoid discussing implementation details for the moment). An edge $i \to j$ is represented by $M_{i,j} = 1$, additionally $M_{j,i} = 1$ if the graph is undirected. For the same graph, an adjacency list is a node-indexed array containing n linked lists. An edge $i \to j$ is represented by the presence of j in the ith linked list, and vice versa if the graph is undirected.

The best solution often depends on the expected host graphs and on the problem that is being solved, and this is no exception. The host graphs on which we will test these programs — grids and stars — are sparse graphs. Intuitively, a sparse graph is one in which the number of edges is linear in the number of nodes. In contrast, the number of edges is quadratic in the number of nodes for dense graphs (complete graphs being the most extreme example). Adjacency lists are preferable for sparse graphs because an adjacency matrix has N^2 space complexity[1], a quantity independent of the number of edges, while adjacency lists have space complexity $N + E$. Furthermore, adjacency matrices are ideal for applications in which one wishes to make edge-based queries: checking the presence of an edge between two nodes is a constant time operation. This is not appropriate in our case, where we will be using DFS-based algorithms. There is no significant theoretical difference in runtime performance for our use cases, so based on the space complexity, we choose to use the adjacency list.

For our purposes there is no requirement to implement a graph data structure that supports all of GP 2's features. Instead, we exploit some of the properties of the algorithms and host graphs we wish to execute in order to develop a minimal graph data structure. Specifically:

- The algorithms do not modify the graph structure, so we do not concern ourselves with dynamic memory allocation.
- We do not need to support GP 2 lists. Although the GP 2 programs work on host graphs with any node or edge labels, the tests are conducted on blank graphs. Therefore we only need to support the minimum labelling required to perform the computation.
- No explicit representation of outgoing edges and incoming edges. The graph traversals we require are either undirected or along outgoing edges only.

The presented C code is adapted from the code in [Sed02]. We preserve Sedgewick's function names, but we change some variable names to assist in readability. Figure 6.24 shows the adjacency-list structure. The Graph structure stores counts of the number of nodes and edges, an array of Link pointers *adj*, and an array of integer node labels *label*. Nodes are represented by linked lists, where each list element stores the node identifier (index into *adj*) of a target of

[1] Assuming a lack of labels or a constant-space label representation

```c
typedef struct Node {
    int node_id;
    struct Node *next;
} Link;

typedef struct Graph {
   int nodes; // The number of nodes in the graph.
   int edges; // The number of edges in the graph.
   Link **adj;
   int *label; // Node-indexed array of labels.
} Graph;
```

Figure 6.24.: Adjacency-list graph representation in C

```c
Graph *GRAPHinit(int nodes) {
    Graph *graph = malloc(sizeof(*graph));
    graph->nodes = nodes;
    graph->edges = 0;
    graph->adj = calloc(nodes, sizeof(Link*));
    graph->label = calloc(nodes, sizeof(int));
    return graph;
}

Link *NEW(int id, Link *next) {
    Link *link = malloc(sizeof(*link));
    link->node_id = id;
    link->next = next;
    return link;
}

void GRAPHinsertE(Graph *G, int src, int tgt, bool directed) {
    G->adj[src] = NEW(tgt, G->adj[src]);
    if(!directed) G->adj[tgt] = NEW(src, G->adj[tgt]);
    G->edges++;
}
```

Figure 6.25.: Functions to build the graph.

one of its outgoing edges. Edge labels are not required for the algorithms. As we shall see, it suffices to store a single integer for each node label to implement 2-colouring and topological sorting algorithms on blank host graphs.

Figure 6.25 shows the functions that build the graph data structure. We omit code to error-check pointers after they have been allocated memory to reduce clutter. It is assumed that the node size N is known in advance, an assumption we can meet when inputting the host graph. This is passed to `GRAPHinit`, which allocates memory for the graph structure itself, N pointers for the `adj` array, and N integers for the `label` array. In the latter two cases, `calloc` is used so that the allocated memory is set to 0, avoiding an explicit linear-cost initialisation process. `NEW` allocates a new Link, sets its *id* to its first argument and prepends the new Link to its second argument. `GRAPHinsertE` uses `NEW` to add the edge's target *id* to the source's list. If the graph is undirected, then the edge's source

id is added to the target's list.

At runtime, the GP 2 compiler's host graph parser is used to read the host graph text file and construct the graph data structure. This is done in order to minimise the gap between the handwritten C code and the code generated from the GP 2 compiler, so that the comparison between the performance of the actual computations on the host graph is as fair as possible.

We now have the tools in place to write the C algorithms for 2-colouring and topological sorting. They are given in Figure 6.26 and Figure 6.27 respectively. Some code is omitted, including code for error checking and host graph building, and the declaration of global variables. The graph is undirected for 2-colouring since edge direction is not considered when colouring a graph. Conversely, the graph is directed for topological sorting because the algorithm requires a directed graph. A global flag `directed` is set to false for 2-colouring, and true for topological sorting. Both programs take a single command line argument: the file path of the host graph. The function `buildHostGraph` initialises and adds edges to the graph (via the global Graph pointer *host*) through the GP 2 host graph parser. `GRAPHinsertE` is called with the global flag *directed*, so it adds edges appropriately depending on the nature of the graph.

In 2-colouring, nodes are labelled 0, 1 or 2. Node labels are initialised to 0, representing an uncoloured and unvisited node. 1 and 2 represent the two colours with which the host graph is coloured. The function `dfsColour` is called recursively on all uncoloured nodes of the host graph. It is passed a node v and a colour c as its argument. It colours v with the contrasting colour c', and goes through v's adjacency list. If an adjacent node is uncoloured, `dfsColour` is called on that node. If an adjacent node is also coloured c', the function returns false, which will propagate through its parent calls and to the main function. If main detects a failure (line 23), it sets the label of all nodes to 0 and exits. Otherwise, the coloured graph is returned.

Like `dfs-topsort`, the topological sorting algorithm conducts a DFS and labels nodes with the inverse of their postorder positions. Unlike the GP 2 program, the number of nodes N is known in advance, so this can be achieved in a single graph traversal. The same recursive structure is used, but there are some notable differences elsewhere. Two additional global variables are maintained. First, an array *visited* that records the visited status of each node. The node labels will not suffice for this as they did in the 2-colouring algorithm because nodes are only relabelled when they are finished. Second, a count of the postorder *po*. When a node is finished, it is assigned $N - po$, and then *po* is incremented. Thus, each node is assigned a distinct integer $1 \leq k \leq N$.

We ran the handcrafted C programs against the GP 2 programs `dfs-2colouring` and `dfs-topsort`. The results for 2-colouring are given in Table 6.4 and Figure 6.28, and for topological sorting in Table 6.5 and Figure 6.29. There is little difference between the time it takes for either program to 2-colour grids. However, the star graph plot makes it clear that tailored C code is not limited by bounds on node degree because it is not required to perform an explicit search from scratch to explore each edge. GP 2's depth-first topological sorting

```
bool dfsColour(int node, int colour) {
   Link *l = NULL;
   int new_colour = colour == 1 ? 2 : 1;
   host->label[node] = new_colour;
   for(l = host->adj[node]; l != NULL; l = l->next)
      if(host->label[l->id] == 0)
      {
         if(!dfsColour(l->id, new_colour)) return false;
      }
      else if(host->label[l->id] != colour) return false;
   return true;
}

int main(int argc, char **argv) {
   host = buildHostGraph(argv[1]);
   bool colourable = true;
   int v;
   for(v = 0; v < host->nodes; v++)
      if(host->label[v] == 0)
         if(!dfsColour(v, 1)) { colourable = false; break; }
   if(!colourable)
      // Reset the host graph by unmarking all its nodes.
      for(v = 0; v < host->nodes; v++) host->label[v] = 0;
   return 0;
}
```

Figure 6.26.: DFS 2-colouring in C

```
static int *visited = NULL, postorder = 0;

void dfsSort(int node) {
   Link *l = NULL;
   visited[node] = 1;
   for(l = host->adj[node]; l! = NULL; l = l->next)
      if(visited[l->id] == 0) dfsSort(l->id);
   host->label[node] = host->nodes - postorder;
   postorder++;
}

int main(int argc, char **argv) {
   host = buildHostGraph(argv[1]);
   visited = calloc(host->nodes, sizeof(int));
   int v;
   for(v = 0; v < host->nodes; v++)
      if(visited[v] == 0) dfsSort(v);
   return 0;
}
```

Figure 6.27.: DFS topological sorting in C

Grid Size	GP 2	C
10,000	41	39
20,164	69	66
30,276	97	95
40,000	127	114
50,176	154	152
60,025	181	178
70,225	213	207
80,089	237	239
90,000	267	260
102,400	311	294

Star Size	GP 2	C
$0.1 \cdot 10^5$	281	21
$0.2 \cdot 10^5$	1,051	45
$0.3 \cdot 10^5$	2,318	62
$0.4 \cdot 10^5$	4,228	66
$0.5 \cdot 10^5$	6,781	95
$0.6 \cdot 10^5$	10,349	108
$0.7 \cdot 10^5$	14,590	118
$0.8 \cdot 10^5$	19,499	136
$0.9 \cdot 10^5$	24,941	147
$1 \cdot 10^5$	31,028	163

Table 6.4.: Comparison of `dfs-2colouring` with a C 2-colouring program. Grid size is given by the number of nodes, star size by the number of edges, and runtime in milliseconds.

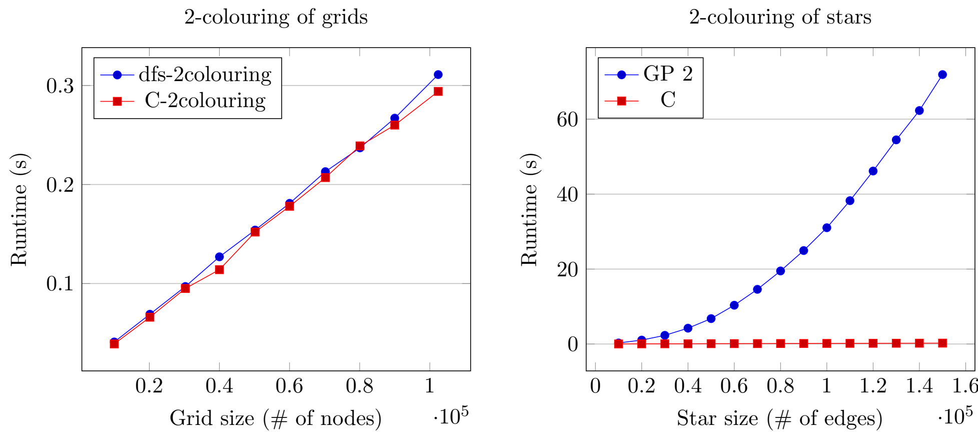


Figure 6.28.: Plots of the runtimes of GP 2 and C 2-colouring programs

program sorts outgrowing forests approximately three times slower than the C program does. Again, we observe that the C program's performance is unconstrained by the class of host graph as it sorts ingrowing trees equally as quickly. The constant factor is partially explained by the fact that the GP 2-generated code performs two depth-first searches in contrast to the single depth-first search executed by the C code. However, that still leaves a constant gap between GP 2 and C, something that was not present in the 2-colouring programs. This is likely because `dfs-topsort` performs frequent relabelling operations which are more computationally demanding than the remarking done by `2-colouring`.

6.6. Summary and Discussion

Rooted graph programs can be used to encode established graph algorithms at a high level of abstraction with the use of rooted graph transformation rules. Using a template for breadth-first search and depth-first search, we have implemented solutions to two common graph algorithms: 2-colouring and topological sorting.

Forest Size	GP 2 (outgoing)	C (outgoing)	C (ingrowing)
$0.1 \cdot 10^5$	0.052	0.019	0.02
$0.2 \cdot 10^5$	0.089	0.036	0.03
$0.3 \cdot 10^5$	0.128	0.05	0.047
$0.4 \cdot 10^5$	0.177	0.059	0.058
$0.5 \cdot 10^5$	0.212	0.07	0.073
$0.6 \cdot 10^5$	0.261	0.084	0.086
$0.7 \cdot 10^5$	0.304	0.099	0.102
$0.8 \cdot 10^5$	0.346	0.116	0.109
$0.9 \cdot 10^5$	0.387	0.121	0.123
$1 \cdot 10^5$	0.426	0.141	0.131

Table 6.5.: Comparison of `dfs-topsort` with a C topological sorting program. Forest size is given by the number of nodes. Runtime is given in milliseconds.

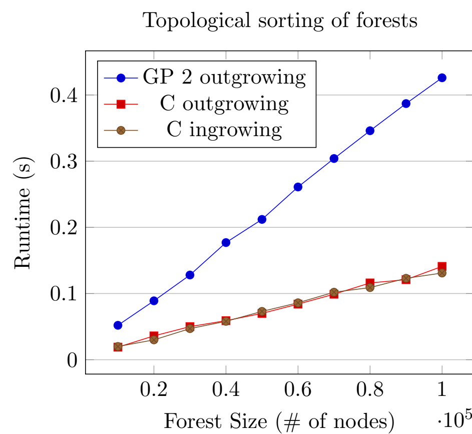


Figure 6.29.: Plots of the runtimes of GP 2 and C topological sorting programs

For some graph classes, these solutions display performance in the same complexity class as hand-coded C solutions, a remarkable feat considering the high level of abstraction of GP 2 programs, the "template" C code that is generated from them, and the lack of any auxiliary data structures in the tailored C code.

The limitations of performing global computations on graphs through graph transformation rules, even rooted ones, mean that linear time graph algorithms cannot always be achieved. Host graphs of unbounded degree remove the constant time matching of rooted rules: even when the root node is matched instantly, an unbounded number of outedges causes a linear overhead in the matching of an edge incident to a root node in the rule.

Another drawback of rooted graphs and rooted rules compared with unrooted graphs and rules is greater complexity in writing graph programs. This is most evident when inspecting the 2-colouring programs: the unrooted program has five rules, while the rooted programs have an average of nine rules and more sophisticated control constructs. Indeed, the unrooted program is purely declarative, while the rooted programs takes a step towards imperative programming

by explicitly encoding the traversal strategy. However, we argue that the programs are instructive, easy to understand, build on the rooted graph traversal templates described at the start of the chapter, and provide a cleaner, simpler and more accessible way of writing graph algorithms than coding them directly in a lower-level language such as C.

7. Conclusion

7.1. Evaluation

To evaluate the work presented in this thesis, we refer to the broad question asked in the introduction: *How close can a high-level graph programming language come to the performance of graph programs written at a much lower level of abstraction?* We addressed the problem by implementing a compiler for the high-level graph programming language GP 2 and testing the performance of its output — a C program — against pure C implementations of graph algorithms.

We introduced rooted graph transformation to tackle the high theoretical complexity of rule matching, a huge problem for the practical execution of graph transformation. Rooted graph transformation extends the established double-pushout approach to support the direct binding of specific rule nodes to dedicated host nodes. We demonstrated its practicality by adding rooted graph transformation to the GP 2 language, although its abstract definition makes it possible for other languages and tools to adopt the approach.

The implementation itself was a success, though not one without challenges. Some of GP 2's language features map nicely to C, such as labels and control constructs. Other features, however, required more sophistication in the code generation phase. The most complex part of the translation step was generating code to apply a high-level graph transformation rule. We achieved this by storing rules internally as a complex data structure that captures features of a particular rule such as how the rule modifies individual items and the presence of nodes and variables in the application condition. This facilitates the generation of code that matches rules using a searchplan-based algorithm interleaved with code to match labels and to evaluate the condition. Another subtlety of implementing GP 2 is the requirement to support the recovery of an old host graph state, which we implemented by maintaining a stack of host graph changes after an empirical comparison with copying the whole host graph to memory.

Overall, we demonstrated that it is possible, in some circumstances, for a compiled GP 2 program to match the runtime of a handcrafted C program that performs the same computation. We consider this to be a strong result: the compiled C code performs explicit subgraph matching and rule application, while the tailored C code recurses over a basic graph data structure. This was achieved by a novel extension to the theory developed to optimise graph matching, and its practical realisation within the GP 2 language and its implementation.

We do not wish to embellish these results. Certainly compiled GP 2 code cannot match equivalent lower-level code in all situations. Even rooted graph programs have quite significant limitations. It is not clear if they can be used to

write efficient graph programs beyond algorithms based on depth-first search. In addition, their performance wanes as the node degree of host graphs increases. We believe that, in general, GP 2's performance does not rival the efficiency of the fastest graph transformation tools of today, although this has not been empirically tested. More work needs to be done in searching for further use cases of rooted graph programs, optimising rules without roots, and extending our suite of case studies with an aim to directly compare performance with related tools.

7.2. Future Work

We discuss several paths for future research, broadly categorised into increasing the efficiency of the current GP 2 implementation and adding tool support for users of the language.

7.2.1. Dynamic Rule Matching

Rules can be matched very quickly by utilising root nodes in rules and graphs, a language-level construct. However, no great effort is made to optimise matching on the implementation level, which hinders the performance of matching unrooted rules. GP 2's static searchplan algorithm is rudimentary compared to the state of the art. We emphasise two dynamic approaches: dynamic searchplan generation and incremental matching. The latter in particular has received a lot of attention in recent years within the graph transformation and model transformation communities.

Dynamic searchplan generation aims to find an optimal searchplan at runtime based on an analysis of the host graph. The cost of the searchplan operations are a function of host graph metrics. A greedy algorithm is used to heuristically select the optimal searchplan based on these costs. Incremental matching computes in advance the set of subgraphs of the host graph that match a rule so that matches can quickly be extracted when necessary. In both cases, the stored data relating to the match — the cost of searchplan operations or the occurrences of matches in the host graph — are dynamically updated as the host graph changes.

Both methods invest memory and runtime overhead into speeding up the expensive rule matching operation. Experimental results have generally justified this approach; we present some work from the literature. The dynamic searchplan generation algorithm implemented by GrGEN.NET generates good searchplans based on a cost model that accurately represents their actual execution times [BKG07], although it is unclear how that translates to the global execution time of a graph transformation system with many rule applications. An alternative approach to searchplan generation based on dynamic programming has also demonstrated promising results [Var+12]. Incremental matching has also proven to be quite successful, particularly in the VIATRA2 model transformation framework [Ber+08]. GROOVE has also ventured into this area: results from experiments conducted with the GROOVE tool show that a RETE-based incremental matching algorithm outperforms a searchplan approach in most situations, one

exception being when structure-modifying rules heavily outweigh querying rules or relabelling rules [JGR12]. A hybrid of these two dynamic techniques has been implemented and tested in VIATRA2, in which the programmer can explicitly specify one of the two matching strategies for each rule, with promising results [Hor+10].

The GP 2 runtime library could be extended to support one or both of these approaches. In particular, a hybrid approach similar to that used by VIATRA 2 might be very effective. This could be enhanced by an automatic selection of the matching strategy based on a static analysis of the program text. For instance, rules present in a loop are expected to be applied many times and possibly in different areas of the host graph, making them a good target for an incremental matching strategy. Rules that are matched fewer times might benefit more from a dynamic searchplan matching strategy. Thorough testing needs to be conducted to discover a good selection of matching strategies with respect to the particular GP 2 program, but a clever dynamic matching strategy should significantly speed up the execution time of non-rooted programs, perhaps by orders of magnitude.

7.2.2. Optimising the Current Implementation

Besides adopting a new matching algorithm, we believe that less ambitious but nevertheless useful measures can be taken to enhance the current codebase with respect to both execution speed and memory management.

The existing host graph data structure is not fine-tuned to graph transformation: it does not support complex querying operations that would speed up graph matching with the current static searchplan algorithm. One way to achieve this is to use the hash values of lists as an index for nodes and edges. Marks could also be used as a second indexing structure. Combined, this two-dimensional indexing would support fine-grained host graph querying by label, pruning the search space at runtime. Another approach is to use a third party library to implement the graph data structure. We highlight GP 1's use of Judy arrays [Sil02] to implement a data structure that supports quick and powerful graph queries such as searching for an edge with a specific target node, or querying atoms in a specific position of an item's label [MP08a].

The code generated for rule matching could be fine-tuned to reduce computation effort at runtime. For instance, the current implementation matches an unrooted node by searching the host graph nodes in a fixed order. This is a source of inefficiency for a looped rule that is applied consecutively at different places in the host graph. Recording the state of the last search and passing it on to the next rule application would remove the redundancy caused by searching an already-matched portion of the host graph at each step. This could be taken a step further by searching for all matches of a single rule and applying them in one atomic step, although this requires some care as pairs of matches could be in conflict. The concept is used in PORGY with its `all` operator which allows simultaneous rule application at disjoint matches in the host graph [FKP14]. Parallel rule application has been studied extensively for term rewriting systems

[BKV03]. An interesting area of research is to transfer these results to graph transformation theory and graph programming languages.

7.2.3. Extending the GP 2 Tool Suite

A graphical editor originally developed for a Master's project [Ell13] is currently being integrated with the GP 2 compiler. The editor enables users to write graph programs, execute them on a host graph of their choice, and see the result. At the moment, the tool does not offer the programmer any practical means for testing and debugging of graph programs. With this in mind, the following tool support would be of great benefit to users.

Graph Program Tracing. The compiler currently offers some rudimentary program tracing facilities, namely the printing of rule matching attempts, host graphs after rule applications, and information related to graph backtracking. While this might be sufficient, it is not up to the standards of a graphical programming environment. For example, a graphical tracing facility for graph programs should highlight the match within the host graph at each step, and present variable-value assignments in an easily digestible format. In the end, the tool should provide a full debugging environment that allows users to step through a graph program with different levels of granularity.

Graph Program Verification. Recent theoretical work has established a basis for formal verification of graph programs. Habel, Pennemann and Rensink extend a base form of graph programs to high-level rules with application conditions to facilitate formal reasoning based on Dijkstra's weakest precondition approach [HPR06]. This was implemented as part of the ENFORCe tool (see the paper [Aza+06] or Pennemann's PhD thesis [Pen09]). Poskitt's PhD thesis [Pos13] defines a Hoare logic for reasoning about GP 2 graph programs. GP 2 is a feasible target language for a implementation of a graph program theorem prover due to its small syntax and semantics.

Critical Pair Analysis of Graph Programs. Confluence is a desirable property of any graph transformation implementation: if it can be proven that a graph program is confluent, the global behaviour of the graph program is deterministic despite the inherent nondeterminism of rule application. Plump introduced critical pair analysis for hypergraph rewriting, and proved it to be undecidable [Plu93; Plu05]. From an implementation point of view, this is made more challenging with attributed graphs. Currently, AGG offers the only implemented critical pair generator for graph transformation [RET11]. Preliminary work has been made towards implementing a confluence checker for GP 2, specifically a unification algorithm for GP 2's lists to faciliate construction of critical pairs for conditional rule schemata [HP15].

7.2.4. Development of Larger Graph Programs

GP 2 has quite a few published (and unpublished) graph programs, most of which fall under the category of graph algorithms or recognition of graph classes by reduction. Extending this suite to a broader range of application areas and more complex graph programs (in the vein of automata minimisation [PSS11]) benefits our research in several ways. We seek more classes of graph programs that can reap the benefits of fast rule schemata to further demonstrate the practicality of rooted graph transformation. In addition, larger case studies in the area of software engineering (model transformation is a particularly ripe target for graph transformation [Gru+05]) would allow us to directly compare the performance of the GP 2 system to other graph transformation systems in, for example, transformation tool contests.

Appendices

Appendix A.

GP 2 Concrete Syntax

A.1. Identifiers

ProcedureID	::=	UpperCase {IDChar}
RuleID	::=	ID
NodeID	::=	ID
EdgeID	::=	ID
Variable	::=	ID
ID	::=	LowerCase {IDChar}
UpperCase	::=	`A` | ... | `Z`
LowerCase	::=	`a` | ... | `z`
Letter	::=	UpperCase | LowerCase
Digit	::=	`0` | ... | `9`
IDChar	::=	Letter | Digit | '_' |

Figure A.1.: Identifier syntax

A.2. Programs and Declarations

Program	::=	Declaration { Declaration }
Declaration	::=	MainDecl
		| ProcedureDecl
		| RuleDecl
MainDecl	::=	`Main` '=' CommandSeq
ProcedureDecl	::=	ProcedureID '=' ['[' LocalDecl ']'] CommandSeq
LocalDecl	::=	(RuleDecl | ProcedureDecl) { LocalDecl }
CommandSeq	::=	Command {';'Command}
Command	::=	Block
		| `if` Block `then` Block [`else` Block]
		| `try` Block [`then` Block] [`else` Block]
Block	::=	'(' CommandSeq ')' ['!']
		| SimpleCommand
		| Block `or` Block
SimpleCommand	::=	RuleSetCall ['!']
		| ProcedureCall ['!']
		| `break`
		| `skip`
		| `fail`
RuleSetCall	::=	RuleID | '{' [RuleID { ',' RuleID }] '}'
ProcedureCall	::=	ProcedureID

Figure A.2.: Program syntax

A.3. Rule Syntax

<table>
<tr><td>RuleDecl</td><td>::=</td><td>RuleID ‘(’ [VarList {‘;’ VarList}] ‘;’ ‘)’
Graphs Interface [<code>where</code> Cond]</td></tr>
<tr><td>VarList</td><td>::=</td><td>Variable {‘,’ Variable} ‘:‘ Type</td></tr>
<tr><td>Graphs</td><td>::=</td><td>‘[’ Graph ‘]’ ‘=>’ ‘[’ Graph ‘]’</td></tr>
<tr><td>Graph</td><td>::=</td><td>[Position] ‘|’ {Nodes} ‘|’ {Edges}</td></tr>
<tr><td>Nodes</td><td>::=</td><td>{ ‘(’ NodeID [‘(R)’] ‘,’ Label [‘,’ Position] ‘)’ }</td></tr>
<tr><td>Edges</td><td>::=</td><td>{ ‘(’ EdgeID [‘(B)’] ‘,’ NodeID ‘,’ NodeID ‘,’ Label ‘)’ }</td></tr>
<tr><td>Position</td><td>::=</td><td>‘⟨’ Float ‘,’ Float ‘⟩’</td></tr>
<tr><td>Float</td><td>::=</td><td>[‘-’ | ‘+’] {Digit} [‘.’ Digit {Digit}] [‘e’ | ‘E’ [‘-’ | ‘+’] {Digit}]</td></tr>
<tr><td>Interface</td><td>::=</td><td><code>interface</code> ‘=’ ‘{’ [NodeID { ‘,’ NodeID }] ‘}’</td></tr>
<tr><td>Type</td><td>::=</td><td><code>int</code> | <code>char</code> | <code>string</code> | <code>atom</code> | <code>list</code></td></tr>
</table>

Figure A.3.: Rule declaration syntax

Positions store layout information for graphical editors. A position is a set of floating point cartesian coordinates. The position in the Graph rule specifies the canvas size of the graph. The position in the Nodes rule specifies the location of that node. Positions have no semantic meaning and are ignored by the parser.

<table>
<tr><td>Label</td><td>::=</td><td>List [‘#’ Mark]</td></tr>
<tr><td>List</td><td>::=</td><td><code>empty</code> | Atom | List ‘:’ List</td></tr>
<tr><td>Mark</td><td>::=</td><td><code>red</code> | <code>green</code> | <code>blue</code> | <code>grey</code> | <code>dashed</code> | <code>any</code></td></tr>
<tr><td>Atom</td><td>::=</td><td>Term {(‘+’ | ‘-’) Term}</td></tr>
<tr><td>Term</td><td>::=</td><td>Factor {(‘*’ | ‘/’ | ‘.’) Factor}</td></tr>
<tr><td>Factor</td><td>::=</td><td>Variable | Number | String | Char
| (<code>indeg</code> | <code>outdeg</code>) ‘(’ NodeID ‘)’
| <code>length</code> ‘(’ Variable ‘)’
| ‘-’ Factor
| ‘(’ Atom ‘)’</td></tr>
<tr><td>Number</td><td>::=</td><td>Digit {Digit}</td></tr>
<tr><td>Char</td><td>::=</td><td>‘ “ ’ Character ‘ ” ’</td></tr>
<tr><td>String</td><td>::=</td><td>‘ “ ’ {Character} ‘ ” ’</td></tr>
<tr><td>Character</td><td>::=</td><td>Printable characters except ‘ ” ’[1]</td></tr>
</table>

Figure A.4.: Rule label syntax

[1]ASCII characters 32, 33, and 35-126

```
Condition   ::=  Disjunct {or Disjunct}
Disjunct    ::=  Conjunct {and Conjunct}
Conjunct    ::=  Subtype '(' Variable ')'
                 | edge '(' NodeID ',' NodeID [ ',' Label ] ')'
                 | List ( '=' | '!=' ) List
                 | Atom RelOp Atom
                 | not Conjunct
                 | '(' Condition ')'
Subtype     ::=  int | char | string | atom
RelOp       ::=  '>' | '>=' | '<' | '<='
```

Figure A.5.: Condition syntax

A.4. Host Graph Syntax

```
HostGraph   ::=  [ Position ] {HostNodes} '|' {HostEdges}
HostNodes   ::=  { '(' NodeID ['(R)'] ',' HostLabel [ ',' Position ] ')' }
HostEdges   ::=  { '(' EdgeID ',' NodeID ',' NodeID ',' HostLabel ')' }
HostLabel   ::=  HostList ['#' HostMark]
HostMark    ::=  red | green | blue | grey | dashed
HostList    ::=  empty | HostExp | HostList ':' HostList
HostExp     ::=  [ '-' ] Number | String | Char
```

Figure A.6.: Host graph syntax

A.5. Type Grammar and Simple Lists

ListExpression	::=	ListTerm {‘:’ ListTerm }
ListTerm	::=	`empty` | ListVariable | AtomExpression
AtomExpression	::=	IntegerExpression | StringExpression | AtomVariable
IntegerExpression	::=	IntegerTerm {(‘`+`’ | ‘`-`’) IntegerTerm}
IntegerTerm	::=	IntegerFactor {(‘`*`’ | ‘`/`’) IntegerFactor}
IntegerFactor	::=	IntegerVariable | Number
		| (`indeg` | `outdeg`) ‘(’ NodeId ‘)’
		| `length` ‘(’ (AtomVariable | StringVariable | ListVariable) ‘)’
		| ‘-’ IntegerFactor
		| ‘(’ IntegerExpression ‘)’
StringExpression	::=	StringTerm {‘`.`’ StringTerm}
StringTerm	::=	CharExpression | StringVariable | String
CharExpression	::=	CharVariable | Char

Figure A.7.: Syntax of well-typed expressions

SimpleList	::=	SimpleListTerm {‘:’ SimpleListTerm}
SimpleListTerm	::=	`empty`
		| Variable
		| [‘-’] Number
		| StringExpression

Figure A.8.: Syntax of simple lists

A.6. Context Conditions

Name	Description
Identifier Length	Identifiers have a maximum length of 63 characters.
Reserved Words	GP keywords must not be used as identifiers
Main Declaration	There is exactly one main declaration in a single program.
Unique Procedure Names	Each procedure name must not be used in more than one procedure declaration.
Rule Declaration Scope	Each rule name must not be used in more than one declaration in the same scope.
Rule Call Validity	Any name in a rule call must belong to a rule declared in a visible scope.
Procedure Call Validity	Any name in a procedure call must belong to a procedure declared in a visible scope.
Break	The `break` statement must only appear within a loop. If the break is in the condition of a branching statement, its containing loop must occur within the same condition.
Variable Declarations 1	Each type must appear at most once in a rule's variable list.
Variable Declarations 2	Variable IDs must be distinct in the declaration list of a rule.
Interface Nodes	Each node ID in the interface list must appear exactly once and must occur in both the left-hand side and the right-hand side.
Bidirectional Edges 1	A right-hand side bidirectional edge must be incident to the same two nodes as a left-hand side bidirectional edge.
Bidirectional Edges 2	There is at most one bidirectional edge between a pair of nodes.
Node/Edge ID Uniqueness	Node IDs and edge IDs must be pairwise distinct within a single graph.

Figure A.9.: Context conditions (1)

Name	Description
Sources and Targets	The two node IDs in an edge declaration must match a node ID declared in the same graph.
Variable Consistency 1	Any variable in a rule must be declared in the variable list of the associated rule.
Variable Consistency 2	Any variable in the right-hand side must be present in the left-hand side of the same rule.
Grey and Dashed	Nodes must not be marked `dashed` and edges must not be marked `grey`.
Wildcard Consistency	A right-hand side item with the mark `any` must be in the interface of the rule and its counterpart in the left-hand side must be a wildcard.
Well-typed Expressions	Any expression in a label or condition must conform to the type grammar of Figure A.7.
Degree Operators	The argument of a degree operator (`indeg` or `outdeg`) must be a node ID occurring in the interface of its containing rule.
Simple Labels	Each expression in the left-hand side of a rule declaration must be a simple list as defined in Figure A.8
Edge Predicate	Each node ID in an edge predicate must occur in the interface of its containing rule.
Integer Comparisons	In a condition, the relational operators >, >=, <, and <= must only be applied to integer expressions.

Figure A.10.: Context conditions (2)

A.7. Keywords and Operators

Keyword	Type	Notes
`Main`	Command sequence	Declares the Main procedure.
`if`	Command sequence	Conditional branch
`try`	Command sequence	Conditional branch
`then`	Command sequence	Conditional branch
`else`	Command sequence	Conditional branch
`or`	Command sequence	Choice of two command sequences. Also used in conditions.
`break`	Command sequence	Exit the enclosing loop
`skip`	Command sequence	Always succeeds
`fail`	Command sequence	Always fails
`int`	Rule declaration	Variable declaration type. Also subtype predicate in conditions.
`char`	Rule declaration	Variable declaration type. Also subtype predicate in conditions.
`string`	Rule declaration	Variable declaration type. Also subtype predicate in conditions.
`atom`	Rule declaration	Variable declaration type. Also subtype predicate in conditions.
`list`	Rule declaration	Variable declaration type.
`interface`	Rule declaration	
`where`	Rule declaration	Declares application condition.
`and`	Condition	
`or`	Condition	Also used in command sequences.
`not`	Condition	

Figure A.11.: GP 2 keywords (1)

Keyword	Type	Notes
`edge`	Condition	Test for existence of an edge.
`int`	Condition	Type query.
`char`	Condition	Type query.
`string`	Condition	Type query.
`atom`	Condition	Type query.
`indeg`	Condition	Also used in rule labels.
`outdeg`	Condition	Also used in rule labels.
`length`	Condition	Also used in rule labels.
`empty`	Label	The empty list.
`red`	Label	Mark.
`blue`	Label	Mark.
`green`	Label	Mark.
`grey`	Label	Mark.
`dashed`	Label	Mark.
`any`	Label	Wildcard mark.

Figure A.12.: GP 2 keywords (2)

Operator	Context	Precedence	Notes
`!`	Command Sequence	1	
`;`	Command Sequence	2	
`or`	Command Sequence	3	
`=`	Condition	-	
`!=`	Condition	-	
`>`	Condition	-	
`>=`	Condition	-	
`<`	Condition	-	
`<=`	Condition	-	
`not`	Condition	1	
`and`	Condition	2	
`or`	Condition	3	
`:`	Expression	3	
`-`	Expression	1	Negation (Unary)
`+`	Expression	2	
`-`	Expression	2	Subtraction (Binary)
`*`	Expression	1	
`/`	Expression	1	Integer Division
`.`	Expression	3	

Figure A.13.: GP 2 operators. Precedence ranges from low (3) to high (1)

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AK08] Oana Andrei and Hélène Kirchner. A Rewriting Calculus for Multigraphs with Ports. In: *Proc. International Workshop on Rule Based Programming (RULE 2007).* Vol. 219. Electronic Notes in Theoretical Computer Science. 2008, pp. 67–82.

[And+11] Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In: *Proc. International Workshop on Computing with Terms and Graphs (TERMGRAPH 2011).* Vol. 48. Electronic Proceedings in Theoretical Computer Science. 2011, pp. 54–68.

[And+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph Transformation for Specification and Programming. In: *Science of Computer Programming* 34.1 (1999), pp. 1–54.

[Aub+12] David Auber, Daniel Archambault, Romain Bourqui, Antoine Lambert, Morgan Mathiaut, Patrick Mary, Maylis Delest, Jonathan Dubois, and Guy Melançon. *The Tulip 3 Framework: A Scalable Software Library for Information Visualization Applications Based on Relational Data.* Research Report RR-7860. INRIA, 2012.

[Aza+06] Karl Azab, Annegret Habel, Karl-Heinz Pennemann, and Christian Zuckschwerdt. ENFORCe: A System for Ensuring Formal Correctness of High-level Programs. In: *Proc. International Workshop on Graph-Based Tools (GraBaTs 2006).* Vol. 1. Electronic Communications of the EASST. 2006.

[Bak+15] Christopher Bak, Glyn Faulkner, Detlef Plump, and Colin Runciman. A Reference Interpreter for the Graph Programming Language GP 2. In: *Proc. Graphs as Models (GaM 2015).* Vol. 181. Electronic Proceedings in Theoretical Computer Science. 2015, pp. 48–64.

[Ber] Dan Bernstein. *djb2 Hash Function.* `http://www.cse.yorku.ca/~oz/hash.html`.

[Ber+08] Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In: *Proc. International Conference on Graph Transformation (ICGT 2008)*. Vol. 5214. Lecture Notes in Computer Science. Springer, 2008, pp. 396–410.

[BGT91] Horst Bunke, Thomas Glauser, and T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE-Matching Algorithm. In: *Proc. Graph Grammars and Their Application to Computer Science and Biology*. Vol. 532. Lecture Notes in Computer Science. Springer, 1991, pp. 174–189.

[BJG08] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. 2nd edition. Springer, 2008.

[BKG07] Gernot Veit Batz, Moritz Kroll, and Rubino Geiß. A First Experimental Evaluation of Search Plan Driven Graph Pattern Matching. In: *Applications of Graph Transformations with Industrial Relevance*. Vol. 5088. Lecture Notes in Computer Science. Springer, 2007.

[BKV03] Marc Bezem, Jan Willem Klop, and Roel de Vrijer, eds. *Term Rewriting Systems*. Cambridge University Press, 2003.

[BP12] Christopher Bak and Detlef Plump. Rooted Graph Programs. In: *Proc. International Workshop on Graph-Based Tools (GraBaTs 2012)*. Vol. 54. Electronic Communications of the EASST. 2012.

[Bun82] Horst Bunke. Attributed Programmed Graph Grammars and Their Application to Schematic Diagram Interpretation. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 4.6 (1982), pp. 574–582.

[Cor+97] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Editor Grzegorz Rozenberg. World Scientific, 1997, pp. 163–246.

[Det+12] Markus von Detten, Christian Heinzemann, Marie Christin Platenius, Jan Rieke, Dietrich Travkin, and Stephan Hildebrandt. *Story Diagrams - Syntax and Semantics*. Tech. rep. Software Engineering Group, Heinz Nixdorf Institute, Universtät Paderborn, 2012.

[Dod08] Mike Dodds. Graph Transformation and Pointer Structures. PhD thesis. Department of Computer Science, University of York, 2008.

[DP06a] Mike Dodds and Detlef Plump. Extending C for Checking Shape Safety. In: *Proc. Graph Transformation for Verification and Concurrency (GT-VC 2005)*. Vol. 154(2). Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[DP06b] Mike Dodds and Detlef Plump. Graph Transformation in Constant Time. In: *Proc. International Conference on Graph Transformation (ICGT 2006)*. Vol. 4178. Lecture Notes in Computer Science. Springer, 2006, pp. 367–382.

[Dör95] Heiko Dörr. *Efficient Graph Rewriting and its Implementation*. Vol. 922. Lecture Notes in Computer Science. Springer, 1995.

[Ehr+06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[Ehr+97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Editor Grzegorz Rozenberg. World Scientific, 1997, pp. 247–312.

[Ehr+99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, eds. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.

[Ell13] Alex Elliott. Towards an Integrated Development Environment for GP 2. MEng Thesis. Department of Computer Science, University of York, 2013.

[ELS87] Gregor Engels, Claus Lewerentz, and Wilheim Schäfer. Graph Grammar Engineering: A Software Specification Method. In: *Proc. International Workshop on Graph-Grammars and Their Application to Computer Science*. Vol. 291. Lecture Notes in Computer Science. Springer, 1987, pp. 186–201.

[Eng+92] Gregor Engels, Claus Lewerentz, Manfred Nagl, Wilheim Schäfer, and Andy Schürr. Building Integrated Software Development Environments. Part I: Tool Specification. In: *ACM Transactions on Software Engineering and Methodology* 1.2 (1992), pp. 135–167.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans J. Schneider. Graph-grammars: An Algebraic Approach. In: *Proc. Symposium on Switching and Automata Theory (SWAT 1973)*. IEEE Computer Society, 1973, pp. 167–180.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In: *Graph Transformations*. Vol. 3256. Lecture Notes in Computer Science. Springer, 2004, pp. 161–177.

[ERT99] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG Approach: Language and Environment. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Editor Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 2. World Scientific, 1999. Chap. 14, pp. 551–603.

[Fau15] Glyn Faulkner. *GraphGen*. `https://github.com/UoYCS-plasma/GP2/blob/master/Haskell/Tools/graphgen.ml`. 2015.

[FB93] Hoda Fahmy and Dorothea Blostein. A Graph Grammar Programming Style for Recognition of Music Notation. In: *Machine Vision and Applications* 6 (1993), pp. 83–99.

[Fis+00] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: *Theory and Application of Graph Transformations (TAGT 1998)*. Vol. 1764. Lecture Notes in Computer Science. Springer, 2000, pp. 296–309.

[FKN12] Maribel Fernández, Hélène Kirchner, and Olivier Namet. A Strategy Language for Graph Rewriting. In: *Logic-Based Program Synthesis and Transformation (LOPSTR 2011)*. Vol. 7225. Lecture Notes in Computer Science. Springer, 2012, pp. 173–188.

[FKP14] Maribel Fernández, Hélène Kirchner, and Bruno Pinaud. Strategic Port Graph Rewriting: An Interactive Modelling and Analysis Framework. In: *Proc. Workshop on GRAPH Inspection and Traversal Engineering (GRAPHite 2014)*. Vol. 159. Electronic Proceedings in Theoretical Computer Science. 2014, pp. 15–29.

[GB95] Ann Grbavec and Dorothea Blostein. Mathematics Recognition Using Graph Rewriting. In: *Proc. International Conference on Document Analysis and Recognition (ICDAR 1995)*. IEEE Computer Society, 1995, pp. 417–421.

[Gei+06] 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)*. Lecture Notes in Computer Science. Springer, 2006, pp. 383 –397.

[Gha+12] Amir H. Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and Analysis using GROOVE. In: *Software Tools for Technology Transfer* 14.1 (2012).

[GJR10] Amir H. Ghamarian, Arash Jalali, and Arend Rensink. Incremental Pattern Matching in Graph-Based State Space Exploration. In: *Proc. International Workshop on Graph-Based Tools (GraBaTs 2010)*. Vol. 32. Electronic Communications of the EASST. 2010.

[Gru+05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Dániel Varró. Using Graph Transformation for Practical Model-Driven Software Engineering. In: *Model-Driven Software Development*. Editor Sami Beydeda, Matthias Book, and Volker Gruhn. Springer, 2005, pp. 91–117.

[HHT95] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. In: *Fundamenta Informaticae* 26 (1995), pp. 287–313.

[HJG08] Berthold Hoffmann, Edgar Jakumeit, and Rubino Geiß. Graph Rewrite Rules with Structural Recursion. In: *Proc. International Workshop on Graph Computation Models (GCM 2008)*. 2008, pp. 5–16.

[Hor+10] Ákos Horváth, Gábor Bergmann, István Ráth, and Dániel Varró. Experimental Assessment of Combining Pattern Matching Strategies with VIATRA2. In: *Software Tools for Technology Transfer* 12.3-4 (2010), pp. 211–230.

[HP01] Annegret Habel and Detlef Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In: *Proc. International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2001)*. Springer, 2001, pp. 230–245.

[HP02] Annegret Habel and Detlef Plump. Relabelling in Graph Transformation. In: *Proc. International Conference on Graph Transformation (ICGT 2002)*. Vol. 2505. Lecture Notes in Computer Science. Springer, 2002, pp. 135–147.

[HP15] Ivaylo Hristakiev and Detlef Plump. A Unification Algorithm for GP 2. In: *Graph Computation Models (GCM 2014), Revised Selected Papers*. Vol. 71. Electronic Communications of the EASST. 2015.

[HPR06] Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest Preconditions for High-Level Programs. In: *Proc. International Conference of Graph Transformation (ICGT 2006)*. Vol. 4178. Lecture Notes in Computer Science. Springer, 2006, pp. 445–460.

[HVV07] Ákos Horváth, Gergely Varró, and Dániel Varró. Generic Search Plans for Matching Advanced Graph Patterns. In: *Proc. International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Vol. 6. Electronic Communications of the EASST. 2007, pp. 57–68.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. GrGen.NET - The Expressive, Convenient and Fast Graph Rewrite System. In: *Software Tools for Technology Transfer* 12.3–4 (2010), pp. 263–271.

[JGR12] Arash Jalali, Amir H. Ghamarian, and Arend Rensink. Incremental Pattern Matching for Regular Expressions. In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2012).* Vol. 47. Electronic Communications of the EASST. 2012.

[Kah62] A. B. Kahn. Topological Sorting of Large Networks. In: *Communications of the ACM* 5.11 (1962), pp. 558–562.

[LE91] Michael Löwe and Hartmut Ehrig. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. In: *Graph-Theoretic Concepts in Computer Science.* Vol. 484. Lecture Notes in Computer Science. Springer, 1991, pp. 338–353.

[Lev09] John Levine. *Flex and Bison.* O'Reilly, 2009.

[LKW93] Michael Löwe, Martin Korff, and Annika Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In: *Term Graph Rewriting: Theory and Practice.* John Wiley, 1993, pp. 185–199.

[MG07] Jens Müller and Rubino Geiß. *Speeding up Graph Transformation through Automatic Concatenation of Rewrite Rules.* Tech. rep. Universität Karlsruhe, 2007.

[Mon70] Ugo Montanari. Separable Graphs, Planar Graphs and Web Grammars. In: *Information and Control* 16.3 (1970), pp. 243–267.

[MP08a] Greg Manning and Detlef Plump. The GP Programming System. In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008).* Vol. 10. Electronic Communications of the EASST. 2008.

[MP08b] Greg Manning and Detlef Plump. The York Abstract Machine. In: *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006).* Vol. 211. Electronic Notes in Theoretical Computer Science. Elsevier, 2008, pp. 231–240.

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. In: *Proc. Workshop on Software Evolution Through Transformations: Model-Based vs. Implementation-Level Solutions.* Vol. 127. Electronic Notes in Theoretical Computer Science. 2005, pp. 113–128.

[Nag78] Manfred Nagl. A Tutorial and Bibliographical Survey on Graph Grammars. In: *Graph-Grammars and Their Application to Computer Science and Biology.* Vol. 73. Lecture Notes in Computer Science. Springer, 1978, pp. 70–126.

[NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA Environment. In: *Proc. International Conference on Software Engineering (ICSE 2000).* ACM Press, 2000, pp. 742–745.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*. Vol. 42. ACM SIGPLAN Notices. ACM, 2007, pp. 89–100.

[Pen09] Karl-Heinz Pennemann. Development of Correct Graph Transformation Systems. PhD thesis. Carl von Ossietzky Universität Oldenburg, 2009.

[PJS04] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals*. 2nd edition. Springer, 2004.

[Plo04] Gordon D. Plotkin. A Structural Approach to Operational Semantics. In: *Journal of Logic and Algebraic Programming* 60-61 (2004), pp. 17–139.

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

[Plu09] Detlef Plump. The Graph Programming Language GP. In: *Proc. Algebraic Informatics (CAI 2009)*. Vol. 5725. Lecture Notes in Computer Science. Springer, 2009, pp. 99–122.

[Plu12] Detlef Plump. The Design of GP 2. In: *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*. Vol. 82. Electronic Proceedings in Theoretical Computer Science. 2012, pp. 1–16.

[Plu93] Detlef Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In: *Term Graph Rewriting: Theory and Practice*. John Wiley, 1993, pp. 201–213.

[PMD12] Bruno Pinaud, Guy Melançon, and Jonathan Dubois. PORGY: A Visual Graph Rewriting Environment for Complex Systems. In: *Computer Graphics Forum*. Eurographics Conference on Visualization (EuroVis 2012) 31.3 (2012), pp. 1265–1274.

[Pos13] Christopher M. Poskitt. Verification of Graph Programs. PhD thesis. Department of Computer Science, University of York, 2013.

[PPEM87] Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph Rewriting with Unification and Composition. In: *Proc. 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*. Vol. 291. Lecture Notes in Computer Science. Springer, 1987, pp. 496–514.

[PR69] John L. Pfaltz and Azriel Rosenfeld. Web Grammars. In: *Proc. 1st International Joint Conference on Artifical Intelligence*. 1969, pp. 609–619.

[PS04] Detlef Plump and Sandra Steinert. Towards Graph Programs for Graph Algorithms. In: *International Conference on Graph Transformation (ICGT 2004)*. Vol. 3256. Lecture Notes in Computer Science. Springer, 2004, pp. 128–143.

[PSS11] Detlef Plump, Robin Suri, and Ambuj Singh. Minimizing Finite Automata with Graph Programs. In: *Graph Computation Models (GCM 2010)*. Vol. 39. Electronic Communications of the EASST. 2011.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In: *Proc. Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003)*. Vol. 3062. Lecture Notes in Computer Science. Springer, 2004, pp. 479–485.

[Ren06a] Arend Rensink. Isomorphism Checking in GROOVE. In: *Proc. International Workshop on Graph-Based Tools (GraBaTs 2006)*. Vol. 1. Electronic Communications of the EASST. 2006.

[Ren06b] Arend Rensink. Nested Quantification in Graph Transformation Rules. In: *Proc. International Conference on Graph Transformation (ICGT 2006)*. Vol. 4178. Lecture Notes in Computer Science. Springer, 2006.

[RET11] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 – New Features for Specifying and Analyzing Algebraic Graph Transformations. In: *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*. Vol. 7233. Lecture Notes in Computer Science. Springer, 2011, pp. 81–88.

[Roz97] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific, 1997.

[Rud98] Michael Rudolf. Utilizing Constraint Satisfaction Techniques for Efficient Graph Pattern Matching. In: *Proc. International Workshop on Theory and Applications of Graph Transformation (TAGT 1998)*. Springer, 1998, pp. 238–251.

[Sch91a] Andy Schürr. Operationales Spezifizieren mit Programmierten Graphersetzungssystemen. In German. PhD thesis. Deutscher Universitäts-Verlag, 1991.

[Sch91b] Andy Schürr. PROGRESS: A VHL-Language Based on Graph Grammars. In: *Proc. International Workshop on Graph-Grammars and Their Application to Computer Science*. Vol. 532. Lecture Notes in Computer Science. Springer, 1991, pp. 641–695.

[Sch97] A. Schürr. Programmed Graph Replacement Systems. In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Editor Grzegorz Rozenberg. World Scientific, 1997, pp. 479–546.

[Sed02] Robert Sedgewick. *Algorithms in C: Part 5: Graph Algorithms.* Addison-Wesley, 2002.

[SG07] Andreas Schösser and Rubino Geiß. Graph Rewriting for Hardware Dependent Program Optimizations. In: *Proc. International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007).* Vol. 5008. Lecture Notes in Computer Science. Springer, 2007.

[Sil02] Alan Silverstein. *Judy IV Shop Manual.* `http://judy.sourceforge.net/`. 2002.

[Ski08] Steven S. Skiena. *The Algorithm Design Manual.* 2nd edition. Springer, 2008.

[Sta01] Richard Stallman. *Using and Porting the GNU Compiler Collection.* Iuniverse Inc, 2001.

[Ste07] Sandra Steinert. The Graph Programming Language GP. PhD thesis. Department of Computer Science, University of York, 2007.

[SWZ99] Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In: *Handbook of Graph Grammars and Computing by Graph Transformation.* Editor Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Vol. 2. World Scientific, 1999. Chap. 13, pp. 487–550.

[Tae+08] 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: *Proc. Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007).* Vol. 5088. Lecture Notes in Computer Science. Springer, 2008, pp. 514–539.

[Tea] The GHC Team. *The Glasgow Haskell Compiler.* `http://www.haskell.org/ghc/documentation.html`.

[THCRS09] Charles E. Leiserson Thomas H. Cormen, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 3rd edition. The MIT Press, 2009.

[Ull76] Julian R. Ullmann. An Algorithm for Subgraph Isomorphism. In: *Journal of the ACM* 23.1 (1976), pp. 31–42.

[Var+12] Gergely Varró, Frederik Deckwerth, Martin Wieber, and Andy Schürr. An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In: *Proc. Theory and Practice of Model Transformations (ICMT 2012).* Vol. 7307. Lecture Notes in Computer Science. Springer, 2012, pp. 224–239.

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