Time and Space Complexity of Rule-based Graph Programs

Brian Courtehoute

Doctor of Philosophy

University of York
Computer Science

March 2023

Abstract

This thesis concerns the time and space efficiency of programs in GP 2, a rule-based graph transformation language that facilitates formal program analysis. Such programs are a sequence of control structures in which rules are called. A rule describes how part of a host graph is changed to another by specifying a subgraph that is to be replaced. We call the process of finding the specified subgraph in the host graph *matching*, which takes polynomial time in general. In practise however, we often want rule application to take constant time since it likely corresponds to a single step in a classical algorithm.

Several case studies show that the time complexity of GP 2 programs can be on the same level as that of their imperative counterparts. We give linear-time programs for connectedness checking, DAG recognition, and topological sorting, as well as an efficient implementation of Boruvka's algorithm for finding minimum spanning trees. This efficiency is achieved via *roots*, which are special nodes in rules and graphs that can be accessed in constant time and allow matching to happen locally around them. The given programs also use depth-first search to traverse graphs in linear time instead of iterating over nodes because GP 2 abstracts away from internal graph data structures.

In the spirit of formal program analysis, we give a framework in which to describe the time complexity of these efficient programs. This framework is underpinned by a formal semantics that describes program execution in a sequence of steps that do not cover more than one rule application.

On the topic of space efficiency, we give a theoretical result that shows GP 2, like some graph-based machine models, can simulate Turing machines using less space and only quadratic time overhead.

List of Contents

List of Figures

Acknowledgements

I am incredibly grateful to my supervisor, Detlef Plump, for his guidance, detailed discussions, and encouragement over the years. His support was a great benefit to my work. I also thank my internal examiner Jeremy Jacob for valuable feedback and new perspectives, as well as my external examiner Frank Drewes for agreeing to travel all the way from Sweden to attend the viva.

I thank both Steve King and Detlef Plump for giving me the opportunity to teach, and for guiding me towards being a better teacher. Teaching is an incredibly fulfilling experience, and I will miss it.

My fellow research students have shown great kindness over the years. I thank Robert, Federico, Tim, and Gia for providing me with a pleasant research environment. Robert especially has been invaluable in the final year, having discussions and feedback that were both fun and helpful, and being there while I have been overcoming the final hurdles.

I thank all the friends I have made during my time in York and Leeds for making these years the most enjoyable part of my life so far. My horizons have been broadened to no end, and I have gained many precious memories.

Finally, I am deeply grateful to my family, especially my mother, for their endless love, encouragement, and support. They were invaluable in getting through the hardships I have faced.

Declaration

I declare that this thesis is a presentation of original work, except where otherwise attributed or cited to another author. This work has not previously been presented for an award at this, or any other, university. Some parts of this thesis have been previously published in journal, conference, and workshop papers, as described in Section 1.4.

Brian Courtehoute, March 2023

1 Introduction

Our first step in this thesis is to outline the motivation behind it in Section 1.1. We then describe the goal of this thesis and how our contributions achieve it in Section 1.2. In Section 1.3, we outline the structure of this thesis. Section 1.4 contains the history of publications that form the basis this thesis.

1.1 Motivation

In rule-based graph transformation, we modify graphs by applying rules to them. A rule specifies how this modification occurs using a left- and a right-hand side graph. A *host graph* is modified by finding a subgraph that matches the left-hand side, and changing it to match the right-hand side.

Rule-based graph transformation is an important area of research because it provides a formal basis for programs that transform graphs, which are ubiquitous in computer science. The use of graph transformation enables the analysis and verification of programs in a formal way, which is crucial for ensuring their correctness and efficiency. Graphs are widely used to model complex systems and processes, such as communication networks [82], social networks [13], relational databases [88], UML diagrams [96], pointer structures [29], longitudinal information systems [92], and chemical reactions [3]. Additionally, graph transformation can be used to convert computer graphics files containing virtual worlds [6], phenotype plants [68], assist in engineering design [80], geometrically model quasi-manifolds [56], and model epidemics [90]. By using rule-based graph transformation, we can naturally express and manipulate graph algorithms concerning topological sortings, minimum spanning trees, and many other problems. This makes rule-based graph transformation a powerful and versatile tool in computer science research and development.

There are numerous theoretical results in this field [35], but also a lot of work on programming languages based on rule-based graph transformation.

Such languages include PROGRES [72], AGG [67], GReAT [1], GROOVE [41, 48], GrGen.Net [47, 48], Henshin [79, 48] and PORGY [36].

In this thesis, we focus on the experimental programming language GP 2, whose aim is to facilitate formal reasoning. It is nondeterministic since applying a rule can result in different graphs, and since it allows for a set of rules to be called nondeterministically. The language comes with a simple formal semantics [25], and a C compiler [8, 11]. GP 2 is computationally complete in that it can implement any computable function on graphs [60, 42]. Research on this language includes Hoare-calculi for program verification [63, 64, 94, 95], static analysis for confluence checking [46], and formalisation in a proof assistant [78]. GP 2 is introduced in [58] as the successor to GP [53].

In general, applying a rule is less efficient than we desire an atomic operation to be due to graph matching. Given the left-hand side L of a rule, and a host graph G, the number of possible matches is bounded by $\text{size}(G)^{\text{size}(L)}$, which is polynomial since a rule is part of a program and hence considered of fixed size. We consider the size of a graph to be the number of nodes and edges plus the size of its labels. The aforementioned bound is reached if L and G have no edges for instance. In the worst case, only one of the possible node mappings makes the labels match, and all mappings have to be checked for that.

Note that graph matching is not the same as the subgraph isomorphism problem, which is NP-complete [39, 84]. In the latter, both graphs are considered part of the input, whereas in the graph matching problem, one of the graphs is considered fixed.

A way to mitigate expensive matching is to use root nodes (or roots), first introduced by Dörr [30]. They are special nodes in a graph that can be accessed in constant time if there are only boundedly many, which can be implemented using a list of pointers for instance. Roots can only match roots, which allows matching to occur locally around them. They can be thought of as nodes that are currently being worked on by an algorithm.

Rules with roots can be shown to apply in constant time under some mild conditions. One such approach [10] uses *fast* rules that can be matched in constant time for connected host graphs of bounded degree with a bounded number of roots.

A bounded number of roots can easily be achieved by carefully managing them during programming. Whether or not the node degree is bounded depends on the application. Traffic networks or electrical circuits for instance can be considered bounded-degree graphs.

The first GP 2 program that used this technique is 2-colouring [10, 11], which is shown to take linear time on connected graphs of bounded degree. Furthermore, the compiled program's performance is measured for input graphs with up to $100,000$ nodes, and found to match that of Sedgewick's C implementation of 2-colouring [75].

Improvements to the GP 2 compiler [21, 20] led to time-efficient matching in more cases. For instance, if we restrict rules to *fast outgoing* rules, but extend host graphs to include graphs of only bounded outdegree, matching can still happen in constant time. This technique is first used in [26].

A deliberate design feature of GP 2 is that internal data structures are abstracted away from. The programmer only has access to rules, graphs, and programs. This means one can establish theoretical results without relying on the implementation (correctness or termination proofs for instance). Because of this, graph data structures cannot be exploited, which is a common practice in many graph algorithms. When needing to iterate over nodes for instance, instead of going through an array of nodes in a data structure, one needs to do a depth-first search. In both approaches, there is a trade-off between convenience and high-level reasoning.

The programmer may wish to use additional data structures, like lists or stacks. These can be implemented within a host graph. The GP 2 implementation of topological sorting for instance [19] implements a stack using nodes and edges to keep track of its top and the overall sequence of nodes.

When it comes to space efficiency, we build on a theoretical result. There are graph-based machine models, namely Schönhage's storage modification machines [69] and Kolmogorov-Uspenskii machines [49], that exhibit a space compression property. That is, they can simulate Turing machines using less space and only polynomial time overhead [85, 52].

To the best of our knowledge, in the field of rule-based graph transformation, there is no other research on concrete time bounds for well-known graph algorithms. That is the starting point of this thesis.

1.2 Aim and Contributions

The aim of this thesis is to show that rule-based graph programs, specifically in GP 2, can be efficient with respect to both time and space, and to provide a framework in which to give formal arguments about these programs' complexity. We do this using the following contributions.

A small-step semantics. We introduce a new semantics for GP 2 that is small-step, i.e. every step described by the semantics corresponds to no more than one computation step of a program. This fine-grained approach to describing program execution provides a solid foundation for a complexity framework. The new semantics is equivalent to the original semantics (in that they can simulate each other) for terminating programs, and improves upon it for non-terminating programs.

A formal complexity framework. We establish what constraints a set of programs and inputs have to satisfy in order for us to define time and space complexity measures. One constraint is constant-time matching for each rule. This can be shown using a variety of results with different assumptions on rule and host graph structure, based on assumptions on the time complexity of various compiler operations [10, 19]. The other constraint is that backtracking (i.e. undoing parts of the program) is avoided, since that can create either time or space overhead. This framework is established by going into implementation details so that a programmer using it only has to reason about rules, graphs, and programs to get complexity results.

A collection of efficient graph programs. We provide GP 2 programs that have the same asymptotic complexity (on graphs of bounded degree) as equivalent imperative implementations. We argue about correctness and complexity by giving theoretical proofs and/or empirical evidence. The programs achieve this by using linear-time depth-first search and rules that can be matched in constant time under mild conditions. These programs are the following.

- `is-connected` recognises whether an input graph is connected or not in linear time.
- `top-sort` recognises whether an input graph is a directed acyclic graph (DAG) and constructs a topological sorting if it is, all in linear time.
- `mst-boruvka` implements Boruvka's algorithm for finding a minimum spanning tree in linear time.

A space compression property. We show that GP 2 shares the property of space compression with the graph-based computation models known as storage modification machines and Kolmogorov-Uspenskii machines. A class of programs in a subset of GP 2 can simulate Turing machines using less space and only quadratic time overhead. We give proofs for correctness as well as time and space complexity.

1.3 Thesis Structure

This thesis is divided into chapters which contain the following.

- *Chapter 2* defines the programming language GP 2 in detail. It describes the mathematical foundations of applying graph transformation rules with roots and labels. On a higher level, it describes the structure of GP 2 programs, i.e. the syntax, as well as program behaviour using semantics. Both the original and an improved semantics are given and compared.

- *Chapter 3* gives a framework for describing the complexity of GP 2 programs. It starts with assumptions on low-level operations of the compiler, and uses those on the matching algorithm to show it is constant-time in some cases. This is then used to establish time and space complexity measures under some assumptions.

- *Chapter 4* provides case studies of GP 2 programs that have efficient time complexities. Arguments for their correctness and efficiency are given. These programs implement connected graph recognition, topological sorting, directed acyclic graph (DAG) recognition, and minimum spanning trees.

- *Chapter 5* shows that GP 2 can simulate Turing machines using less space and a quadratic time overhead. This is done by giving proofs for the time and space complexity of a class of programs in a subset of GP 2.

- *Chapter 6* caps this thesis off by drawing conclusions and providing some items of future work.

Chapters for which it is relevant come with a section or subsection on related work to provide context. Literature on relevant topics is cited and discussed. They are in a separate section in order to preserve the narrative flow of the other sections.

We assume that the reader has elementary knowledge in theoretical computer science and mathematics, particularly when it comes to discrete mathematics, logic, and notation. However we do give some basic definitions for the sake of rigour. We also assume a familiarity with basic graph theory terms [23, 93] such as incoming edges, outgoing edges, indegree, adjacent nodes, loops, connectedness, reachability, etc.

1.4 Publication History

The results of this thesis are published in several peer-reviewed journal, conference, and workshop papers, as listed below. Each entry is followed by a description of this thesis' author's contributions for co-authored papers, and of which part of the thesis they are related to.

[18] Graham Campbell, Brian Courtehoute & Detlef Plump (2019): *Linear-Time Graph Algorithms in GP 2.* In Markus Roggenbach & Ana Sokolova, editors: *Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019), Leibniz International Proceedings in Informatics (LIPICS)* 139, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, pp. 16:1–16:23, doi:10.4230/LIPIcs.CALCO.2019.16.

This thesis' author contributes everything related to the recognition of binary directed acyclic graphs (DAGs) and to topological sorting to this paper. Those programs do not feature in this thesis as such because they are improved upon in the extended journal version [19] of this paper.

[19] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs. Science of Computer Programming* 214, doi:10.1016/j.scico.2021.102727.

An extended journal version of [18], adding further programs and more detailed proofs. This thesis' author contributes everything related to connectedness recognition, topological sorting, and general directed acyclic graph (DAG) recognition. These programs form the basis for two of the case studies in Chapter 4. Elements of the approach for showing a program's time complexity can be found in Chapter 3.

[24] Brian Courtehoute & Detlef Plump (2020): *Fast Graph Program for Computing Minimum Spanning Trees.* In: *Proc. 11th International Workshop on Graph Computation Models (GCM 2020), Electronic Proceedings in Theoretical Computer Science* 330, pp. 163-180, doi:10.4204/eptcs.330.10. Revised Selected Papers.

We introduce an efficient GP 2 implementation of Boruvka's algorithm for finding minimum spanning trees, including empirical time measurements. This case study can be found in Chapter 4.

[25] Brian Courtehoute & Detlef Plump (2021): *A Small-Step Operational Semantics for GP 2.* In: *Proc. 12th International Workshop on Graph Computation Models (GCM 2021)*, *Electronic Proceedings in Theoretical Computer Science* 350, pp. 89-110, doi:10.4204/eptcs.350.6. Revised Selected Papers.

We introduce a new small-step operational semantics for GP 2. We show some of its properties and prove that it is equivalent to the original semantics on terminating programs. These results are presented in Section 2.3.

[26] Brian Courtehoute & Detlef Plump (2022): *Time and Space Measures for a Complete Graph Computation Model.* In: *Proc. 13th International Workshop on Graph Computation Models (GCM 2022)*, *Electronic Proceedings in Theoretical Computer Science* 374, pp. 23-44, doi:10.4204/eptcs.374.4. Revised Selected Papers.

We show that GP 2 can simulate Turing machines in less space with only a quadratic time overhead, which can be found in Chapter 5. This paper also establishes the foundations of the complexity framework in Chapter 3.

2 The Programming Language GP 2

GP 2 is a programming language that takes graphs as input, runs a program on it that transforms it using various control constructs, and outputs the resulting graph, as illustrated in Figure 2.1. The language is first introduced in [58] and implemented as a GP 2-to-C compiler in [8]. Updates to the compiler are introduced in [21, 20]. GP 2 is computationally complete, in that every computable function on graphs can be programmed [60].

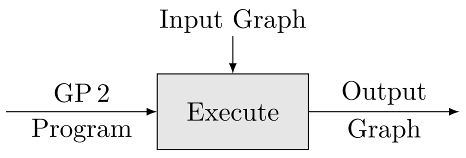


Figure 2.1: The GP 2 computational model

In GP 2, we use attributed graphs, i.e. nodes and edges carry information via marks and labels. Marks are represented using 5 colours, a fixed set of values that are hard-coded into GP 2. Labels include integers, strings, and lists thereof. Edges are directed, and we allow loops and parallel edges. Our graphs also feature roots, a subset of nodes that are singled out for fast access. Computation happens on the structure of graphs, as well as on attributes. GP 2 allows for various operations on labels, such as arithmetic expressions and list concatenation. This facilitates programming algorithms with weighted edges for instance, such as finding a minimum spanning tree, or computing a shortest path.

In Section 2.1 we give the theory behind double-pushout graph transformation. Subsection 2.1.1 introduces graphs, and 2.1.2 morphisms, which we use to map graphs into each other. Subsection 2.1.3 defines the rules we use to transform graphs, which is followed by an introduction to the concepts of category theory in Subsection 2.1.4. We then use these concepts to define direct derivation in Subsection 2.1.5, and talk about existence and uniqueness in Subsection 2.1.6.

We then move on to attributed graph transformation in Section 2.2, where we introduce GP 2 labels in Subsection 2.2.1, and more expressive rules, called rule schemata, that use these labels in Subsection 2.2.2. Next, we describe how we apply rule schemata in Subsection 2.2.3, and go into the problem of relabelling and rooting in Subsection 2.2.4.

Section 2.3 gives an overview of GP 2 programs and the theory behind them. The syntax is defined in Subsection 2.3.1, followed by a description of how components of a program behave in Subsection 2.3.2, and an example program in Subsection 2.3.3. Next, Subsection 2.3.4 gives motivation and context for why we have two GP 2 semantics. The original semantics is given in Subsection 2.3.5, and the small-step semantics in Subsection 2.3.6. We show several properties of the latter semantics in Subsection 2.3.7 before comparing the two in Subsection 2.3.8. In Subsection 2.3.9, we talk about semantics in the literature.

2.1 Double-Pushout Graph Transformation

In this section, we describe double-pushout graph transformation with relabelling and re-rooting. This serves as the foundation of GP 2 programs.

2.1.1 Graphs

We use finite graphs with directed edges that allow for loops (edges from a node to itself) and parallel edges (multiple edges between two nodes). Graphs have a set of labels which will remain unspecified in this section.

Definition 2.1 (Function)**.** Let A and B be sets. A *function* $f\colon A \to B$ associates to each element $a \in A$ a unique element $f(a) \in B$.

We say $g\colon A \xrightarrow{par} B$ is a *partial function* if there is a subset $A' \subseteq A$ such that $g\colon A' \to B$ is a function and g does not associate any element of B to any element of $A - A'$. We say $A' = \mathrm{dom}(g)$ is the *domain* of g. We call g *undefined* on $A - A'$. We say that g is a *total function* if $A' = A$. □

Definition 2.2 (Function Properties)**.** Let $f\colon A \to B$ be a function.

- We say f is *injective* if for each $b \in B$, there is at most one $a \in A$ such that $f(a) = b$.
- We say f is *surjective* if for each $b \in B$, there is at least one $a \in A$ such that $f(a) = b$.

- We say f is *bijective* if it is both injective and surjective, i.e. for each $b \in B$, there is a unique $a \in A$ such that $f(a) = b$. We call a the *preimage* of b. If f is bijective, we define its *inverse* $f^{-1}: B \to A$ by $f^{-1}(b) = a$ for each $b \in B$, where a is the unique preimage of b.

- We say two functions $f: A \to B$ and $g: C \to D$ are *equal* (commonly known as equality by extensionality), i.e. $f = g$, if $A = C$, $B = D$, and for each $x \in A = C$, $f(x) = g(x)$.

- We call $\mathrm{id}_A: A \to A$ the *identity function* on A if, for each $a \in A$, $\mathrm{id}_A(a) = a$. □

Note that the inverse of a bijective function is a bijective function.

Definition 2.3 (Operations on Functions)**.** We define the following operations on functions.

- Given two functions $f: A \to B$ and $g: B \to C$, their *composition* $g \circ f: A \to C$ is defined by $g \circ f(x) = g(f(x))$ for each $x \in A$.

- Given a function $f: A \to B$ and a subset $C \subseteq A$, the *restriction* $f|_C: C \to B$ of f to C is defined by $f|_C(x) = f(x)$ for all $x \in C$. □

The restriction of partial functions is defined analogously. Note that the restriction of a partial function to its domain is a function.

Definition 2.4 (Graph)**.** Let $\mathcal{L}$ be a set of *labels*. A *graph* over $\mathcal{L}$ is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G, p_G \rangle$ whose components are as follows.

- V_G and E_G are finite sets of *nodes* and *edges*.

- $s_G: E_G \to V_G$ is the *source* function for edges.

- $t_G: E_G \to V_G$ is the *target* function for edges.

- $l_G: V_G \xrightarrow{par} \mathcal{L}$ is the partial *node labelling* function.

- $m_G: E_G \to \mathcal{L}$ is the *edge labelling* function.

- $p_G: V_G \xrightarrow{par} \{0, 1\}$ is the partial *rootedness* function. A node $v \in V_G$ is *rooted* or a *root* if $p_G(v) = 1$, and *unrooted* or a *non-root* if $p_G(v) = 0$.

We call elements of $V_G \cup E_G$ *items*, and denote them as $x \in G$.

The graph G is called *totally labelled* if its node labelling function l_G is total. We call nodes v for which l_G is undefined *unlabelled*, denoted as $l_G(v) = \perp$ (assuming $\perp \notin \mathcal{L}$), thereby extending l_G to a total function on $\mathcal{L} \cup \{\perp\}$.

The graph G has *defined rootedness* if its rootedness function p_G is total. We say nodes v for which p_G is undefined have *undefined rootedness*, denoted as $p_G(v) = \perp$, thereby extending p_G to a total function on $\{0, 1, \perp\}$. □

Rootedness is a special status that, if given only to a handful of nodes, allows for fast access to the local area around the roots. Intuitively, they can be seen as the "current node" described in graph algorithms, and used to traverse a graph for instance. Roots are drawn with double borders. They are modelled using a partial function that maps a node to either 0 or 1, meaning that node is unrooted or rooted respectively. We model it as a partial function for the same reason node labels are modelled with a partial function. Note that a graph of defined rootedness need not have roots. It only means that each of its nodes is either a root or not.

Not that we require the node labelling and rootedness functions to be partial. This is because it allows us to map nodes while abstracting away from their labels and rootedness. More details on this can be found in Subsection 2.2.4.

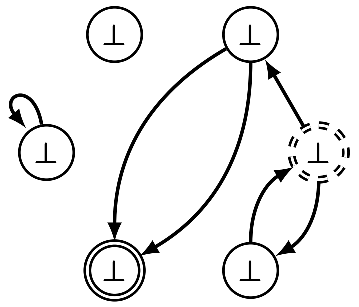


Figure 2.2: Example of a graph

An example of how we represent graphs can be seen in Figure 2.2. We draw nodes using circles or pill shapes. Root nodes have double borders. Nodes of undefined rootedness are drawn with dashed double borders. Edges are drawn as arrows from their source to their target nodes. Note that nodes contain the symbol $\perp$, which stands for an undefined label. We elaborate on how labels are drawn once the labels we use in GP 2 specifically are defined (Subsection 2.2.1).

2.1.2 Morphisms

Let us define how we map graphs into each other with structure-preserving functions called graph morphisms. We sometimes omit the word 'graph' since we do not use other kinds of morphisms.

Definition 2.5 (Graph Morphism)**.** Let G and H be graphs. A *(graph) morphism* $g\colon G \to H$ is a pair $g = \langle g_V,\, g_E\rangle$ of functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ such that the following holds.

- *Preservation of sources and targets:* for each edge $e \in E_G$, we have $g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$.
- *Preservation of edge labels:* for each edge $e \in E_G$, we have $m_G(e) = m_H(g_E(e))$.
- *Preservation of defined node labels:* for each node of defined label $v \in \mathrm{dom}(l_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(l_H)$ and $l_G(v) = l_H(g_V(v))$.
- *Preservation of defined rootedness:* for each node of defined rootedness $v \in \mathrm{dom}(p_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(p_H)$ and $p_G(v) = p_H(g_V(v))$. □

We omit the name of the morphism if it is not needed. Note that node labels and rootedness are treated analogously.

Definition 2.6 (Morphism Properties)**.** Let $g\colon G \to H$ be a morphism.

- We say g is *injective/surjective* if both g_V and g_E are injective/surjective.
- We say g is *bijective* if it is injective and surjective.
- We say morphisms $g\colon G \to H$ and $f\colon I \to J$ are *equal*, denoted by $g = f$, if $G = I$, $H = J$, $g_V = f_V$, and $g_E = f_E$.
- We call $\mathrm{id}_G\colon G \to G$ where $\mathrm{id}_G = \langle \mathrm{id}_{V_G}, \mathrm{id}_{E_G}\rangle$ the *identity (graph) morphism.* □

Definition 2.7 (Graph Isomorphism)**.** A (graph) morphism $g\colon G \to H$ is a *(graph) isomorphism* if it is a bijective morphism that preserves undefined labels and undefined rootedness, i.e. for each $v \in V_G$, if $l_G(v) = \perp$ then $l_H(g_V(v)) = \perp$, and if $p_G(v) = \perp$ then $p_H(g_V(v)) = \perp$. If such an isomorphism exists, we call G and H *isomorphic*, denoted by $G \cong H$. □

Definition 2.8 (Graph Inclusion, Subgraph)**.** A (graph) morphism $g: H \to G$ is a *(graph) inclusion* if for each $v \in V_H$ and each edge $e \in E_H$, we have $g_V(v) = v$ and $g_E(e) = e$.

If g is an inclusion, we may use the notation $g: H \hookrightarrow G$, and we say H is a *subgraph* of G, denoted as $H \subseteq G$. □

Note that inclusions are injective due to their definition. Furthermore note that, since morphisms only preserve defined labels and defined rootedness, nodes that are labelled or have defined rootedness may be unlabelled or of undefined rootedness in a subgraph. Also, we have that $G \subseteq H$ and $H \subseteq G$, if and only if $G \cong H$.

Definition 2.9 (Operations on Morphisms)**.** Let $g: G \to H$ be a morphism.

- Given morphisms $g: G \to H$ and $f: H \to I$, we define their *composition* $f \circ g: G \to I$ as $f \circ g = \langle f_V \circ g_V, f_E \circ g_E \rangle$. We sometimes omit the composition's name by writing $G \to H \to I$.
- Given a morphism $f: G \to H$, and a subgraph I of G with a strict inclusion $g: I \hookrightarrow G$, we define the *restriction* of f to I as $f|_I: I \to H$ where $f|_I = \langle f_V|_{V_I}, f_E|_{E_I} \rangle$. □

Note that if two functions are morphisms, isomorphisms, or inclusions then so is their composition.

Furthermore note that restriction preserves the properties of being a morphism, or an inclusion. It does not preserve isomorphisms, but it does preserve injectivity.

2.1.3 Rules

The principal, most basic graph computation construct we use is a rule, which has a left-hand side graph L and a right-hand side graph R. When a rule is applied to a graph G, we find a morphism from L to G, and transform the image of L in G so it resembles R.

Definition 2.10 (Rule)**.** A *rule* $\langle L \hookleftarrow K \hookrightarrow R \rangle$ consists of graph K with unlabelled nodes of undefined rootedness and no edges, totally labelled graphs with defined rootedness L and R, and inclusions $K \hookrightarrow L$ and $K \hookrightarrow R$. We call L the *left-hand side*, R the *right-hand side*, and K the *interface.* □

Note that K consists of unlabelled nodes of undefined rootedness. This is to enable the inclusions to map these nodes to rooted or unrooted nodes with any label. Without this ability, labels and rootedness of nodes would have to be the same in L and R, making it impossible for a rule to relabel, or to change the rootedness status of a node, which is an ability we want GP 2 to have. A discussion of this can be found in Subsection 2.2.4. There is no need to change labels of edges because rules can delete and re-insert an edge of a different label between the same pair of nodes.

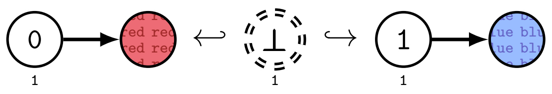


Figure 2.3: Example of a rule

An example of a rule can be seen in Figure 2.3. It uses GP 2 labels (numbers `0` and `1`, as well as red and blue colours), which are defined in Subsection 2.2.1. Nodes that are drawn without a label are considered to have the GP 2 label `empty`. Furthermore, nodes with small numbers next to them identify nodes in the interface. These numbers also show how interface nodes are mapped by the inclusions. Because of this, we can omit the interface K when drawing a rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$, in which case we depict the rule as $L \Rightarrow R$.

2.1.4 Basic Category Theory Notions for Graphs

In order to apply a rule to a host graph and get a unique outcome, we use double-pushouts from category theory, and properties thereof, as described in [34]. Categories are defined as follows.

Definition 2.11 (Category)**.** A *category* $\mathcal{C} = \langle \mathcal{O}, \mathcal{M}, \circ, id\rangle$ consists of

- a class of *objects* $\mathcal{O}$,
- a set of *morphisms* $\mathcal{M}(A, B)$ for each pair of objects $A, B \in \mathcal{O}$,
- a *composition* function $\circ_{A,B,C}\colon \mathcal{M}(B, C) \times \mathcal{M}(A, B) \to \mathcal{M}(A, C)$ for all objects $A, B, C \in \mathcal{O}$, and
- an *identity morphism* $id_A \in \mathcal{M}(A, A)$ for each object $A \in \mathcal{O}$.

such that the following holds.

- *Associativity*: For all objects $A, B, C, D \in \mathcal{O}$, and morphisms $f \in \mathcal{M}(A, B)$, $g \in \mathcal{M}(B, C)$, and $h \in \mathcal{M}(C, D)$, we have $h \circ (g \circ f) = (h \circ g) \circ f$.

- *Identity*: For all objects $A, B \in \mathcal{O}$ and morphisms $f \in \mathcal{M}(A, B)$, we have $f \circ id_A = f$ and $id_B \circ f = f$. $\square$

Instead of stating results for categories in general, we restrict ourselves to the category of graphs. Our objects are graphs as we have defined them. Morphisms are graph morphisms. Composition is composition of graph morphisms. The identity morphism is the identity graph morphism. Associativity and identity properties follow from those of function composition and identity.

Definition 2.12 (Pushout, Pullback, Natural Pushout)**.** Let A, B, C, and D be graphs with morphisms $A \to B$, $B \to D$, $A \to C$, and $C \to D$, as illustrated in Figure 2.4. Assume square (1) commutes ($A \to B \to D = A \to C \to D$).

We say square (1) is a *pushout* if for all graphs D', and all morphisms $B \to D'$ and $C \to D'$ such that $A \to B \to D' = A \to C \to D'$ (see Figure 2.4a), there is a unique morphism $D \to D'$ such that triangles (2) and (3) commute ($B \to D' = B \to D \to D'$ and $C \to D' = C \to D \to D'$).

We say square (1) is a *pullback* if for all graphs A', and all morphisms $A' \to B$ and $A' \to C$ such that $A' \to B \to D = A' \to C \to D$ (see Figure 2.4b), there is a unique morphism $A' \to A$ such that triangles (2) and (3) commute ($A' \to B = A' \to A \to B$ and $A' \to C = A' \to A \to C$).

We call a square (1) a *natural pushout* if it is both a pushout and a pullback.

We call B and C *pushout/pullback complements.* $\square$

$$\begin{tikzcd}
 & B \arrow[dr] \arrow[drr, bend left] \arrow[drr, phantom, "(2)"] & & \\
A \arrow[ur] \arrow[dr] & (1) & D \arrow[r, dashed] & D' \\
 & C \arrow[ur] \arrow[urr, bend right] \arrow[urr, phantom, "(3)"] & &
\end{tikzcd}$$

(a) A pushout

$$\begin{tikzcd}
 & & B \arrow[dr] & \\
A' \arrow[r, dashed] \arrow[urr, bend left] \arrow[urr, phantom, "(2)"] \arrow[drr, bend right] \arrow[drr, phantom, "(3)"] & A \arrow[ur] \arrow[dr] & (1) & D \\
 & & C \arrow[ur] &
\end{tikzcd}$$

(b) A pullback

Figure 2.4: A pushout and a pullback

Intuitively, pushouts can be seen as gluings. Graph D is obtained by combining B and C, glued (or merged) along the nodes and edges they share with A. Pullbacks on the other hand can be seen as intersections. Graph A is obtained by taking the parts of B and C that are glued (or merged) in D, and glue (or merge) them in the same way.

Definition 2.13 (Natural Double-pushout)**.** Let the squares in Figure 2.5 be natural pushouts. Then we call the diagram a *natural double-pushout.* □

$$\begin{tikzcd}
L \arrow[d] \arrow[rd, phantom, "\text{(NPO)}"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[rd, phantom, "\text{(NPO)}"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 2.5: A natural double-pushout

Natural double-pushouts allow us to relabel nodes, which is explained in Subsection 2.2.4. This approach to relabelling with pushouts is introduced in [43]. Since node labelling and rootedness are modelled analogously, the same approach can be extended to changing rootedness, as introduced in [21].

2.1.5 Direct Derivation

We use natural double-pushouts as depicted in Figure 2.5 to apply a rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$ to a graph G, resulting in graph H. Note that when we apply a rule, we want an injective morphism $g\colon L \to G$ that matches L with part of G. In order to apply a rule, we use a construct called direct derivation. An example of a direct derivation can be seen in Figure 2.6.

Definition 2.14 (Direct Derivation, Derivation)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ be a rule, G a totally labelled graph, and $L \to G$ an injective morphism. Let D and H be graphs such that the squares in Figure 2.5 is a natural double-pushout. Then $G \Rightarrow_r H$ is a *direct derivation.* We write $G \Rightarrow H$ if the name of the rule is not relevant.

Derivation $\Rightarrow^*$ is the reflexive (with respect to isomorphism) and transitive closure of $\Rightarrow$, i.e. $G \Rightarrow^* H$ if either $G \cong H$, or there is a finite sequence of direct derivations $G \Rightarrow \ldots \Rightarrow H$. □

We want direct derivation to exist and to be unique up to isomorphism. Uniqueness is in part due to the fact that the double-pushout is natural, contributing to the pushout complement being unique. Figure 2.7 shows two distinct pushout complements. Only the pushout on the left is natural.

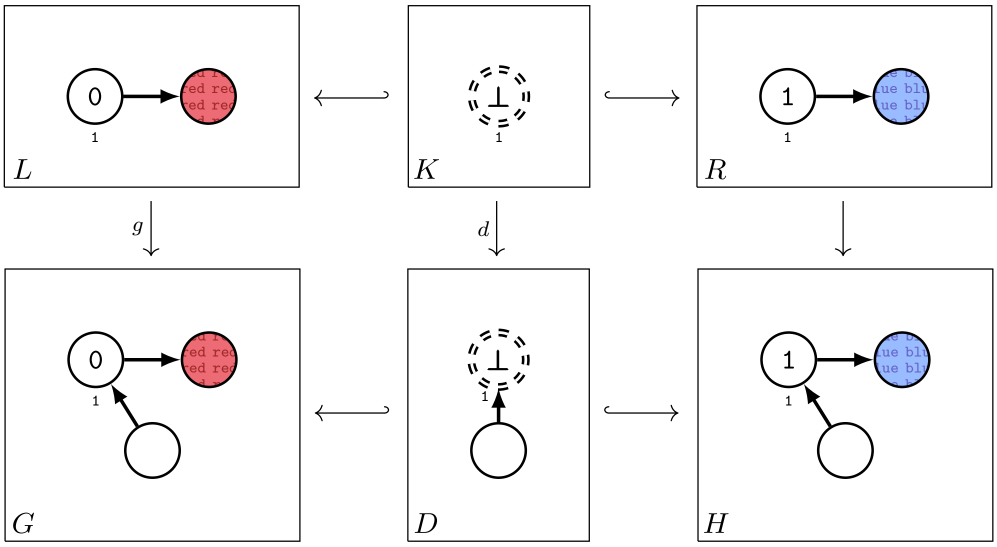


Figure 2.6: Example of a direct derivation

In a direct derivation (Figure 2.10), since L is totally labelled, and K is completely unlabelled, and has no edges, the double-pushout being natural forces nodes in $d(K) \subseteq D$ to be unlabelled.

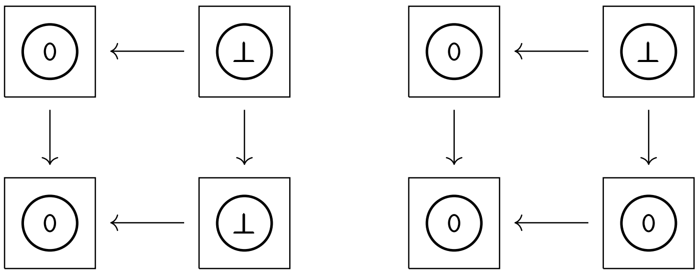


Figure 2.7: Two distinct pushout complements

Note that, while nodes drawn without labels in K are truly unlabelled, nodes drawn without labels in L, G, R, and H are considered labelled with the "empty label". A detailed definition of labels is given in Subsection 2.2.1. In graph D, unlabelled nodes in $d(K) \subseteq D$ are considered truly unlabelled, and any other unlabelled nodes are considered to have the empty label.

This distinction is not needed for edges. Since K has no edges by definition of a rule, D can use edge labels of G without breaking its double-pushout.

2.1.6 Construction of Direct Derivation

We can show existence of direct derivation by showing its construction, similar to that found in [43]. One of the prerequisites for this construction is called the "dangling condition". If a rule deletes a node in G, there might be additional edges that are left without a source or target (dangling), resulting in an invalid graph. Hence we require the dangling condition, forbidding incident edges in G to nodes that get deleted by the rule (nodes not in the interface K).

Definition 2.15 (Dangling Condition)**.** Given a rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$ and a graph G, an injective morphism $g\colon L \to G$ satisfies the *dangling condition* if no edge in $E_G - g_E(E_L)$ is incident to a node in $g_V(V_L - V_K)$. □

Direct derivation is constructed as follows. First D is obtained by deleting nodes and edges in G that are part of the left-hand side L, but not of the interface K. Nodes that are also in K have their labels removed, and get undefined rootedness. Then H is obtained by gluing graphs D and R such that the parts of them that stem from K get merged. The nodes that are merged get their labels and rootedness from R since their counterparts in K and D are unlabelled and have undefined rootedness.

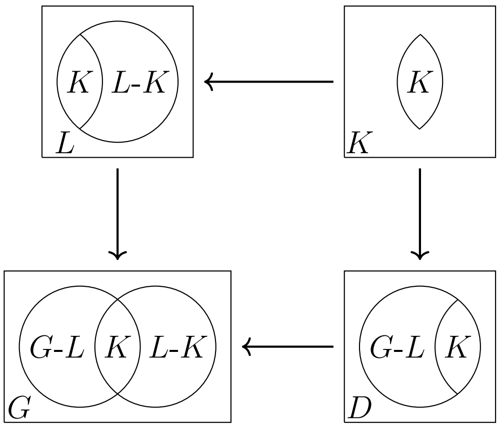


Figure 2.8: Abstract illustration of the deletion construction

Proposition 2.16 (Deletion with Dangling Condition)**.** *Let $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ be a rule, G a graph, and $g\colon L \to G$ an injective morphism that satisfies the dangling condition. Construct a graph D as follows.*

- $V_D = V_G - g_V(V_L - V_K)$.

- $E_D = E_G - g_E(E_L - E_K)$.
- $s_D = s_G|_{E_D}$, $t_D = t_G|_{E_D}$, $m_D = m_G|_{E_D}$.
- $l_D(v) = \begin{cases} \bot & \textit{if there is } u \in V_K \textit{ such that } g(u) = v, \\ l_G(v) & \textit{otherwise.} \end{cases}$
- *p_D is analogous to l_D.*

Then the square consisting of graphs K, L, D, and G and the morphisms between them is a natural pushout.

The proof of Proposition 2.16 follows from arguments about the equivalent construction in [43], treating roots the same as node labels.

For the next step, we need the *disjoint union*, which we denote by $+$. It is simply a matter of renaming all elements to be distinct and taking the union. This can be done with cross products as follows for instance. For sets A and B, let $A + B = (A \times \{0\}) \cup (B \times \{1\})$.

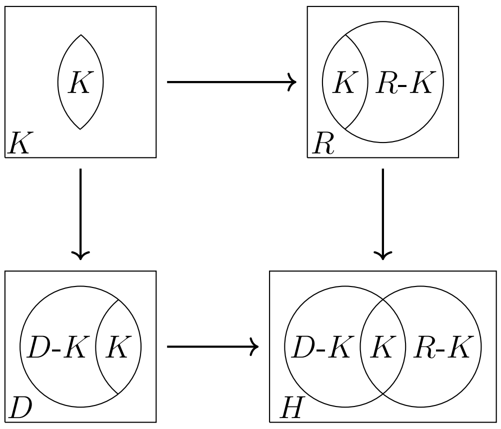


Figure 2.9: Abstract illustration of the gluing construction

Proposition 2.17 (Gluing). *Let $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ be a rule, G a graph, $g\colon L \to G$ an injective morphism that satisfies the dangling condition, and D a graph constructed by the deletion described in Proposition 2.16. Construct graph H as follows.*

- $V_H = V_D + (V_R - V_K)$.
- $E_H = E_D + (E_R - E_K)$.

- $s_H(e) = \begin{cases} s_D(e) & \textit{if } e \in E_D, \\ d_V(s_R(e)) & \textit{if } e \in E_R - E_K \textit{ and } s_R(e) \in V_K, \\ s_R(e) & \textit{otherwise.} \end{cases}$
- *t_H is analogous to s_H.*
- $l_H(v) = \begin{cases} l_R(v) & \textit{if } v \in V_R, \\ l_D(v) & \textit{otherwise.} \end{cases}$
- *m_H and p_H are analogous to l_H.*

Then the square consisting of graphs K, R, D, and H and the morphisms between them is a natural pushout.

Again, the proof of Proposition 2.17 is analogous to [43].

2.1.7 Existence and Uniqueness of Direct Derivation

Theorem 2.18 formalises existence and uniqueness of direct derivation, assuming the right square is only a pushout. In [43], it is shown that the right square of a direct derivation is a pullback as well. However, with the introduction of roots, that fact no longer holds. Since we now require that non-roots in a rule cannot match with roots in a host graph [17], we require the right square to be a pullback again, making rules invertible.

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "\text{(NPO)}"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, "d"'] \arrow[dr, phantom, "\text{(NPO)}"] & R \arrow[d] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

Figure 2.10: A direct derivation

Theorem 2.18 (Existence and Uniqueness of Direct Derivation)**.** *Consider rule $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$, graph G, and injective morphism $g\colon L \to G$ as in Figure 2.10. Then there exists a direct derivation $G \Rightarrow_r H$ if and only if g satisfies the dangling condition.*

If $G \Rightarrow_r H$ does exist, D and H are unique up to isomorphism, i.e. if D' and H' form another direct derivation $G \Rightarrow_r H'$, then $D \cong D'$ and $H \cong H'$.

Furthermore, since G is totally labelled, H is totally labelled.

Existence is given by the deletion and gluing constructions from Propositions 2.16 and 2.17.

The proof of Proposition 2.18 can be found in [43]. The graphs in that paper do not have rootedness. However, since rootedness is modelled analogously to node labels, reasoning about them is also analogous [17].

2.2 Attributed Graph Transformation

For graph programming in GP 2, we want rules to be more expressive than the graph transformation rules we have seen before. We extend rules so they allow for variables in labels and for application conditions. Natural double-pushouts are still the underlying framework.

2.2.1 Labels

The labels we use follow the subtype hierarchy in Figure 2.11. They consist of single characters `char`, sequences of characters `string`, integers `int`, either strings or integers `atom`, and lists of atoms `list`. Lists can be empty, and atoms are considered to be lists of length 1. Similarly, every character is considered a string of length 1.

$$\begin{tikzcd}
& \texttt{list} & \\
& \texttt{atom} \arrow[u, phantom, sloped, "\subseteq"] & \\
\texttt{int} \arrow[ur, phantom, sloped, "\subseteq"] & & \texttt{string} \arrow[ul, phantom, sloped, "\subseteq"] \\
& & \texttt{char} \arrow[u, phantom, sloped, "\subseteq"]
\end{tikzcd}$$

Figure 2.11: GP 2 subtype hierarchy

GP 2 labels are generated with context-free string grammars. The grammars' abstract syntaxes [8] are given in Figures 2.13 and 2.14. We call the syntaxes abstract because the grammars they define are ambiguous. The integer expressions `5*3+4` for instance can be generated in multiple ways. This ambiguity can be resolved using parentheses. For parsing, we use unambiguous concrete syntaxes that can be found in [8]. We use abstract syntaxes because it facilitates associating types to labels. There are two grammars because we distinguish between host graphs, which are the graphs we modify, and rule graphs, which help us describe these modifications. Intermediate graphs are used as an intermediate step to facilitate relabelling.

Definition 2.19 (Intermediate, Host, and Rule Graph)**.** An *intermediate graph* is a graph over the set of labels generated by the grammar in Figure 2.13.

A *host graph* is a totally labelled intermediate graph.

A *rule graph* is a totally labelled graph over the set of labels generated by the grammar in Figure 2.14. □

Note that labels generated by the grammar in Figure 2.13 are a subset of labels generated by the grammar in Figure 2.14. Hence host graphs can be considered a subset of rule graphs. Host graphs are also a subset of intermediate graphs by definition.

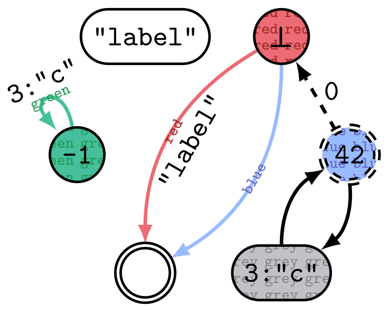


Figure 2.12: Example of a host graph

An example of how a host graph is represented on paper can be seen in Figure 2.12. Marks are represented using corresponding colours. The name of the mark is written within the colours for accessibility purposes. Note that the `grey` mark is reserved for nodes and the `dashed` mark for edges. Label expressions are drawn inside their node or next to their edge. Items with the `empty` label are drawn without label (i.e. blank), the `empty` keyword is reserved for textual representation only. Unlabelled nodes (undefined labelling function) are drawn using ⊥ in order to avoid confusion.

In the grammars of Figures 2.13 and 2.14, terminals are represented in a typewriter font within single quotation marks. Nonterminals start with capital letters. The ones ending in 'Label' are the starting nonterminals. The rule graph label syntax reuses the definitions of HostMark, Character, and Digit from the host graph label syntax. NodeID is a set containing distinct identifiers for each node in a given graph. More details on these can be found in the concrete syntax of GP 2 in [8]. The sets ending in 'Var'

```
HostLabel    ::=  HostList [HostMark]
HostList     ::=  ‘empty’ | HostAtom {‘:’ HostAtom}
HostAtom     ::=  HostInteger | HostString
HostInteger  ::=  [‘-’] Digit {Digit}
HostString   ::=  ‘"’{Character}‘"’ | HostChar
HostChar     ::=  ‘"’Character‘"’
HostMark     ::=  ‘red’ | ‘green’ | ‘blue’ | ‘grey’ | ‘dashed’
Character    ::=  Printable characters except for ‘"’
Digit        ::=  ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

Figure 2.13: Abstract syntax of host graph labels

```
RuleLabel    ::=  RuleList [RuleMark]
RuleList     ::=  LVar | ‘empty’ | RuleAtom | RuleList ‘:’ RuleList
RuleAtom     ::=  AVar | RuleInteger | RuleString
RuleInteger  ::=  IVar | [‘-’] Digit {Digit} | ‘(’RuleInteger‘)’
                  | RuleInteger (‘+’ | ‘-’ | ‘*’ | ‘/’) RuleInteger
                  | (‘indeg’ | ‘outdeg’) ‘(’NodeID‘)’
                  | ‘length’ ‘(’(LVar | AVar | SVar)‘)’
RuleString   ::=  SVar | RuleChar | ‘"’{Character}‘"’
                  | RuleString ‘.’ RuleString
RuleChar     ::=  CVar | ‘"’Character‘"’
RuleMark     ::=  HostMark | ‘any’
```

Figure 2.14: Abstract syntax of rule graph labels

contain typed variables, which we later elaborate on. For rule graph labels, we associate types to strings of terminals by naming them as follows.

- *(List) expression*: string of terminals generated by RuleList.
- *Atom expression*: expression generated by RuleAtom.
- *Integer expression*: atom expression generated by RuleInteger.
- *String expression*: atom expression generated by RuleString.
- *Character expression*: string expression generated by RuleChar.

Note that in their grammar, the nonterminals given a type above call each other in a way that matches the subtype hierarchy given in Figure 2.11. Associating types to host graph labels is analogous.

Compared to those of host graphs, the labels of rule graphs allow for more general expressions and variables which are intended to match with a host graph label. The functions `indeg` and `outdeg` return the indegree and outdegree of a node respectively. The `length` function returns the length of a variable. The `any` mark matches with either of the marks in a host graph, but not with a lack of mark, i.e. `any` matches either `red`, `green`, `blue`, or `grey`.

2.2.2 Rule Schemata

Since rule schemata labels contain variables, there can be some ambiguity. For instance with two concatenated string variables `x.y`, there can be multiple variable assignments that result in the same string. Lists have the same issue.

Definition 2.20 (Simple Label/Expression)**.** We say a rule graph label is *simple* if its expression is simple. An expression is *simple* if it has the following properties

- It contains no length, degree, or arithmetic operators, except for the unary minus.
- It contains at most one list variable.
- Every string expression it contains has at most one string variable. □

Definition 2.21 (Rule Schema)**.** A *rule (schema)* $\langle L \hookleftarrow K \hookrightarrow R\rangle$ is a rule where L and R are rule graphs, and K an intermediate graph such that:

- All labels in L are simple.
- All variables occurring in R must also occur in L.
- All nodes in R with the `any` mark must be the image of an interface node whose image in L also has the `any` mark.

A *(conditional) rule (schema)* $\langle L \hookleftarrow K \hookrightarrow R, c\rangle$ consists of a rule schema and a *(application) condition* c generated by the grammar in Figure 2.16. We require that all variables occurring in c must also occur in L. □

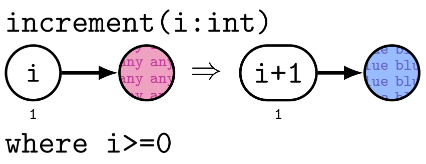


Figure 2.15: Example of a conditional rule schema

We omit 'conditional' or 'schema' when talking about concrete GP 2 programs since we only need this distinction for the theoretical foundations of rule application. We sometimes consider rule schemata without condition to be conditional rule schemata whose conditions are always true (using the condition `1=1` for instance). An example of how we draw a conditional rule schema can be seen in Figure 2.15. The rule is named `increment`. The integer variable `i` is declared right after the rule name. The right-hand side contains the arithmetic expression `i+1`. The `any` mark is represented with a pink/magenta colour. The condition requires `i` to be non-negative.

```
Condition ::= (int | char | string | atom) '('(LVar | AVar | SVar)')'
            | RuleList ('=' | '!=') RuleList
            | RuleInteger ('>' | '>=' | '<' | '<=') RuleInteger
            | edge '(' NodeID ',' NodeID [',' RuleLabel] ')'
            | not Condition
            | Condition (and | or) Condition
            | '(' Condition ')'
```

Figure 2.16: Abstract syntax of conditions

The rule application condition is a Boolean statement depending on properties of the host graph. The left-hand side of a rule is only matched if the condition is fulfilled. The functions `int`, `char`, `string`, and `atom` are *subtype predicates* that check whether a given variable is of a certain type. The `edge` function checks whether there is an edge (potentially with a given label) between two given nodes. We have equality comparisons for list expressions, inequality comparisons for integer expressions, and logical operators.

In order to apply a rule schema, we need a way to map rule graphs with variable labels to host graphs with constant labels. We use variable assignments to assign variables to values, and premorphisms to map one graph into another while ignoring labels.

Definition 2.22 (Variable Assignment)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema. Let LVar_r, AVar_r, IVar_r, SVar_r, and CVar_r be the set of variables used in r of the corresponding type. A *variable assignment* is a family of functions $\alpha = (\alpha_{\mathrm{X}})_{X\in\{\mathrm{L,A,I,S,C}\}}$, where $\alpha_{\mathrm{L}}\colon \mathrm{LVar}_r \to \mathrm{HostList}$, $\alpha_{\mathrm{A}}\colon \mathrm{AVar}_r \to \mathrm{HostAtom}$, $\alpha_{\mathrm{I}}\colon \mathrm{IVar}_r \to \mathrm{HostInteger}$, $\alpha_{\mathrm{S}}\colon \mathrm{SVar}_r \to \mathrm{HostString}$, and $\alpha_{\mathrm{C}}\colon \mathrm{CVar}_r \to \mathrm{HostChar}$. We often omit the subscript from α since only one function is applicable to a given variable. □

Definition 2.23 (Graph Premorphism)**.** Let G and H be graphs. A *(graph) premorphism* $g\colon G \rightharpoonup H$ is a pair $g = \langle g_V,\, g_E\rangle$ of functions $g_V\colon V_G \to V_H$ and $g_E\colon E_G \to E_H$ such that the following holds.

- *Preservation of sources and targets:* for each edge $e \in E_G$, we have $g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$.
- *Preservation of defined rootedness:* for each node of defined rootedness $v \in \mathrm{dom}(p_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(p_H)$ and $p_G(v) = p_H(g_V(v))$. □

We can now instantiate a rule schema. This means that variables and `any` marks are substituted according to a variable assignment and a premorphism.

Definition 2.24 (Rule Schema Instance)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R,\, c\rangle$ be a conditional rule schema, $g\colon L \rightharpoonup G$ an injective premorphism, where G is a host graph, and α a variable assignment. A *rule schema instance* is a tuple $r^{g,\alpha} = \langle L^{g,\alpha} \hookleftarrow K \hookrightarrow R^{g,\alpha},\, c^{g,\alpha}\rangle$, where $L^{g,\alpha}$, $R^{g,\alpha}$, and $c^{g,\alpha}$ are obtained from L, R, and c by substituting variables with their assignments, replacing the `any` mark with the corresponding mark in G, and evaluating any arithmetic, list, and logic operations, as well as statements about the host graph G (i.e. `indeg`, `outdeg`, and `edge`), all with respect to the mapping g. □

Note that since all variables have been substituted, and all operations have been evaluated, $L^{g,\alpha}$ and $R^{g,\alpha}$ can be considered either host, rule, or intermediate graphs.

2.2.3 Rule Schema Application

In order to apply a conditional rule schema $r = \langle L \hookleftarrow K \hookrightarrow R,\, c\rangle$ to a host graph G, we need to find an injective premorphism $g\colon L \rightharpoonup G$ and a variable assignment α such that the Boolean statement $c^{g,\alpha}$ is true, and the labels of L match the labels of G. Specifically, the latter means that we have a morphism $g\colon L^{g,\alpha} \to G$, which is obtained from the premorphism g by swapping out nodes and edges of L for those of the instance $L^{g,\alpha}$. We give both the morphism and the premorphism the same name g since they map nodes and edges in the same way.

Furthermore, we need g to satisfy the dangling condition in order to preserve the structural integrity of the host graph (otherwise edges may be left dangling, i.e. with no adjacent node), and hence ensure the applicability of the rule.

Definition 2.25 (Match)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema, G a host graph, $g\colon L \rightharpoonup G$ an injective premorphism, and α a variable assignment. We say $\langle g, \alpha\rangle$ is a *match* for r in G if the following holds.

- $g\colon L^{g,\alpha} \to G$ is an injective morphism that satisfies the dangling condition.
- $c^{g,\alpha}$ evaluates to true. □

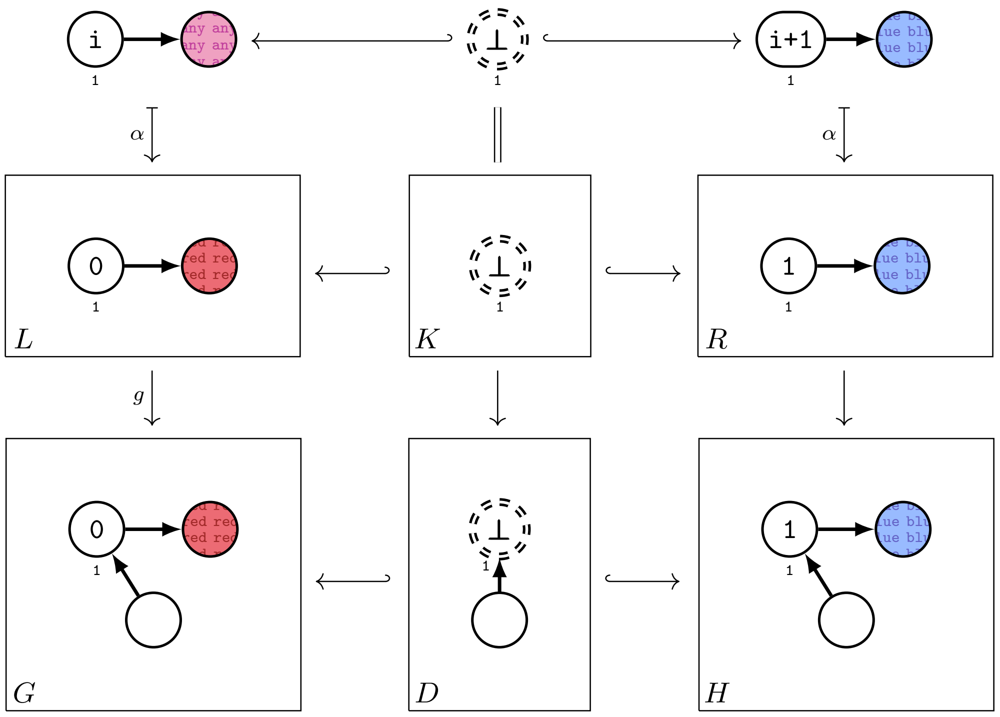


Figure 2.17: Example of a rule application

We then use natural double-pushouts from the previous section on the instantiated rule, injective morphism g, and graph G to apply the rule to G. An example of such a rule application based on previous rules and their application can be seen in Figure 2.17.

Definition 2.26 (Derivation Using Attributed Graphs)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R,\ c\rangle$ be a conditional rule schema, G a host graph, and $\langle g, \alpha\rangle$ a match for r in G. We say $G \Rightarrow_r H$ if $G \Rightarrow_{r^{g,\alpha}} H$ (in the way described in the previous section with a natural double-pushout, as illustrated in 2.18).

We define *derivation* $\Rightarrow^*$ using attributed graphs analogously to derivation in Definition 2.14 as the reflexive (with respect to isomorphism) and transitive closure of $\Rightarrow$. □

$$\begin{tikzcd}
L \arrow[d, mapsto, "\alpha"'] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, equal] & R \arrow[d, mapsto, "\alpha"'] \\
L^{g,\alpha} \arrow[d, "g"'] \arrow[dr, phantom, "\text{(NPO)}"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "\text{(NPO)}"] & R^{g,\alpha} \arrow[d] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

Figure 2.18: A direct derivation

Based on the results from the previous section, we get uniqueness of this direct derivation as well.

Proposition 2.27 (Uniqueness of Matching [62]). *Let r* $= \langle L \hookleftarrow K \hookrightarrow R,\ c\rangle$ *be a conditional rule schema and G a host graph. Then there is at most one instance of r for which there is a match.*

The proof of Proposition 2.27 can be found in [62].

2.2.4 Relabelling and Changing Rootedness

In traditional double-pushout graph transformation, only totally labelled graphs are used. Since morphisms preserves labels, this means there's no way to change the label of a node. In order to enable relabelling, we allow for partially labelled graphs, but require the double-pushout to be natural. This allows for a unique direct derivation [43].

Figure 2.19 shows a direct derivation that relabels a node. By assumption, L and G are totally labelled, while K is completely unlabelled. If we only require (1) to be a pushout, the pushout complement D may be either labelled or unlabelled. However, we require (1) to be a pullback as well, forcing D to be unlabelled. Then H can be uniquely constructed on the assumption that (2) is a pushout.

Figure 2.20 shows a derivation that roots an unrooted node. Morphisms like g preserving defined rootedness means we cannot match the unrooted L with a rooted G, which could lead to unexpected behaviour of the rule, i.e. instead of rooting an unrooted node, a rooted node stays rooted.

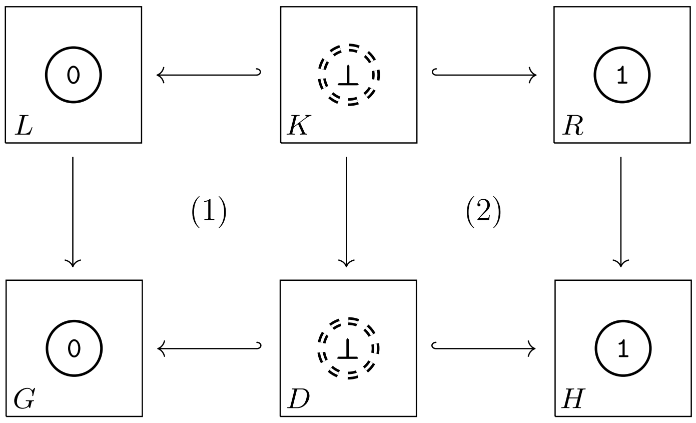


Figure 2.19: A direct derivation that relabels

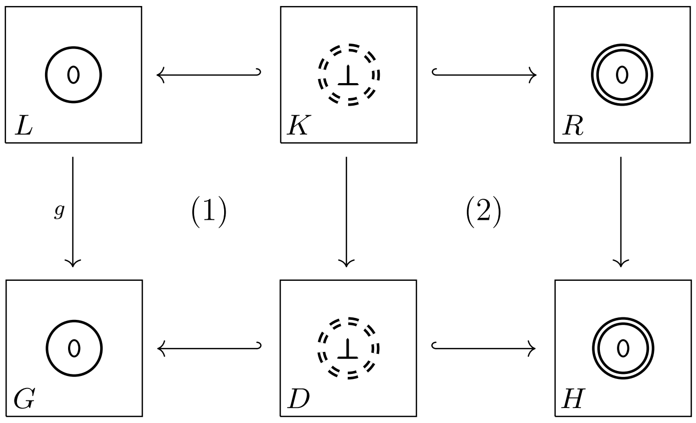


Figure 2.20: A direct derivation that roots a node

Similarly to the situation with labels, since K has undefined rootedness and L and G are unrooted, (1) being a pullback forces D to have undefined rootedness. Again, H can be uniquely constructed on the assumption that (2) is a pushout.

The reasons why roots behave as expected are morphisms that preserve defined rootedness, the fact that some nodes can have undefined rootedness, and the fact that the double-pushout is natural [20].

2.3 GP 2 Programs: Syntax and Semantics

Now that we have defined rules, the fundamental operations that allow GP 2 to transform graphs, let us define programs. The main constructs of GP 2 programs are loops and conditional branching statements. They allow us to implement entire graph algorithms.

2.3.1 Syntax

Figure 2.21 contains the abstract syntax of a GP 2 program in Extended Backus-Naur form. It is abstract in that it is used for writing programs on paper and designed to be interpreted by humans. Programs given to the compiler for execution use the concrete syntax, which is designed to help the compiler interpret programs, can be found in an appendix in [8].

```
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 2.21: GP 2 Program Syntax

A GP 2 program consists of different types of declarations. MainDecl is a sequence of commands that are executed when running the program. This sequence can contain procedures, which are named command sequences that are declared in ProcedureDecl. When a procedures name is called, its command sequence is executed. Procedures can have local declarations, which are declarations of rules and procedures that are only for use within the scope of their containing procedure. Rules are declared in RuleDecl. Its definition is omitted from the figure since we do not use the textual representation of rules in this thesis, only the graphical representation, as seen in the previous section (Figure 2.15 for instance). Note that each rule comes with a name denoted as RuleID.

Command sequences consist of commands separated by semicolons. Commands can be either an `if` statement (with an optional `else` branch), a `try` statement (with optional `then` and `else` branches), a command sequence in brackets, a loop denoted by ‘`!`’, an `or` statement, or a simple command.

2.3.2 The Behaviour of Command Sequences

As described by the syntax, GP 2 programs and procedures consist of command sequences. Calling a command sequence on a host graph can have three different outcomes. Either it results in another host graph (possibly transformed by rules), it results in failure (if a rule is inapplicable or the `fail` statement is called), or it does not terminate.

Before we introduce the formal semantics, let us give an idea of how the different statements within a command sequence behave.

The call of a rule set $\{r_1, \ldots, r_n\}$ nondeterministically applies one of the rules whose left-hand graph matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution in case C succeeds.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body is entered for the last time. The `break` command inside a loop terminates that loop with the current graph and transfers control to the command following the loop.

A program P `or` Q non-deterministically chooses to execute either P or Q, which can be simulated by a rule-set call and the other commands [58]. The commands `skip` and `fail` can also be expressed by the other commands: `skip` is equivalent to an application of the rule $\emptyset \Rightarrow \emptyset$ (where $\emptyset$ is the empty graph) and `fail` is equivalent to an application of $\{\}$ (the empty rule set).

2.3.3 Example Program

Let us look at an example. The program in Figure 2.22 shows a GP 2 program that produces a node colouring, i.e. every node gets assigned an integer (each integer represents a different colour) such that no two adjacent nodes have the same integer.

```
Main = mark!; init!; inc!
```

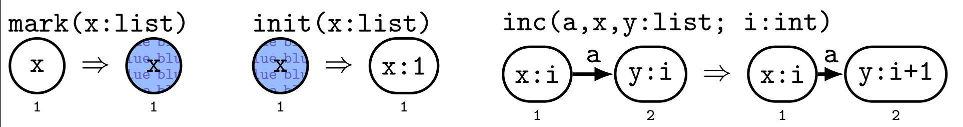


Figure 2.22: A GP 2 program producing a node colouring

The loop `mark!` calls the rule `mark` for as long as possible, marking every node blue. A node being blue means it has not yet been assigned an integer. The loop `init!` appends the integer `1` to the label of every node. Then `inc!` makes sure that, as long as there are adjacent nodes with the same integer, one integer gets incremented, producing a colouring.

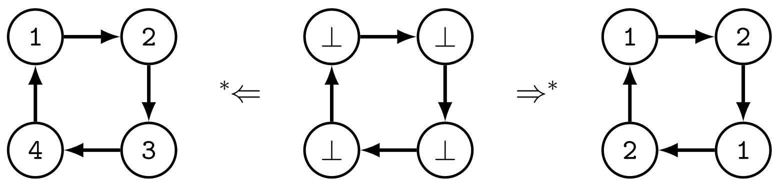


Figure 2.23: Two possible results from applying the colouring program

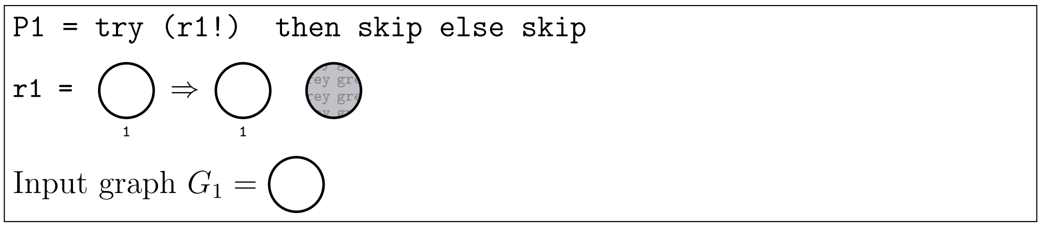
Note that GP 2 programs are nondeterministic, specifically concerning rule matching. Depending on how the rules called in `inc!` match and in which order, different colourings may be produced, as can be seen in Figure 2.23. The two results differ because of where the rule `inc` matched.

In order to describe different outcomes of a GP 2 program and how we get there, we use a formal semantics. We define a transition sequence on program states that break the execution of programs down into steps. Using that, a semantic function lists the possible outcomes of executing a given program on a given graph.

2.3.4 Two Semantics

There are two operational semantics for GP 2. An operational semantics is *small-step* if atomic computation steps during program execution correspond to transition steps between states in a semantics, meaning that the language can be implemented by translating the transition steps into corresponding code. The original semantics is defined by both small-step and big-step inference rules [60]. However there is also a truly small-step operational semantics for GP 2 which, in particular, accurately models diverging computations, introduced by this thesis' author in [25].

While the original semantics (Figure 2.27) has small-step elements, the branching and loop constructs are not small-step. This can lead to the semantic transition sequence *blocking* or *getting stuck* [55], i.e. reaching a configuration which is neither a graph nor the failure state, such that no inference rule is applicable.



```
P1 = try (r1!)  then skip else skip
```

Figure 2.24: The GP 2 program `P1` and input graph G_1

To illustrate this situation, consider the program `P1` in Figure 2.24 applied to graph G_1 from the same figure. Remember the statement `r1!` means that the rule `r1` is called until it is no longer applicable. The `try` statement attempts to evaluate `r1!` but will neither branch to the `then` nor the `else` part because the loop `r1!` diverges on G_1. In the original semantics, `try` statements are handled with the following inference rules:

$$[\text{try}'_1]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle P,\, H\rangle}$$

$$[\text{try}'_2]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle Q,\, G\rangle}$$

The premises of these inference rules are that the conditional part C of a `try` statement applied to host graph G results in either a graph H or failure, which determines whether P or Q is called. If $\langle C, G\rangle$ diverges (does not terminate) however, neither rule applies. Since there are no other `try` rules, the transition sequence gets stuck.

The small-step semantics handles `try` statements with the following rules:

$$[\text{try}_1]\ \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, S\rangle \to \langle \text{TRY}(C, P, Q),\, \text{push}(\text{top}(S),\, S)\rangle$$

$$[\text{try}_2]\ \frac{\langle C,\, S\rangle \to \langle C',\, S'\rangle}{\langle \text{TRY}(C, P, Q),\, S\rangle \to \langle \text{TRY}(C', P, Q),\, S'\rangle}$$

$$[\text{try}_3]\ \frac{\langle C,\, S\rangle \to S'}{\langle \text{TRY}(C, P, Q),\, S\rangle \to \langle P,\, \text{pop2}(S')\rangle}$$

$$[\text{try}_4]\ \frac{\langle C,\, S\rangle \to \text{fail}}{\langle \text{TRY}(C, P, Q),\, S\rangle \to \langle Q,\, \text{pop}(S)\rangle}$$

Here S and S' are stacks of graphs. The rule [try$_1$] duplicates the top of the stack, and the TRY construct signals that the copy operation has happened. Repeated applications of the inference rule [try$_2$] model the evaluation of the condition in a small-step fashion. If the condition loops, [try$_2$] can be applied indefinitely, and we get an infinite transition sequence.

Intuitively, `P1` should loop, which is what happens in the implementation of GP 2. In the original semantics however, `P1` gets stuck because `r1!` diverges, which means that we cannot apply either of the inference rules [try$'_1$] or [try$'_2$] to resolve the `try` statement.

The original semantics tries to remedy this issue in the *semantic function* which associates to a program `P1` and host graph G the set $[\![\texttt{P1}]\!]G$ of all possible outcomes of the execution of `P1` on G. These outcomes can be a graph, the element fail, or $\perp$ which represents an infinite transition sequence. The original semantic function uses $\perp$ as an outcome if the transition sequence gets stuck. However, there are problems with this approach.

```
P2 = try Loop then skip else skip
Loop = {r1,r2}!
```

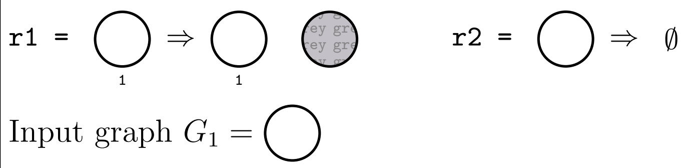


Figure 2.25: The GP 2 program `P2` and input graph G_1

Consider the program `P2` in Figure 2.25. The command `{r1,r2}` is a *rule set call*, meaning that rules `r1` and `r2` are selected nondeterministically. When `P2` is executed on the host graph G_1, an application of `r2` causes the loop to terminate since it removes the marked node which is necessary for either rule to be applicable. Hence `r1` may be applied a number of times, and then `r2` is applied once. But it should also be possible that `r2` is never called, resulting in a diverging computation. Hence the set of outcomes we want is $\{\perp,\ \emptyset,$ ○, ○○, ○○○, $\ldots\}$. According to the original semantics, however, the execution of `P2` on G_1 cannot get stuck since `Loop` *can* always transition to a graph; and by the rules [try$'_1$] and [try$'_2$], the execution cannot diverge either. So $\perp \notin [\![\texttt{P2}]\!]G_1 = \{\emptyset,$ ○, ○○, ○○○, $\ldots\}$. The small-step semantics corresponds exactly to our intuition of the operational behaviour of GP 2 programs. Moreover, we conjecture that the implementation is sound with respect to the small-step semantics, in that the behaviour of the implementation is covered by the small-step semantics.

```
P3 = try ({r3,r2}!)  then skip else skip
P4 = try r2 then skip else skip
```

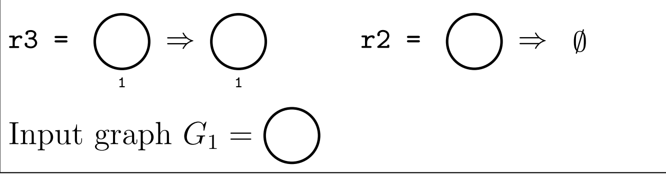


Figure 2.26: The GP 2 programs P3 and P4, and input graph G_1

In the original semantics, the behaviour of diverging programs may also lead to two programs being semantically equivalent, even though they should not be. Programs P and P' are *semantically equivalent* if $[P] = [P']$, i.e. they have the same outcomes for all host graphs. Consider program P3 from Figure 2.26. It can diverge on G_1 but is semantically equivalent to program P4 from the same figure, since the original semantics cannot detect that divergence. For instance, $[\texttt{P3}]G_1 = [\texttt{P4}]G_1 = \{\emptyset\}$, but $[\texttt{P3}]G_1$ should include $\perp$.

The aforementioned issues can also happen with `if` statements, which work similarly to `try` statements, except that the changes the condition made to the host graph are reversed, even if the evaluation of the condition succeeds. Nested loops such as `Loop!` can get stuck as well since their inference rules also assume that the loop body either results in a graph or fails.

Diverging computations not being modelled properly entails an undesirable property, namely *infinite nondeterminism*, i.e. there can be infinitely many configurations reachable in a single transition step. Consider the program P2 again. We have $[\texttt{Loop}]G = \{\emptyset,$ ○, ○○, ○○○, $\ldots, \perp\}$. In a transition sequence starting with $\langle \texttt{P2}, G\rangle$, since the `try` statement is resolved within a single step, it only takes one step to transition to either of the graphs in the set $\{\emptyset,$ ○, ○○, ○○○, $\ldots\}$, of which there are infinitely many.

The small-step semantics truly is small-step and as such, it accurately models looping computations with diverging transition sequences. When starting with a GP 2 program, it cannot get stuck, which is a property we call *non-blocking.* As a consequence of the small-step approach, we get *finite nondeterminism*, meaning we can only reach a finite number of configurations within a single transition step.

2.3.5 Original Semantics

Like Plotkin's structural operational semantics [57], the original GP 2 semantics is given by inference rules.

Most of these inference rules in Figure 2.27 have a horizontal bar. These rules consist of a *premise* above the bar and a *conclusion* below. The conclusion defines a transition step provided that the premise holds. A rule without a bar is called an *axiom* and can be applied to a configuration without any precondition.

The semantics operate on configurations, which are different states a program can be in during its execution. Configurations can contain elements of $\mathcal{G}$, the set of all GP 2 host graphs or of ComSeq, the set of command sequences as defined in the syntax (see Figure 2.21), where we assume that procedure IDs have been eliminated by macro expansion. This means that procedure IDs have been replaced with their defining command sequence, and name clashes arising from local declarations have been resolved by renaming. The element fail represents the program resulting in a failure state.

Definition 2.28 (Original Configuration)**.** We call a member of the set $(\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\}$ an *original configuration.*

- An original configuration in $\text{ComSeq} \times \mathcal{G}$ is called *non-terminal.*
- An original configuration in $\mathcal{G} \cup \{\text{fail}\}$ is called *terminal.* □

Definition 2.29 (Original Semantic Transition)**.** The rules in Figure 2.27 define the transition relation $\rightsquigarrow$ over the following set:

$$(\text{ComSeq} \times \mathcal{G}) \times ((\text{ComSeq} \times \mathcal{G}) \cup \mathcal{G} \cup \{\text{fail}\}).$$

The inference rules contain universally quantified variables, namely host graphs G and H, command sequences in ComSeq C, P, P', and Q, and rule set call R. The transitive closure of $\rightsquigarrow$ is denoted by $\rightsquigarrow^+$, and the reflexive transitive closure by $\rightsquigarrow^*$. □

In general, the execution of a program on a host graph may result in another graph, fail, or diverge. Also, executions can get *stuck* in that they reach a non-terminal configuration (neither a graph nor fail) to which no inference rule is applicable. Let $\mathcal{G}$ be the set of all host graphs and $\mathcal{G}^\oplus = \mathcal{G} \cup \{\perp, \text{fail}\}$. These outcomes are described by the semantic function. By $\mathcal{P}(\mathcal{G}^\oplus)$, we denote the powerset of $\mathcal{G}^\oplus$, i.e. the set of all subsets of $\mathcal{G}^\oplus$.

$$[\text{call}'_1]\ \frac{G \Rightarrow_R H}{\langle R,\, G\rangle \rightsquigarrow H} \qquad [\text{call}'_2]\ \frac{G \not\Rightarrow_R}{\langle R,\, G\rangle \rightsquigarrow \text{fail}}$$

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

$$[\text{seq}'_3]\ \frac{\langle P,\, G\rangle \rightsquigarrow \text{fail}}{\langle P; Q,\, G\rangle \rightsquigarrow \text{fail}}$$

$$[\text{if}'_1]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle P,\, G\rangle}$$

$$[\text{if}'_2]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle Q,\, G\rangle}$$

$$[\text{try}'_1]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle P,\, H\rangle}$$

$$[\text{try}'_2]\ \frac{\langle C,\, G\rangle \rightsquigarrow^+ \text{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, G\rangle \rightsquigarrow \langle Q,\, G\rangle}$$

$$[\text{alap}'_1]\ \frac{\langle P,\, G\rangle \rightsquigarrow^+ H}{\langle P!,\, G\rangle \rightsquigarrow \langle P!,\, H\rangle} \qquad [\text{alap}'_2]\ \frac{\langle P,\, G\rangle \rightsquigarrow^+ \text{fail}}{\langle P!,\, G\rangle \rightsquigarrow G}$$

$$[\text{alap}'_3]\ \frac{\langle P,\, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle}{\langle P!,\, G\rangle \rightsquigarrow H} \qquad [\text{break}']\ \langle \texttt{break}; P,\, G\rangle \rightsquigarrow \langle \texttt{break},\, G\rangle$$

(a) Inference rules for core commands

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

$[\text{skip}']$ $\langle \texttt{skip},\, G\rangle \rightsquigarrow G$ $\qquad$ $[\text{fail}']$ $\langle \texttt{fail},\, G\rangle \rightsquigarrow \text{fail}$

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

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

$[\text{try}'_4]$ $\langle \texttt{try}\ C\ \texttt{else}\ P,\, G\rangle \rightsquigarrow \langle \texttt{try}\ C\ \texttt{then skip else}\ P,\, G\rangle$

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

(b) Inference rules for derived commands

Figure 2.27: Original GP 2 Semantics

Definition 2.30 (Original Semantic Function)**.** The *original semantic function* is the function $[\_] : \text{ComSeq} \to (\mathcal{G} \to \mathcal{P}(\mathcal{G}^{\oplus}))$ which, for a command sequence P and a host graph G, is defined as

$$\begin{aligned} [P]G \quad = \quad & \{X \in \mathcal{G} \cup \{\text{fail}\} \mid \langle P, G\rangle \rightsquigarrow^{+} X\} \\ & \cup \{\bot \mid P \text{ can diverge or get stuck from } G\}. \end{aligned}$$

By *divergence* we mean non-termination, that is the existence of an infinite transition sequence starting in $\langle P, G\rangle$. □

2.3.6 Small-Step Semantics

In this subsection, we provide a fully small-step semantics defined by inference rules, and provide examples of transition sequences. This is introduced by this thesis' author in [25].

One major difference compared to the original semantics is that, instead of operating on a single graph, the small-step semantics operates on a stack of graphs. The top of that stack is the graph that rules are applied to, while the rest provide a history of earlier graphs that can be reverted to quickly.

Another change is the addition of ITE and TRY statements that stand in for `if` and `try` statements whose condition is currently being evaluated.

Due to these additional constructs, the small-step semantics needs to distinguish between command sequences that are valid GP 2 programs, and command sequences that are intermediary (extended command sequences).

Definition 2.31 (Command Sequence)**.** A *command sequence* is a member of ComSeq from Definition 2.29 that satisfies the following.

A `break` must be enclosed within a loop. If a `break` is in the condition of a branching statement (i.e. an `if`, `try`, ITE, or TRY statement), the enclosing loop must be within the same condition.

We call this constraint the *break condition*, and the set of these command sequences CommandSeq. □

The break condition is one of the context conditions in [8]. This constraint is not specific to graph programs: Java, C, and Python have similar restrictions on the use of `break` statements.

Definition 2.32 (Extended Command Sequence)**.** An *extended command sequence* is a member of ComSeq from Definition 2.29, where commands can also be of the form ITE(Block,Block,Block) or TRY(Block,Block,Block) (see the definition of Command in the syntax in Figure 2.21).

We define ExtComSeq as the set of extended command sequences. □

The auxiliary constructs ITE and TRY do not follow the break condition since we may want a `break` outside of a loop in an intermediary transition step. The ITE and TRY statements serve to advance the command sequence in the condition in a small-step fashion, as well as to maintain the stack of host graphs. When we enter an ITE or TRY statement, the top of the stack (and current host graph) is duplicated in order to keep a backup. When these statements end, we either pop the top, modified graph, or the second graph on the stack which is the unmodified backup copy depending on the outcome of the condition. The stack structure is needed because `if` and `try` statements may be nested. Whenever we enter an ITE or TRY construct, we push a graph, and whenever we exit one, we pop a graph. This ensures that the stack always contains enough graphs to pop and that the current host graph is always on top.

Definition 2.33 (Graph Stack)**.** A *graph stack* $S = [G_1, G_2, G_3, \ldots, G_n]$ is a finite ordered list of GP 2 host graphs with the following operations.

- $\text{top}(S) = G_1$.
- $\text{pop}(S) = [G_2, G_3, \ldots, G_n]$.
- $\text{pop2}(S) = [G_1, G_3, \ldots, G_n]$
- $\text{push}(G, S) = [G, G_1, G_2, \ldots, G_n]$, where G is a host graph. □

We call $\mathcal{S}$ the set of all non-empty graph stacks.

Definition 2.34 (Small-Step Semantic Transition)**.** The rules in Figure 2.28 inductively define a transition relation $\rightarrow$ over the following set:

$$(\text{ExtComSeq} \times \mathcal{S}) \times ((\text{ExtComSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}).$$

We call this transition the *small-step (semantic) transition.* □

There are several universally quantified meta-variables within the inference rules. P, P', Q, Q', C, and C' stand for extended command sequences in ExtComSeq, S stands for a graph stack in $\mathcal{S}$, G represents a host graph, and R represents a rule set. We denote the transitive closure of $\rightarrow$ by $\rightarrow^+$, and the reflexive transitive closure by $\rightarrow^*$.

Definition 2.35 (Small-Step Configuration)**.** We call an element of the set $(\text{CommandSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}$ a *(small-step) configuration.*

[call$_1$] $\dfrac{\mathrm{top}(S) \Rightarrow_R G}{\langle R,\, S\rangle \to \mathrm{push}(G, \mathrm{pop}(S))}$ [call$_2$] $\dfrac{\mathrm{top}(S) \not\Rightarrow_R}{\langle R,\, S\rangle \to \mathrm{fail}}$

[seq$_1$] $\dfrac{\langle P,\, S\rangle \to \langle P',\, S'\rangle}{\langle P; Q,\, S\rangle \to \langle P'; Q,\, S'\rangle}$ [seq$_2$] $\dfrac{\langle P,\, S\rangle \to S'}{\langle P; Q,\, S\rangle \to \langle Q,\, S'\rangle}$

[seq$_3$] $\dfrac{\langle P,\, S\rangle \to \mathrm{fail}}{\langle P; Q,\, S\rangle \to \mathrm{fail}}$ [break] $\langle \texttt{break}; P,\, S\rangle \to \langle \texttt{break},\, S\rangle$

[alap$_1$] $\langle P!\,,\, S\rangle \to \langle \texttt{try}\ P\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip},\, S\rangle$

[alap$_2$] $\langle \mathrm{TRY}(\texttt{break},\, P!\,,\, \texttt{skip}),\, S\rangle \to \mathrm{pop2}(S)$

[if$_1$] $\langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, S\rangle \to \langle \mathrm{ITE}(C, P, Q),\, \mathrm{push}(\mathrm{top}(S),\, S)\rangle$

[if$_2$] $\dfrac{\langle C,\, S\rangle \to \langle C',\, S'\rangle}{\langle \mathrm{ITE}(C, P, Q),\, S\rangle \to \langle \mathrm{ITE}(C', P, Q),\, S'\rangle}$

[if$_3$] $\dfrac{\langle C,\, S\rangle \to S'}{\langle \mathrm{ITE}(C, P, Q),\, S\rangle \to \langle P,\, \mathrm{pop}(S')\rangle}$

[if$_4$] $\dfrac{\langle C,\, S\rangle \to \mathrm{fail}}{\langle \mathrm{ITE}(C, P, Q),\, S\rangle \to \langle Q,\, \mathrm{pop}(S)\rangle}$

[try$_1$] $\langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q,\, S\rangle \to \langle \mathrm{TRY}(C, P, Q),\, \mathrm{push}(\mathrm{top}(S),\, S)\rangle$

[try$_2$] $\dfrac{\langle C,\, S\rangle \to \langle C',\, S'\rangle}{\langle \mathrm{TRY}(C, P, Q),\, S\rangle \to \langle \mathrm{TRY}(C', P, Q),\, S'\rangle}$

[try$_3$] $\dfrac{\langle C,\, S\rangle \to S'}{\langle \mathrm{TRY}(C, P, Q),\, S\rangle \to \langle P,\, \mathrm{pop2}(S')\rangle}$

[try$_4$] $\dfrac{\langle C,\, S\rangle \to \mathrm{fail}}{\langle \mathrm{TRY}(C, P, Q),\, S\rangle \to \langle Q,\, \mathrm{pop}(S)\rangle}$

(a) Inference rules for core commands

[or$_1$] $\langle P\ \texttt{or}\ Q,\, S\rangle \to \langle P,\, S\rangle$ [or$_2$] $\langle P\ \texttt{or}\ Q,\, S\rangle \to \langle Q,\, S\rangle$

[skip] $\langle \texttt{skip},\, S\rangle \to S$ [fail] $\langle \texttt{fail},\, S\rangle \to \mathrm{fail}$

[if$_5$] $\langle \texttt{if}\ C\ \texttt{then}\ P,\, S\rangle \to \langle \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip},\, S\rangle$

[try$_5$] $\langle \texttt{try}\ C\ \texttt{then}\ P,\, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ \texttt{skip},\, S\rangle$

[try$_6$] $\langle \texttt{try}\ C\ \texttt{else}\ P,\, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ P,\, S\rangle$

[try$_7$] $\langle \texttt{try}\ C,\, S\rangle \to \langle \texttt{try}\ C\ \texttt{then}\ \texttt{skip}\ \texttt{else}\ \texttt{skip},\, S\rangle$

(b) Inference rules for derived commands

Figure 2.28: Small-Step GP 2 Semantics

- A configuration in CommandSeq $\times\,\mathcal{S}$ is called *non-terminal.*
- A configuration in $\mathcal{S} \cup \{\text{fail}\}$ is called *terminal.* □

Definition 2.36 (Small-Step Extended Configuratuions)**.** We call an element of the set $(\text{ExtComSeq} \times \mathcal{S}) \cup \mathcal{S} \cup \{\text{fail}\}$ a *(small-step) extended configuration.*

- An extended configuration in ExtComSeq $\times\,\mathcal{S}$ is called *non-terminal.*
- An extended configuration in $\mathcal{S} \cup \{\text{fail}\}$ is called *terminal.* □

The inference rules inductively define the transition relation $\rightarrow$. The rules [call_1] and [call_2] are base cases. Their premises are GP 2 derivations. Which of the two premises is satisfied depends on whether $\text{top}(S) \Rightarrow_R G$ or $\text{top}(S) \not\Rightarrow_R$, i.e. whether a rule in the rule set can be applied to the current host graph or not. The `if` and `try` statements are modelled by the [if_i] and [try_i] rules.

Sequential composition of commands is covered by [seq_1], [seq_2], and [seq_3], covering the cases of whether the first command called on a host graph results in a configuration, a graph stack, or fail.

Loops are semantically described as a try statement in [alap_1]. Calling a command sequence as long as possible is modelled by trying to apply the command sequence, and if it succeeds, keep applying it as long as possible. Breaking from a loop is handled by [break], which makes sure commands following the break are discarded, and [alap_2], which terminates the loop if there is an isolated break in the TRY condition.

Figure 2.28a shows the inference rules for the core commands of GP 2, while Figure 2.28b gives the inference rules for derived commands such as `or`, `skip`, and `fail`, as well as some `if` and `try` statements with omitted `then` and `else` clauses. These commands are referred to as *derived* commands because they are defined by the core commands (see [58] for the case of the original semantics).

Let us look at a couple of examples of transition sequences in Figures 2.29 and 2.30, the first to illustrate loops, and the second to illustrate `if` and `try` statements. For each transition, we note the applied inference rule as a subscript. If the conclusion of [rule_1] is used as a premise for [rule_2], we denote it by $\frac{[\text{rule}_1]}{[\text{rule}_2]}$.

Example 2.37. Consider the program `P=r!` and the rule `r` : $_1$○→○ $\Rightarrow$ $_1$○. Let us examine a transition sequence of `P` applied to the graph ○→○→○, as seen in Figure 2.29.

⟨`r!`, [○→○→○]⟩

$\to_{[\text{alap}_1]}$ ⟨`try r then r! else skip`, [○→○→○]⟩

$\to_{[\text{try}_1]}$ ⟨TRY (`r`, `r!`, `skip`), [○→○→○, ○→○→○]⟩

$\to_{\frac{[\text{call}_1]}{[\text{try}_3]}}$ ⟨`r!`, [○→○]⟩

$\to_{[\text{alap}_1]}$ ⟨`try r then r! else skip`, [○→○]⟩

$\to_{[\text{try}_1]}$ ⟨TRY (`r`, `r!`, `skip`), [○→○, ○→○]⟩

$\to_{\frac{[\text{call}_1]}{[\text{try}_3]}}$ ⟨`r!`, [○]⟩

$\to_{[\text{alap}_1]}$ ⟨`try r then r! else skip`, [○]⟩

$\to_{[\text{try}_1]}$ ⟨TRY (`r`, `r!`, `skip`), [○, ○]⟩

$\to_{\frac{[\text{call}_2]}{[\text{try}_4]}}$ ⟨`skip`, [○]⟩

$\to_{[\text{skip}]}$ [○]

Figure 2.29: Example transition sequence of program `P`.

We start by applying [alap$_1$] which turns the loop into a `try` statement. Unlike in the original semantics, we model a loop by trying to apply its body, and if it is successful, we call the loop again.

The inference rule [try$_1$] transforms the `try` statement into the auxiliary TRY construct, which advances the program in a small-step fashion, unlike the original semantics. There is a similar ITE construct which models `if` statements. The top of the graph stack is duplicated since the changes made by the condition of the `try` may be discarded.

We then apply `r` to the current host graph (top of the stack) so [call$_1$] can be applied. This serves as a premise for [try$_3$], which ends the TRY statement, pops the second element of the stack, and moves on to the `then` part which is the original loop.

We repeat this process until `r` is no longer applicable to the host graph. At this point, [call$_2$] serves as the premise for [try$_4$] which exits the TRY statement. This time, the condition results in fail, so we move on to the `else` part which is `skip` and the loop terminates.

Now consider program `P'=try(if (r1;r1) then (r1;r1))` and the rule `r1` : $_1$○→○ $\Rightarrow$ $_1$○. A transition sequence of `P'` applied to host graph ○←○→○ can be found in Figure 2.30.

Since the `try` statement does not have a `then` or `else` part, we first apply

⟨`try(if(r1;r1) then(r1;r1))`, [○←○→○]⟩

$\to_{[\mathrm{try}_7]}$ ⟨`try (if (r1;r1) then (r1;r1)) then skip else skip`, [○←○→○]⟩

$\to_{[\mathrm{try}_1]}$ ⟨TRY(`if (r1;r1) then (r1;r1)`, `skip`, `skip`), [○←○→○, ○←○→○]⟩

$\to_{\frac{[\mathrm{if}_5]}{[\mathrm{try}_2]}}$ ⟨TRY(`if (r1;r1) then (r1;r1) else skip`, `skip`, `skip`), [○←○→○, ○←○→○]⟩

$\to_{\frac{[\mathrm{if}_1]}{[\mathrm{try}_2]}}$ ⟨TRY(ITE(`r1;r1`, `r1;r1`, `skip`), `skip`, `skip`), [○←○→○, ○←○→○, ○←○→○]⟩

$\to_{\frac{\frac{\frac{[\mathrm{call}_1]}{[\mathrm{seq}_2]}}{[\mathrm{if}_2]}}{[\mathrm{try}_2]}}$ ⟨TRY(ITE(`r1`, `r1;r1`, `skip`), `skip`, `skip`), [○→○, ○←○→○, ○←○→○]⟩

$\to_{\frac{\frac{[\mathrm{call}_1]}{[\mathrm{if}_3]}}{[\mathrm{try}_2]}}$ ⟨TRY(`r1;r1`, `skip`, `skip`), [○←○→○, ○←○→○]⟩

$\to_{\frac{[\mathrm{call}_1]}{[\mathrm{try}_2]}}$ ⟨TRY(`r1`, `skip`, `skip`), [○→○, ○←○→○]⟩

$\to_{\frac{[\mathrm{call}_1]}{[\mathrm{try}_3]}}$ ⟨`skip`, [○]⟩

$\to_{[\mathrm{skip}]}$ [○]

Figure 2.30: Example transition sequence of program `P'`.

[try$_7$], which adds `skip` as both the `then` and `else` parts.

The inference rule [try$_1$] turns the `try` statement into the auxiliary TRY statement and duplicates the top of the stack. For most of the remaining transition sequence, we apply [try$_2$] under various premises to advance the condition.

Since the `if` has no `else` part, [if$_5$] completes it with a `skip`. The `if` statement is then turned into the auxiliary ITE statement, duplicating the top of the stack once again.

The rule `r1` is applied to the host graph which advances the concatenation with [seq$_2$], the ITE with [if$_2$], and the TRY with [try$_2$]. Calling `r1` a second time resolves the ITE, and the top of the stack is popped since changes made by the conditions of `if` statements are reversed.

We keep applying the condition of the TRY, until we resolve it with [try$_3$]. This time the second graph on the stack is popped since changes made by the condition of a `try` that did not result in fail are preserved. □

2.3.7 Properties of the Small-Step Semantics

In this subsection, we show that the small-step semantics is non-blocking, i.e. if a transition sequence leads to an extended configuration, we can always apply an inference rule (Proposition 2.40). Note that we can only guarantee the non-blocking property for extended configurations that are part of a transition sequence originating in a valid GP 2 program. We call those *reachable* extended configurations. This is reasonable because there can be no other types of configurations in a transition sequence modelling a GP 2 program.

Furthermore, we will describe the outcomes of a transition sequence starting with a valid GP 2 program (Proposition 2.42), and show that we have finite nondeterminism (Proposition 2.45), i.e. there are only finitely many one-step transitions starting from any configuration, and what it means for the semantic function.

Let us first look at a lemma that guarantees we can make a transition step from extended configurations that do not contain a `break`, which is the first step towards showing the non-blocking property.

Lemma 2.38 (Progress from Extended Configurations). *Let $\langle P, S\rangle$ be an extended configuration. Then one of the following applies:*

- $\langle P, S\rangle \to \langle P', S'\rangle$ *for some extended configuration* $\langle P', S'\rangle$*.*
- $\langle P, S\rangle \to S'$ *for some graph stack* $S' \in \mathcal{S}$*.*
- $\langle P, S\rangle \to \mathrm{fail}$.
- *P is not a command sequence and contains a* `break`*.*

Proof. We shall prove this lemma by going through what P could be according to the syntax and the semantics.

Case 1: P is a rule set call. Then either $\mathrm{top}(S) \Rightarrow_P G$ or $\mathrm{top}(S) \not\Rightarrow_P$. So either $[\mathrm{call}_1]$ or $[\mathrm{call}_2]$ can be applied.

Case 2: P is a loop. If P is a loop, $[\mathrm{alap}_1]$ can be applied.

Case 3: P is fail*,* `skip` *or an* `or` *statement.* Then [fail], [skip], or $[\mathrm{or}_1]$ can be applied respectively.

Case 4: P is of the form `if` P_1 `then` P_2 `else` P_3 *or* `try` P_1 `then` P_2 `else` P_3. Then $[\mathrm{if}_1]$ or $[\mathrm{try}_1]$ can be applied. If any then-clause or else-clause is omitted as specified by the syntax, $[\mathrm{if}_5]$, $[\mathrm{try}_5]$, $[\mathrm{try}_6]$, or $[\mathrm{try}_7]$ can be applied.

Case 5: P is of the form ITE(P_1, P_2, P_3) or TRY(P_1, P_2, P_3). If P contains a `break`, the fourth point of the lemma is satisfied. Specifically, P is not a command sequence since it contains an ITE or TRY statement, and

P contains a `break` by assumption. So for the remainder of this case, assume P does not contain a `break`. Let P_1' be the first component of P_1, i.e. P_1 is not a sequential composition. It is possible for the first component of P_1' to contain an ITE or TRY statement, whose condition has a first component that also contains such a statement. We shall show the lemma's statement by induction on how many ITE or TRY statements are nested in P_1' in this way, i.e. via the first sequential component of the condition.

- For the base case, assume P_1' is not an ITE or TRY statement. Then P_1' must be covered by one of the cases 1-4, satisfying the lemma.
- Now for the induction step, assume that P_1' is an ITE or TRY statement. Then P_1' does derive either a configuration $\langle P_1''', S'\rangle$, a graph stack S' or fail by the induction hypothesis. Hence one of [if$_2$], [if$_3$], [if$_4$], [try$_2$], [try$_3$], or [try$_4$] can be applied to $\langle P, S\rangle$.

Case 6: P is a sequential composition. Then we can decompose P into $P = P_1; P_2$ where P_1 is not a sequential composition. We can apply [seq$_1$], [seq$_2$], or [seq$_3$] since $\langle P_1, S\rangle \to \langle P_1', S'\rangle$, $\langle P_1, S\rangle \to S'$, or $\langle P_1, S\rangle \to$ fail respectively by cases 1 to 5.

Case 7: P contains a `break`.

If P is not a command sequence, the final point of the lemma is satisfied. Otherwise, P satisfies context conditions, meaning it must be enclosed within a loop, so either case 2 or one of the other previous cases is applicable. □

Lemma 2.38 has a case where the extended command sequence contains a `break`. This is because for a transition sequence not to get stuck on a `break`, we need to start with a command sequence where the `break` is within a loop, which we cannot guarantee if we consider a single transition step like in Lemma 2.38. In order to deal with this case, we prove that we can construct a transition sequence that leads to a state with no `break` in the following lemma. However, we need to restrict it to extended configurations reachable from a valid GP 2 program. We say that an extended configuration C is *reachable* if there is a configuration $\langle P, [G]\rangle$ such that $\langle P, [G]\rangle \to^* C$. This will still allow us to work towards non-blocking, since we only care about transition sequences that start with valid GP 2 programs.

Lemma 2.39 (Removing the `break` Statement). *Let $\langle P, S\rangle$ be an extended configuration that is reachable and non-terminal. Suppose that P contains a `break`. Then one of the following applies.*

- *There is an extended configuration $\langle P', S'\rangle$ containing no `break` statement such that $\langle P, S\rangle \to^* \langle P', S'\rangle$.*

- *There is a graph stack S' such that $\langle P, S\rangle \to^+ S'$.*
- $\langle P, S\rangle \to^+$ fail

Proof. First assume that $\langle P, S\rangle$ satisfies context conditions, i.e. the `break` is contained within a loop, and if the `break` is in the condition of an `if` or `try` statement, the enclosing loop must be in the same condition.

We will apply various inference rules to construct a transition sequence starting in $\langle P, S\rangle$. Remember that whenever we apply such an inference rule, it results in either a non-terminal extended configuration, a graph stack, or fail. If it results in a graph stack or fail, the second or third point of the lemma is satisfied. So at each step of the transition sequence we construct, we only need to consider the case where an inference rule results in a non-terminal extended command sequence.

If there are multiple loops with `break` statements, they are either in different sequential composition components, or nested. So let us show this lemma by induction on nesting and sequential composition.

As a base case, assume P contains a single loop with a `break`, and want to show we can apply a sequence of inference rules that ultimately removes the `break`. So P is of the form Q_0`;`Q_1`!;`Q_2, where Q_1 contains a `break`, and neither Q_0 nor Q_2 do. (What follows also applies if P is of the form Q_1`!;`Q_2, Q_0`;`Q_1`!`, or Q_1`!`.) We can repeatedly apply Lemma 2.38 to transition to Q_1`!;`Q_2. Then we apply [alap$_1$] followed by [try$_1$] to get TRY(Q_1, Q_1!, `skip`). We can then use Lemma 2.38 repeatedly as a premise for [try$_2$] until we get TRY(Q_3`;`Q_4, Q_1!, `skip`), where Q_3 contains `break` and is not a sequential composition. If Q_3 is a `break`, we can apply [try$_2$] under the premise of [break], followed by [alap$_2$] to get rid of the `break`. We know Q_3 cannot be a loop since we assumed Q_1! is the enclosing loop of the `break`. So Q_3 is either an `or`, `if`, or `try` statement. If it is an `or` statement, we can apply [or$_1$] or [or$_2$] to either remove the `break` or lead to TRY(`break;`Q_5, Q_1!, `skip`). Similarly, if Q_3 is an `if` or `try` statement, the `break` must be in the `then` or `else` part due to context conditions, and we can use inference rules to either remove the `break` or lead to TRY(`break;`Q_5, Q_1!, `skip`). We can now apply [try$_2$] under the premise of [break] to get TRY(`break`, Q_1!, `skip`). To this, we can apply [alap$_2$], which gets rid of the `break`.

For the induction step, let us first consider the case of nesting. Assume that P is of the form $Q_0; (Q_1; Q_2; Q_3)!$`;` Q_4, where Q_2 satisfies the lemma statement, and Q_1 or Q_2 contains a `break`. We can use the same arguments as in the base case in addition to [seq$_1$] under the premise of the induction hypothesis to get rid of any `break`.

Now consider sequential composition. As an induction step, assume that P is of the form $Q_0; Q_1$!`;`Q_2; Q_3!`;`Q_4, where one of Q_1 or Q_3 satisfies the lemma statement, and the other contains a `break`. Again, we can use the arguments from the base case as well as the induction hypothesis in conjunction with [seq_1] to remove any `break`.

Finally, assume that $\langle P, S\rangle$ does not satisfy context conditions, i.e. either there is a `break` without an enclosing loop, or there is a `break` in the condition of a branching statement whose enclosing loop is not within that condition. Since $\langle P, S\rangle$ is reachable, the latter cannot be the case: transitions steps cannot separate a `break` from its enclosing loop in a way that they stop being within the same condition of a branching statement (loops can only be removed by inference rules, they cannot be "moved"). So suppose there is a `break` without an enclosing loop. This must be because [alap_1] is applied earlier in the transition sequence, so it must be within the condition of a `try` or TRY. So we can use the same arguments as earlier in the proof, except that we need not argue that some of the inference rule, such as [alap_1] or [try_1] need to be applied. □

Now that we have Lemmata 2.38 and 2.39, we can prove that the non-blocking property holds.

Proposition 2.40 (Non-Blocking Property). *Let $\langle P, S\rangle$ be an extended configuration that is reachable and non-terminal. Then there is a transition step $\langle P, S\rangle \to C$ for some extended configuration C.*

Proof. If P does not contain a `break`, this proposition follows from Lemma 2.38. Otherwise, it follows from Lemma 2.39. □

Let us now introduce a lemma that makes various statements about the size of host graph stacks in order to ensure that the inference rules are well-defined. Since we defined stacks to be nonempty, we want to make sure that if a transition sequence starts with a nonempty stack, it cannot lead to an empty stack, which the following lemma shows. Furthermore, when a transition sequence terminates in a graph stack, we want that stack to only contain one host graph.

For this lemma, we want to start from a valid GP 2 program, not extended command sequences in general (since they may contain auxiliary constructs like ITE and TRY). So we consider configurations in CommandSeq$\times\mathcal{S}$. These follow the context conditions on where the `break` statement can appear as specified in [8].

Lemma 2.41 (Stack Size). *Let $\langle P, [G]\rangle$ be a configuration in CommandSeq$\times$ $\mathcal{S}$.*

(a) If $\langle P, [G]\rangle \to^* \langle P', S\rangle$, where $\langle P', S\rangle$ is an extended configuration, then $|S| \geq 1$.

(b) If $\langle P, [G]\rangle \to^+ S$, where S is a graph stack, then $|S| = 1$.

Proof. The statement in (a), is satisfied for zero transition steps. So let us examine the inference rules that contain push, pop, and pop2. The rule [call_1] contains both push and pop, but preserves the size of the stack. The rules that push a graph onto the stack are [if_1] and [try_1] which are exactly the rules that introduce an ITE or a TRY. The rules that pop a graph from the stack are [alap_2], [if_3], [if_4], [try_3], and [try_4]. These are exactly rules that remove an ITE or TRY from the extended command sequence. Since $\langle P, [G]\rangle$ contains no ITE or TRY statements and only one host graph, we have $|S| = \#(P') + 1$, where $\#$ counts the combined number of ITE and TRY statements in an extended command sequence. Since $|S| = \#(P') + 1$, we have $|S| \geq 1$.

Now in case (b), we can break down the transition sequence into $\langle P, [G]\rangle \to^* \langle P', S'\rangle \to S$. Like in the proof of (a), the formula $|S'| = \#(P') + 1$ applies. Let us examine which inference rules can be applied in the final step of the transition. It can only be either [skip], [call_1], or [alap_2]. To apply [skip], P' must be `skip` and $\#(\texttt{skip}) = 0$, so $|S| = |S'| = 1$. To apply [call_1], P' must be a rule set call, and hence cannot contain ITE or TRY, so $|S| = |S'| = 1$. To apply [alap_2], P' must be of the form $\text{TRY}(\texttt{break}, P''!, \texttt{skip})$, where P'' is an extended command sequence. We know P'' cannot contain an ITE or TRY statement because they can only be nested in their first argument. Indeed, if an extended command sequence already starts with an ITE or TRY, no inference rule allows for said ITE or TRY statement to be nested within another one. So the only way to nest statements is via the rule [try_2], which modifies the first argument. But the first argument of P' is `break`, which contains no ITE or TRY statements. So $\#(P') = 1$ and $|S'| = 2$. Since we apply [alap_2], we have $S = \text{pop2}(S')$, so $|S| = |S'| - 1 = 1$. □

We also want to make sure that if we call pop2 on a stack to pop its second element, the stack does indeed contain at least two elements. More precisely, under the premise of Lemma 2.41, if $\langle P, [G]\rangle \to^+ \langle P', \text{pop2}(S)\rangle$ (an extended configuration) or $\langle P, [G]\rangle \to^+ \text{pop2}(S)$ (a graph stack), then $|S| \geq 2$. This follows directly from Lemma 2.41 since $|\text{pop2}(S)| = |S| - 1$.

Let us now use Lemmata 2.38, 2.39, and 2.41 is to describe what the possible outcomes of a transition sequence starting in a valid GP 2 program are.

Theorem 2.42 (Outcomes of Transition Sequences)**.** *Let $\langle P, [G]\rangle$ be a configuration. Then one of the following applies:*

- *There is an infinite transition sequence $\langle P, [G]\rangle \to \langle P_1, S_1\rangle \to \langle P_2, S_2\rangle \to \ldots$ where $\langle P_i, S_i\rangle$ is an extended configuration for all $i \geq 1$.*
- *$\langle P, [G]\rangle \to^+ [G']$ for some host graph G'.*
- *$\langle P, [G]\rangle \to^+$ fail.*

Proof. Lemma 2.41 guarantees that if a transition sequence starts in $\langle P, [G]\rangle$ and ends in a stack, that stack only contains one graph. So for this proposition, it is enough to show that transition sequences end in a stack in the relevant cases.

In order to get rid of a potential `break` statement in P, we can apply Lemma 2.39 to $\langle P, [G]\rangle$. If we get a graph stack or fail, we fulfil the second or third case of this proposition. Otherwise we get an extended configuration $\langle P', S\rangle$ that contains no `break`.

Since there is now no `break` in either $\langle P, [G]\rangle$ or $\langle P', S\rangle$, we can apply the first, second, and third cases of Lemma 2.38 either indefinitely to get an infinite transition sequence, or until we get a graph stack or fail. □

Now that we know the possible outcomes of a transition sequence, we can define the semantic function for the small-step semantics.

Definition 2.43 (Small-Step Semantic Function)**.** The *small-step semantic function* is defined as $[\![\_]\!] : \text{CommandSeq} \to (\mathcal{G} \to 2^{\mathcal{G}^{\oplus}})$, where $\mathcal{G}$ is the set of host graphs, $[\mathcal{G}]$ the set of stacks consisting of exactly one host graph (which we can identify with single host graphs), and $\mathcal{G}^{\oplus} = [\mathcal{G}] \cup \{\text{fail}, \bot\}$. The symbol $\bot$ is used to represent an infinite transition sequence, i.e. divergence. The function is defined as

$$[\![P]\!]G = \{X \in [\mathcal{G}] \cup \{\text{fail}\} \mid \langle P, G\rangle \to^+ X\} \cup \{\bot \mid P \text{ can diverge from } G\}.$$

□

This functions differs from the original semantic function presented in [60] and Subsection 2.3.5 since $\bot$ is only used when P diverges, because we know by Proposition 2.40 that P cannot get stuck. We will show in Lemma 2.51 that every infinite or stuck transition sequence in the original semantics corresponds to an infinite transition sequence in the small-step semantics.

Let us now examine the property of *finite nondeterminism* as specified by Apt in Section 4.1 of [4], i.e. the set of elements reachable from a configuration in one transition step is finite. A related concept is *bounded nondeterminism*, where the cardinality of the aforementioned set is finite and depends on the program only (and not on the size of the current state). An example for a language with bounded nondeterminism is Dijkstra's language of guarded commands [65]. Many references [28, 37, 65, 81] equate the concepts of finite and bounded nondeterminism and call it "bounded nondeterminism". However, rule-based languages generally have unbounded nondeterminism because they rely on nondeterministic rule matching. This also applies to GP 2 which the following example illustrates.

Example 2.44. Consider the rule $\mathtt{r} : {}_1\bigcirc\!\leftarrow\!\bigcirc \Rightarrow {}_1\bigcirc$ and the *comb graph* G_4 as shown in Figure 2.31. There are four possible matches for the left-

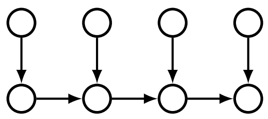


Figure 2.31: The comb graph G_4

hand side of rule r in graph G_4 since due to the dangling condition, the node without a number can only be matched with nodes that do not have additional adjacent edges. So applying the rule can result in four different non-isomorphic graphs, which is a finite amount. When applying r to comb graph G_k, we get k non-isomorphic graphs, which depends on the size of the host graph and hence is not bounded. □

We now show that GP 2 does have finite nondeterminism.

Proposition 2.45 (Finite Nondeterminism). *Let $\gamma \in \mathit{ExtComSeq} \times \mathcal{S}$ be an extended configuration, and $T_\gamma = \{\gamma' \,|\, \gamma \to \gamma' \in \mathit{ExtCommSeq} \times \mathcal{S}\}$. Then $|T_\gamma|$ is finite.*

Proof. The only inference rules that cause nondeterminism are [or$_1$], [or$_2$], and [call$_1$]. If the rules [or$_1$] and [or$_2$] are applicable to γ then there are exactly two configurations reachable from γ. In [call$_1$], the nondeterminism comes from several GP 2 rules being called non-deterministically as part of a rule set, as well as from all the ways these rules can be matched in the host graph. Since rule sets and host graphs are finite, the number of configurations reachable from γ in one step via the inference rule [call$_1$] is finite as well. □

Reynolds [65] defines this kind of nondeterminism using the semantic function instead of the set of configurations reachable in one step. The following corollary shows that this semantics fulfils that definition as well.

Corollary 2.46. *Let $P \in$ CommandSeq and $G \in \mathcal{G}$ such that $[\![P]\!]G$ is infinite. Then $\bot \in [\![P]\!]G$.*

Proof. Let $\gamma_0 = \langle P, [G] \rangle$. Then $T^*_{\gamma_0} = \{\gamma \mid \gamma_0 \rightarrow^* \gamma \in \text{ExtCommSeq} \times \mathcal{S}\}$ is infinite as well since it contains all elements of $[\![P]\!]G$ except perhaps fail or $\bot$. The set $T^*_{\gamma_0}$ can be seen as a tree whose nodes are configurations and whose edges are defined by transitions. Since T_γ is finite for all configurations γ by Proposition 2.45, each node in the tree only has finitely many adjacent nodes. By König's Lemma [50], the tree contains an infinite path. Since every node of the tree is reachable from the root γ_0, there is an infinite path starting from γ_0. By definition of the tree, this means there is an infinite transition sequence starting with γ_0. By definition of the small-step semantic function, we can conclude that $\bot \in [\![P]\!]G$. $\square$

2.3.8 Comparison of the Two Semantics

In this subsection, we show that the small-step semantics is a conservative extension of the original one, i.e. their behaviour is equivalent on converging configurations, and if a configuration diverges in the original semantics, it also diverges in the small-step one.

When we mention graph stacks in this subsection, we allow them to be empty. We use the notation $[G_1, G_2, \ldots, G_k, S]$ (where G_i are graphs, S is a graph stack, and $k > 0$) to denote a stack whose top k elements are $G_1, G_2, \ldots, G_k$, and whose remaining elements are the elements of S.

Lemma 2.47 (Simulating Finite Original Transition Sequences)**.** *Let $P \in$ CommandSeq, $G \in \mathcal{G}$, and $X \in \{\langle P', G' \rangle, G', \mathit{fail}\}$, where $P' \in$ CommandSeq and $G' \in \mathcal{G}$. If $\langle P, G \rangle \rightsquigarrow^* X$, then, for any graph stack S, there is a transition sequence*

- *$\langle P, [G, S] \rangle \rightarrow^* \langle P', [G', S] \rangle$ if $X = \langle P', G' \rangle$.*
- *$\langle P, [G, S] \rangle \rightarrow^* [G']$ if $X = G'$.*
- *$\langle P, [G, S] \rangle \rightarrow^* \mathit{fail}$ if $X = \mathit{fail}$.*

Proof. We shall prove this lemma by induction on the number of `if`, `try`, and `!` statements in P combined.

If P has no `if`, `try`, or `!` statements, none of the [if], [try], and [alap] inference rules are applicable. All other rules behave identically in both semantics when identifying the tops graph of the stacks in the small-step rules with the graphs in the original rules. Hence the base case is satisfied.

As the induction hypothesis, assume the lemma holds for P containing k `if`, `try`, or `!` statements. Now consider the case where P contains $k+1$ of them. Let $\langle P_1, G_1\rangle \rightsquigarrow \langle P_2, G_2\rangle$ be a derivation step of $\langle P, G\rangle \rightsquigarrow^* X$ that uses an [if], [try], or [alap] rule (possibly as a premise for another rule such as [seq$_1$]). If such a step does not exist, the lemma holds by the same argument used in the base case. Consider the [if], [try], or [alap] rule [r] that relates to the `if`, `try`, or `!` statement enclosing all others resolved in the same step. Then no rule where [r] is a premise (or the premise of a premise) is an [if], [try], or [alap] rule. Since those are identical in both semantics, we only need to show that the part of P resolved by [r] is resolved in a way that fulfils the lemma statement.

If [r] = [if$_1$], we have $\langle \texttt{if}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, G_3\rangle \rightsquigarrow \langle P_3,\, G_3\rangle$ under the premise of $\langle C,\, G_3\rangle \rightsquigarrow^+ H$. Since P contains $k+1$ `if`, `try`, or `!` statements, C contains at most k of them. So by the induction hypothesis, there is a transition sequence $\langle C,\, [G_3, S]\rangle \to^* [H, S]$ (for any graph stack S). We can decompose this transition sequence into $\langle C,\, [G_3, S]\rangle \to^l \langle C_4,\, S_4\rangle \to [H, S]$ where $l \geq 0$, and S_4 is a graph stack. This fulfills the premise of [if$_2$] l times, and then the premise of [if$_3$] once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle \texttt{if}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, [G_3, S']\rangle\ &\to_{[\mathrm{if}_1]} \langle \mathrm{ITE}(C, P_3, Q),\, [G_3, G_3, S']\rangle\\ &\to^l_{[\mathrm{if}_2]} \langle \mathrm{ITE}(C_4, P_3, Q),\, S_4\rangle\\ &\to_{[\mathrm{if}_3]} \langle P_2, [G_3, S']\rangle.\end{aligned}$$

If [r] = [if$_2$], we have $\langle \texttt{if}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, G_3\rangle \rightsquigarrow \langle Q,\, G_3\rangle$ under the premise of $\langle C,\, G_3\rangle \rightsquigarrow^+$ fail. Again, we can use the induction hypothesis to conclude $\langle C,\, [G_3, S]\rangle \to^*$ fail (for any graph stack S), which is a sequence of $l \geq 0$ transitions $\langle C,\, [G_3, S]\rangle \to^l \langle C_4,\, S_4\rangle$ between extended configurations, followed by 1 transition $\langle C_4,\, S_4\rangle \to$ fail. This fulfills the premise of [if$_2$] l times, and then the premise of [if$_4$] once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle \texttt{if}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, [G_3, S']\rangle &\to_{[\mathrm{if}_1]} \langle \mathrm{ITE}(C,\, P_3,\, Q),\, [G_3, G_3, S']\rangle\\ &\to^l_{[\mathrm{if}_2]} \langle \mathrm{ITE}(C_4,\, P_3,\, Q),\, S_4\rangle\\ &\to_{[\mathrm{if}_4]} \langle Q,\, [G_3, S']\rangle.\end{aligned}$$

If [r] = [try$_1$], we can use the same arguments as when [r] = [if$_1$] to get a transition sequence $\langle \texttt{try}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, [G_3, S']\rangle \to^+ \langle P_3,\, [G_4, S']\rangle$ for any graph stack S'.

If [r] = [try$_2$], we can use the same arguments as when [r] = [if$_2$] to get a transition sequence $\langle \texttt{try}\ C\ \texttt{then}\ P_3\ \texttt{else}\ Q,\, [G_3, S']\rangle \to^+ \langle Q,\, [G_3, S']\rangle$ for any graph stack S'.

If [r] = [alap$_1$], we have $\langle P_3!\,, G_3\rangle \rightsquigarrow \langle P_3!\,, H\rangle$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^+ H$. Since P contains $k+1$ `if`, `try`, or `!` statements, P_3 contains at most k of them. So by the induction hypothesis, we can conclude $\langle P_3, [G_3, S]\rangle \rightarrow^* [H, S]$ for any graph stack S. We can decompose this transition sequence into $\langle P_3, [G_3, S]\rangle \rightarrow^l \langle P_4, S_4\rangle \rightarrow [H, S]$, where $l \geq 0$, and S_4 is a graph stack. These derivations fulfil the premise of [try$_2$] l times and then the premise of [try$_3$] once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!\,, [G_3, S']\rangle &\rightarrow_{[\text{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, [G_3, S']\rangle\\ &\rightarrow_{[\text{try}_1]} \langle \text{TRY}(P_3, P_3!\,, \texttt{skip}), [G_3, G_3, S']\rangle\\ &\rightarrow^l_{[\text{try}_2]} \langle \text{TRY}(P_4, P_3!\,, \texttt{skip}), S_4\rangle\\ &\rightarrow_{[\text{try}_3]} \langle P_3!\,, [H, S']\rangle.\end{aligned}$$

If [r] = [alap$_2$], we have $\langle P_3!\,, G_3\rangle \rightsquigarrow G_3$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^+$ fail. Again, we can use the induction hypothesis to conclude $\langle P_3, [G_3, S]\rangle \rightarrow^*$ fail for any graph stack S. We can decompose this transition sequence into $\langle P_3, [G_3, S]\rangle \rightarrow^l \langle P_4, [G_4, S]\rangle \rightarrow$ fail, where $l \geq 0$, and G_4 is a graph. Let us argue that the graph stack right before the fail is $[G_4, S]$. The stack ends in S because P_3 is a command sequence and hence does not contain ITE or TRY statements, which are the only constructs that could pop and hence modify S. Any `if`, `try`, and `!` statements in P_3 must resolve before the configuration that leads to fail in one step because fail is not reachable in one step from `if`, `try`, `!`, ITE, and TRY statements. Since they all resolved, any push has a corresponding pop, meaning the number of graphs in the stack before S remains 1. The derivations fulfil the premise of [try$_2$] l times and then the premise of [try$_4$] once. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!\,, [G_3, S']\rangle &\rightarrow_{[\text{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P!\ \texttt{else}\ \texttt{skip}, [G_3, S']\rangle\\ &\rightarrow_{[\text{try}_1]} \langle \text{TRY}(P_3, P_3!\,, \texttt{skip}), [G_3, G_3, S']\rangle\\ &\rightarrow^l_{[\text{try}_2]} \langle \text{TRY}(P_4, P_3!\,, \texttt{skip}), [G_4, G_3, S']\rangle\\ &\rightarrow_{[\text{try}_4]} \langle \texttt{skip}, [G_3, S']\rangle\\ &\rightarrow_{[\text{skip}]} [G_3, S'].\end{aligned}$$

If [r] = [alap$_3$], we have $\langle P_3!\,, G_3\rangle \rightsquigarrow H$ under the premise of $\langle P_3, G_3\rangle \rightsquigarrow^*$ $\langle \texttt{break}, H\rangle$. Again, we can use the induction hypothesis to conclude $\langle P_3, [G_3, S]\rangle \rightarrow^* \langle \texttt{break}, [H, S]\rangle$ (for any graph stack S). Let $l \geq 0$ be the number of steps in that transition sequence. These derivations fulfil the premise of [try$_2$] l times. So for any graph stack S', the premises are fulfilled by choosing $S = [G_3, S']$, and we have

$$\begin{aligned}\langle P_3!\,,\,[G_3,S']\rangle &\rightarrow_{[\text{alap}_1]} \langle \texttt{try}\ P_3\ \texttt{then}\ P_3!\ \texttt{else}\ \texttt{skip},\ [G_3,S']\rangle\\ &\rightarrow_{[\text{try}_1]} \langle \text{TRY}(P_3,\ P_3!\,,\texttt{skip}),\ [G_3,G_3,S']\rangle\\ &\rightarrow^{l}_{[\text{try}_2]} \langle \text{TRY}(\texttt{break},P_3!\,,\texttt{skip}),\ [H,G_3,S']\rangle\\ &\rightarrow_{[\text{alap}_2]} [H,S']. \qquad \square\end{aligned}$$

Lemma 2.48 (Configurations That Get Stuck)**.** *Let $\langle P,G\rangle$ be a configuration to which no inference rule of the original semantics is applicable. Then P starts with an* `if`*,* `try`*, or* `!` *statement such that the condition (or the body in the case of* `!`*) diverges or gets stuck in the original semantics.*

Proof. For this proof, whenever we say a rule is applicable, we mean it is either applicable, or it can be used as a premise for a [seq] rule.

Let us first show that, if P does not start with an `if`, `try`, or `!` statement, we can apply an inference rule to $\langle P,G\rangle$. If P starts with a rule set, that rule set is either applicable to the host graph or not, so either [call$_1$] or [call$_2$] can be applied. If P starts with a `break`, `skip`, or `fail`, then [break], [skip], or [fail] can be applied respectively. If P starts with an `or` statement, either [or$_1$] or [or$_2$] can be applied.

Now assume that P starts with an `if`, `try`, or `!` statement with a condition or body C. If $\langle C,G\rangle$ neither converges nor gets stuck, there is a transition sequence $\langle C,G\rangle \rightsquigarrow$ fail, or $\langle C,G\rangle \rightsquigarrow H$ for some graph H. Hence one of [if$_1$], [if$_2$], [try$_1$], [try$_2$], [alap$_1$], or [alap$_2$] is applicable. $\square$

Lemma 2.49 (Loop-Free Command Sequences Do Not Get Stuck)**.** *For every loop-free command sequence P and graph G, no transition sequence starting with $\langle P,G\rangle$ gets stuck in the original semantics.*

Proof. We prove this lemma by structural induction. For a base case, consider programs consisting of a single rule set R. Then on any graph G, R is either applicable or not. So to $\langle R,G\rangle$ we can apply either [call$_1$] or [call$_2$], leading to a graph or fail. They are both transition sequences that end in a terminal state, and hence do not get stuck.

Another base case is `skip` or `fail`, which always lead to a graph or fail in a single step, and hence cannot lead to stuck transition sequences.

The `break` statement cannot be in P since context conditions require it to have an enclosing loop.

For the induction step, we assume that every proper subprogram of P cannot get stuck, and show that P cannot get stuck either.

Assume $P = P_1; P_2$. By the induction hypothesis, for any graph G, no transition sequence starting with $\langle P_1,G\rangle$ gets stuck, i.e. they can all be extended to either $\langle P_1,G\rangle \rightsquigarrow^+$ fail or $\langle P_1,G\rangle \rightsquigarrow^+ H$ for some graph H. This fulfils the premise of [seq$_1$] some number of times, and then the premise of either [seq$_2$] or [seq$_3$] once. The same applies to P_2. So each transition sequence staring with $\langle P, G\rangle$ must be of the form $\langle P, G\rangle \rightsquigarrow^+_{[\text{seq}]}$ fail or $\langle P, G\rangle \rightsquigarrow^+_{[\text{seq}]} H$.

Assume $P = \texttt{if}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. Then by induction hypothesis, for any graph G, no transition sequence starting with $\langle C, G\rangle$ can get stuck, i.e. they can all be extended to either $\langle C, G\rangle \rightsquigarrow^+$ fail or $\langle C, G\rangle \rightsquigarrow^+ H$ for some graph H. This satisfies the premise of either [if$_1$] or [if$_2$]. So each transition sequence staring with $\langle P, G\rangle$ must be of the form $\langle P, G\rangle \rightsquigarrow_{[\text{if}_1]} \langle P_1, G\rangle$ or $\langle P, G\rangle \rightsquigarrow_{[\text{if}_2]} \langle P_2, G\rangle$. Any continuation of these sequences cannot get stuck because P_1 and P_2 satisfy the induction hypothesis.

The case $P = \texttt{try}\ C\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ is analogous to the previous one.

If P is an `if` or `try` with omitted `then` or `else` clauses, one of [if$_3$], [try$_3$], [try$_4$], or [try$_5$] can be applied, and then the arguments used in the `if` and `try` cases can be applied.

Assume $P = P_1\ \texttt{or}\ P_2$. Then for any graph G, any transition sequence starting with $\langle P, G\rangle$ starts with either $\langle P, G\rangle \rightsquigarrow_{[\text{or}_1]} P_1$ or $\langle P, G\rangle \rightsquigarrow_{[\text{or}_2]} P_2$. Any continuation of these sequences cannot get stuck because P_1 and P_2 satisfy the induction hypothesis. □

Lemma 2.50 (Non-Nested Loops Do Not Get Stuck). *For every loop-free command sequence P and graph G, no transition sequence starting with $\langle P!, G\rangle$ gets stuck in the original semantics.*

Proof. It is enough to show that either $\langle P, G\rangle \rightsquigarrow^+ H$, $\langle P, G\rangle \rightsquigarrow^+$ fail, or $\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle$. Because then, one of [alap$_1$], [alap$_2$], or [alap$_3$] is applicable. And if [alap$_1$] was applicable, we get $\langle P!, G\rangle \rightsquigarrow \langle P!, H\rangle$, the same arguments can be used on $\langle P!, H\rangle$, and hence on all its successors.

First of all, since P contains no `!`, $\langle P, G\rangle$ cannot diverge. If P does not contain a `break`, $\langle P, G\rangle$ cannot get stuck by Lemma 2.49. If P does contain a `break`, that is never called, the arguments of the proof of Lemma 2.49 still apply, and $\langle P, G\rangle$ does not get stuck. If P contains a `break` that is called, that means there is a transition sequence $\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle$, or $\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}; Q, H\rangle$, to which we can apply [break] to get the former.

So either $\langle P, G\rangle \rightsquigarrow^* \langle \texttt{break}, H\rangle$, or $\langle P, G\rangle$ neither diverges nor gets stuck, which means $\langle P, G\rangle$ must resolve to either a graph or fail. □

Lemma 2.51 (Simulating Original Transition Sequences That Are Infinite or Stuck). *Assume there is an infinite transition sequence $\langle P, G\rangle \rightsquigarrow \ldots$, or a stuck transition sequence $\langle P, G\rangle \rightsquigarrow^* \langle P', G'\rangle$. Then for any graph stack S, there is an infinite transition sequence $\langle P, [G, S]\rangle \to \ldots$.*

Proof. First assume there is an infinite transition sequence $\langle P, G\rangle \rightsquigarrow \langle P_1, G_1\rangle \rightsquigarrow \ldots$. To each step $\langle P_i, G_i\rangle \rightsquigarrow \langle P_{i+1}, G_{i+1}\rangle$ in that transition sequence, we can apply Lemma 2.47 to get a transition sequence $\langle P_i, [G_i, S_i]\rangle \to^* \langle P_{i+1}, [G_{i+1}, S_{i+1}]\rangle$ for any graph stack S_i and some S_{i+1}. We can concatenate these into an infinite transition sequence.

Now assume there is a stuck transition sequence $\langle P, G\rangle \rightsquigarrow^* \langle P', G'\rangle$. We can apply Lemma 2.47 to each step in that sequence to get $\langle P, [G, S]\rangle \to^* \langle P', [G', S]\rangle$ for each graph stack S.

We claim that for any command sequence P' such that $\langle P', G'\rangle$ is stuck with respect to $\rightsquigarrow$, there is a diverging transition sequence $\langle P', [G', S]\rangle \to \ldots$ for any graph stack S. This is enough to prove the Lemma. Let us show this by induction on the combined number of `if`, `try`, and `!` statements in P'.

If there are no such statements, then $\langle P', G'\rangle$ cannot get stuck by Lemma 2.48, so there has to be at least one, and P' starts with it. If there is exactly one statement and that one statement is an `if` or a `try`, then $\langle P', G'\rangle$ cannot get stuck by Lemma 2.49. If it is a `!` statement, $\langle P', G'\rangle$ cannot get stuck by Lemma 2.50. So there have to be at least two `if`, `try`, or `!` statements.

So for our base case, assume P' contains exactly two `if`, `try`, or `!` statements. By Lemma 2.49, P' must contain a `!` statement. If the two statements are not nested, we can apply Lemmata 2.49 and/or 2.50 sequentially to conclude that $\langle P', G'\rangle$ does not get stuck. So the two statements must be nested, and they must be the start of P' by Lemma 2.48. Assume the "inner" statement is not a `!` statement. Then the "outer" statement must be the `!` statement. By Lemma 2.50, $\langle P', G'\rangle$ does not get stuck. So the "inner" statement must be a `!` statement Q!. The loop Q! cannot resolve to a graph or fail because then, the "outer" statement could be resolved and $\langle P', G'\rangle$ would not be stuck. By Lemma 2.50, Q! cannot get stuck either. So Q! must diverge, and so must the condition or body C of the starting statement of P'. So there is an infinite transition sequence $\langle C, G'\rangle \rightsquigarrow \ldots$, and hence by Lemma 2.47 to each step in that transition sequence, we get $\langle C, [G', S]\rangle \to \ldots$ for any graph stack S. This serves as a premise for $[\text{if}_2]$ or $[\text{try}_2]$, which we can use to get an infinite transition sequence $\langle P', G'\rangle \to \ldots$.

Now for the induction step, assume we get infinite transition sequences for programs with at most k `if`, `try`, and `!` statements. Assume P' has $k{+}1$ such statements. By Lemma 2.48, since $\langle P', G'\rangle$ is stuck, P' must start with one of those statements. So we can apply either $[\text{if}_1]$, $[\text{try}_1]$, or $[\text{alap}_1]$ followed by $[\text{try}_1]$ to get $\langle P', [G', S']\rangle \to^+ \langle P'', [G', G', S']\rangle$ for any graph stack S', where P'' starts with either $\text{ITE}(C, Q, Q')$ or $\text{TRY}(C, Q, Q')$. Now C has at least one less `if`, `try`, or `!` statement than P', so we can apply the induction hypothesis to get an infinite transition sequence $\langle C, [G', G', S']\rangle \to \ldots$, which can serve as premises for infinitely many applications of $[\text{if}_2]$ or $[\text{try}_2]$ (or these inference rules provide the premises for $[\text{seq}_1]$). Hence we have an infinite transition sequence $\langle P', [G', S']\rangle \to^+ \langle P'', [G', G', S']\rangle \to \ldots$. $\square$

Lemma 2.52 (Simulating Finite Small-Step Transition Sequences)**.** *Let $P \in$ CommandSeq, $G \in \mathcal{G}$, S a graph stack, and $X \in \{\langle P', [G', S']\rangle, [G', S'], \mathit{fail}\}$, where $P' \in$ CommandSeq, S' is a graph stack, and $G' \in \mathcal{G}$. If $\langle P, [G, S]\rangle \to^* X$, then there is a transition sequence*

- $\langle P, G\rangle \rightsquigarrow^* \langle P', G'\rangle$ *if* $X = \langle P', [G', S']\rangle$.
- $\langle P, G\rangle \rightsquigarrow^* G'$ *if* $X = [G', S']$.
- $\langle P, G\rangle \rightsquigarrow^* \mathit{fail}$ *if* $X = \mathit{fail}$.

Proof. Let us show this lemma by induction on the combined number of `if`, `try`, and `!` statements in P. If P has no such statements, no step in $\langle P, [G, S]\rangle \to^* X$ uses [if], [try], or [alap] rules. So no pushing or popping occurs and only the top of the stack is modified. The applied rules behave identically to those in the original semantics when identifying the top of the stacks in the small-step rules with the host graphs in the original rules. Hence the lemma is satisfied in the base case.

Now assume that the lemma holds for command sequences with k `if`, `try`, and `!` statements, and assume P has $k + 1$ of them. By the previous paragraph, we can simulate transition steps that do not involve [if], [try], or [alap] rules. So let $\langle Q, [H, S']\rangle$ be a configuration (we will handle the case where $\langle Q, [H, S']\rangle$ is an extended configuration, but not a configuration later) in the sequence $\langle P, [G, S]\rangle \to^* X$ that uses such a rule to get the next state. Let [r] be that rule. It can only be [if$_1$], [try$_1$], or [alap$_1$] since $\langle Q, [H, S']\rangle$ is a configuration and hence does not contain ITE or TRY.

If [r] = [if$_1$], then we get

$$\langle Q, [H, S']\rangle \to \langle \mathrm{ITE}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_4; Q_3, [H, S']\rangle,$$

where Q_1, Q_2, and Q_3 are command sequences, and where $Q_4 \in \{Q_1, Q_2\}$ (since this is part of a transition sequence that ends in X, a configuration, graph, or fail, we know that the `if` eventually resolves). So either $\langle C, [H, H, S']\rangle \to^* [H', H, S']$ or $\langle C, [H, H, S']\rangle \to^*$ fail. We can apply the induction hypothesis to C, which has at most k `if`, `try`, and `!` statements, to get that either $\langle C, H\rangle \rightsquigarrow^* H'$ or $\langle C, H\rangle \rightsquigarrow^*$ fail. We can use this as a premise for either [if$_1$] or [if$_2$] to get $\langle Q, H\rangle \rightsquigarrow \langle Q_4; Q_3, H\rangle$, where $Q_4 \in \{Q_1, Q_2\}$, which simulates the sequence outlined at the beginning of this case.

If [r] = [try$_1$], then we get either

$$\langle Q, [H, S']\rangle \to \langle \mathrm{TRY}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_1; Q_3, [H', S']\rangle \text{ or}$$

$$\langle Q, [H, S']\rangle \to \langle \mathrm{TRY}(C, Q_1, Q_2); Q_3, [H, H, S']\rangle \to^+ \langle Q_2; Q_3, [H, S']\rangle,$$

where Q_1, Q_2, and Q_3 are command sequences, and H' some graph. We can use the same arguments as in the previous case to get a transition sequence $\langle Q, H\rangle \rightsquigarrow \langle Q_1; Q_3, H'\rangle$ or $\langle Q, H\rangle \rightsquigarrow \langle Q_2; Q_3, H\rangle$.

If [r] = [alap$_1$], we have $Q = Q_1!\,; Q_2$ so either

$$\langle Q_1!\,; Q_2, [H, S']\rangle \to^* \langle Q_1!\,; Q_2, [H', S']\rangle,$$

$$\langle Q_1!\,; Q_2, [H, S']\rangle \to^* \langle Q_2, [H, S']\rangle, \text{ or}$$

$$\begin{aligned}\langle Q_1!\,; Q_2, [H, S']\rangle &\to^* \langle \mathrm{TRY}(\texttt{break}, Q_1!\,, \texttt{skip}); Q_2, [H', H, S']\rangle \\ &\to \langle Q_2, [H', S']\rangle.\end{aligned}$$

We can conclude that either $\langle Q_1, [H, S']\rangle \to^* [H', S']$, $\langle Q_1, [H, S']\rangle \to^*$ fail, or $\langle Q_1, [H, S']\rangle \to^* \langle \texttt{break},\ [H', S']\rangle$. By induction hypothesis, we get that either $\langle Q_1, H\rangle \rightsquigarrow^* H'$, $\langle Q_1, H\rangle \rightsquigarrow^*$ fail, or $\langle Q_1, H\rangle \rightsquigarrow^* \langle \texttt{break}, H'\rangle$. These can be used as premises of [alap$_1$], [alap$_2$], or [alap$_3$] to conclude that either

$$\langle Q_1!\,; Q_2, H\rangle \rightsquigarrow \langle Q_1!\,; Q_2, H'\rangle,$$

$$\langle Q_1!\,; Q_2, H\rangle \rightsquigarrow \langle Q_2, H\rangle, \text{ or}$$

$$\langle Q_1!\,; Q_2, H\rangle \rightsquigarrow \langle Q_2, H'\rangle.$$

Finally, if $\langle Q, [H, S']\rangle$ is an extended configuration, but not a configuration, Q must contain an ITE, TRY, or a `break` that does not satisfy the context conditions. All of these must originate in an `if`, `try`, or `!` statement, and are hence covered by the previous part of the proof. □

Note that the first point of Lemma 2.52 only applies to transition sequences between command sequences (they do not contain TRY or ITE constructs). So diverging transition sequences in the small-step semantics where all command sequences contain ITE or TRY after some step cannot necessarily be simulated with diverging transition sequences in the original semantics.

Theorem 2.53. *Let $P \in$ CommandSeq and $G \in \mathcal{G}$. Then*

(a) $[P]G \subseteq [\![P]\!]G$ *and*

(b) $[P]G \setminus \{\bot\} = [\![P]\!]G \setminus \{\bot\}$.

Proof. Lemma 2.47 guarantees that (a) holds for graphs and fail, and Lemma 2.51 guarantees that (a) holds for $\bot$. Furthermore, (b) follows from (a) and Lemma 2.52. □

Definition 2.54 (Termination)**.** A program P is *terminating* if for all graphs G, $\bot \notin [\![P]\!]G$. □

Note that in the small-step semantics, $\perp \in [\![P]\!]G$ corresponds to there being an infinite transition sequence starting in $\langle P, [G] \rangle$ by Theorem 2.42 and the definition of the semantic function. This is not the case for the original semantics, which can also get stuck if $\perp \in [P]G$.

2.3.9 Related Work: Semantics

The Nielson and Nielson book [55] gives an overview of different approaches of defining the semantics of a programming language.

Operational semantics describe how the state changes during program execution, either overall which we call *big-step*, or in individual steps which we call *small-step*. This gives rise to a sequence of transitions that not only describes input and output relations, but also how the program got there. Plotkin [57] builds up his structural operational semantics using inference rules. This is the chosen approach for the GP 2 semantics since, unlike others, it describes program execution in step-wise detail, making it especially useful for time complexity analysis.

Denotational semantics are based exclusively on inputs and outputs of programs, as described by a semantic function. This function is defined compositionally, which means to every syntactic element, we associate a mathematical object, usually a function. To programs, we associate a composition of those objects. There are denotational semantics for Standard ML [91] and Scheme [22], a dialect of Lisp. While GP 2 has a semantic function, it is not defined compositionally.

Axiomatic semantics are a very different approach where we care about program verification like the one proposed by Hoare [44]. In this context, we care about formal pre- and post-conditions on a program execution. While there is substantial research on formal verification in GP 2 [63, 64, 94, 95], it is out of the scope of this thesis since we focus on complexity. Furthermore, axiomatic semantics are generally needed for reactive programs, which GP 2 does not have.

The only other graph transformation language with formal semantics we are aware of is PROGRES [72]. Its semantics, given in [70], is based on programmed graph replacement systems [71].

The semantics only depends on input and output graphs. A grammar in Extended Backus-Naur form [73] generates control-flow diagrams that describe the execution of a program.

The downside is that over 300 rules are used to define the semantics, while GP 2 uses around a dozen rules without sacrificing formality or the ability to describe nondeterminism. Instead of a control-flow diagram, we get a transition sequence that describes how the the state of program and host graph changes.

3 Time and Space Complexity

In this chapter, we give a framework for analysing the time and space complexity of a GP 2 program. We do this in terms of worst-case asymptotic complexity using the big-O notation. As is customary in complexity analysis, the program is considered to be fixed, and complexity is measured in terms of graph size, which we define in Subsection 3.3. Essentially, it is the number of nodes, edges, and the sum of the sizes of all labels.

In some cases, one might wish to count integer labels as part of the size, like in problems where integers are part of the input. Consider for example a program constructing an $n \times n$ grid starting from a single node labelled with the integer n. If n is not considered part of the input size, input graphs would have the same size, and any asymptotic complexity analysis would be meaningless. In this thesis however, we only consider problems that do not require such an integer input.

In Section 3.1, we briefly talk about the GP 2-to-C compiler we use to run GP 2 programs, and as a starting point to analyse complexity. We describe how it handles nondeterminism in Subsection 3.1.1, outline its data structures in Subsection 3.1.2, and state a few assumptions on its runtime complexity in Subsection 3.1.3.

In Section 3.2, we discuss the complexity of applying a GP 2 rule. First, we give the abstract matching algorithm for rules in Subsection 3.2.1. We then analyse how matching is implemented in the GP 2 compiler in Subsection 3.2.2. Using these algorithms, we then conclude that matching is constant-time for fast rules under mild conditions in Subsection 3.2.3, outgoing fast rules in Subsection 3.2.4, and some other rules in Subsections 3.2.5 and 3.2.6.

Next, in Section 3.3, we analyse what conditions programs and input graphs must fulfil in order for us to define complexity measures of GP 2 programs, and we define these measures.

We finish this chapter in Section 3.4 by talking about different approaches to matching in the literature, including the topics incremental matching and dynamic search plans.

3.1 The GP 2-to-C Compiler

GP 2 comes with a compiler that takes a program and an input graph, and turns it into C code that can be executed [8, 11]. The aim of this section is to the describe this compiler and based on that, to make assumptions about the complexity of its low-level operations. These assumptions provide the foundation for our complexity theory.

We believe the implementation is sound with respect to the semantics. There is also a Haskell interpreter [9], but in this thesis, we focus solely on the C compiler for efficiency reasons.

3.1.1 Nondeterminism

GP 2 is nondeterministic in two ways. A rule can have multiple matches, which we call *match nondeterminism*. And when a set of rules is called, several rules could be applicable, which we call *rule nondeterminism*. The `or` construct is a derived command, i.e. it is semantically equivalent to a sequence of other commands [58]. Hence we do not consider it another type of nondeterminism.

Instead of giving every possible nondeterministic result, the implementation of GP 2 chooses a nondeterministic path. Whenever there is a nondeterministic choice, the implementation picks the first one that does not fail (which happens when a rule has no match for instance). This approach is picked for the speed of the implementation, since backtracking is costly. It is up to the programmer to write programs so that it does not matter which nondeterministic path is picked. Otherwise, this local choice of a nondeterministic path may lead to non-termination, even though a terminating path exists.

This kind of relation between theory and implementation is not unique to GP 2. The Prolog implementation for instance adds a construct that allows cutting off nondeterministic paths, sacrificing nondeterministic completeness for efficiency [5]. In fact, GP 2 compromises less since no constructs are cut, its implementation just picks a nondeterministic path.

3.1.2 Data Structures

As described in more detail in [20], graphs consist of three components: the array of nodes, the array of edges, and a linked list of nodes named `NodeList`.

Each entry of `NodeList` consists of a pointer to a node in the array of nodes, and a pointer to the next entry of `NodeList`. This linked list skips over holes in the node array left by deletion, and allows deletion of all nodes and edges of a graph to happen in linear time.

Nodes contain their in- and outdegree (number of incoming and outgoing edges), and a linked list of outgoing edges, and one of incoming edges. This is done to speed up finding an edge of either type during matching, since it limits the search space.

As described in more detail in [8], host graph labels are stored as an enumeration of marks and a doubly-linked list whose entries are C strings and integers. This corresponds to how GP 2 labels are defined.

Strings are stored as C strings. However, concatenations of strings are stored as doubly linked lists for the sake of efficiency during string comparison.

Morphisms are represented using arrays of node identifiers, edge identifiers, as well as an array and stack keeping track of assignments.

3.1.3 Complexity Assumptions

As a basis for a GP 2 complexity model, we make a few assumptions about the runtime of low-level operations, shown in Figure 3.1, and first made in [18]. These assumptions are made on which worst-case asymptotic complexity class the operations belong to, where n is the size of a graph, as defined below. In Definition 3.24 of a later section, worst-case asymptotic time complexity classes are defined for GP 2 programs.

Definition 3.1 (Graph Size)**.** The size of a graph G is defined as

$$|G| = |V_G| + |E_G| + \sum_{v \in \mathrm{dom}(l_G)} |l_G(v)| + \sum_{e \in E_G} |m_G(e)|,$$

where the size of a label l is defined as

$$|l| = \begin{cases} 1 & \text{if } l \text{ is an integer or character,} \\ \mathrm{len}(l) & \text{if } l \text{ is a string,} \\ \sum_{i=1}^{n} |a_i| & \text{if } l = a_1 : a_2 : \cdots : a_n \text{ is a list.} \end{cases}$$

□

Note that we assign unit cost to integers. While it is perfectly reasonable to charge non-constant cost to integer labels, we do not do so here since no problem presented in this thesis features unbounded integer labels.

Definition 3.2 (Asymptotic Complexity Class)**.** Let $f, g\colon \mathbb{N} \to \mathbb{R}$ be functions from non-negative integers to real numbers. We say $f \in \mathrm{O}(g)$ or $f(n) = \mathrm{O}(g(n))$ if there is a constant $c \in \mathbb{N}$ such that for all $n \in \mathbb{N}$, $f(n) \leq c \cdot g(n)$. □

Procedure	Description	Complexity
`alreadyMatched`	Test if the given item has been matched in the host graph.	$\mathrm{O}(1)$
`clearMatched`	Clear the `is matched` flag for a given item.	$\mathrm{O}(1)$
`setMatched`	Set the `is matched` flag for a given item.	$\mathrm{O}(1)$
`firstHostNode`	Fetch the first node in the host graph.	$\mathrm{O}(1)$
`nextHostNode`	Given a node, fetch the next node in the host graph.	$\mathrm{O}(1)$
`firstHostRootNode`	Fetch the first root node in the host graph.	$\mathrm{O}(1)$
`nextHostRootNode`	Given a root node, fetch the next root node in the host graph.	$\mathrm{O}(1)$
`firstInEdge`	Given a node, fetch the first incoming edge.	$\mathrm{O}(1)$
`nextInEdge`	Given a node and an edge, fetch the next incoming edge.	$\mathrm{O}(1)$
`firstOutEdge`	Given a node, fetch the first outgoing edge.	$\mathrm{O}(1)$
`nextOutEdge`	Given a node and an edge, fetch the next outgoing edge.	$\mathrm{O}(1)$
`getInDegree`	Given a node, fetch its incoming degree.	$\mathrm{O}(1)$
`getOutDegree`	Given a node, fetch its outgoing degree.	$\mathrm{O}(1)$
`getMark`	Given a node or edge, fetch its mark.	$\mathrm{O}(1)$
`isRooted`	Given a node, determine if it is rooted.	$\mathrm{O}(1)$
`getSource`	Given an edge, fetch the source node.	$\mathrm{O}(1)$
`getTarget`	Given an edge, fetch the target node.	$\mathrm{O}(1)$
`updateMap`	Given a map, add a single node or edge mapping.	$\mathrm{O}(1)$
`updateAssignment`	Given a variable assignment, assign a single label to a variable.	$\mathrm{O}(1)$
`list.first`	Access the first element of a list label.	$\mathrm{O}(1)$
`list.last`	Access the last element of a list label.	$\mathrm{O}(1)$
`listElement.next`	Access the next element of a list label.	$\mathrm{O}(1)$
`listElement.prev`	Access the previous element of a list label.	$\mathrm{O}(1)$
`list.length`	Determine the length of a list label.	$\mathrm{O}(1)$
`string.first`	Access the first character of a string.	$\mathrm{O}(1)$
`string.last`	Access the last character of a string.	$\mathrm{O}(1)$
`stringChar.next`	Access the next character of a string.	$\mathrm{O}(1)$
`stringChar.prev`	Access the previous character of a string.	$\mathrm{O}(1)$
`string.length`	Determine the length of a string.	$\mathrm{O}(1)$
`expression.type`	Checking whether an expression is of a certain type.	$\mathrm{O}(1)$
`expression.var`	Checking whether an expression is a variable of a certain type.	$\mathrm{O}(1)$
`parseInputGraph`	Parse and load the input graph into memory: the host graph.	$\mathrm{O}(n)$
`printHostGraph`	Write the current host graph state as output.	$\mathrm{O}(n)$

Figure 3.1: Assumptions on the complexity of various compiler operations

Empirical time measurements of GP 2 programs in [19] and Chapter 4 are evidence in support of these complexity assumptions. These time measurements include the time it takes to read, parse and load the input graph into memory, but not the time it takes to compile the program.

Figure 3.1 shows assumptions that we make on the time complexity of some low-level compiler operations. This is originally done by Graham Campbell in [19], and expanded upon with more operations by this thesis' author.

We also assume that operations on integers in C take constant time. Only in applications that require computation on large integers is a more sensitive approach needed [2]. We make the unit time assumption in the scope of this thesis for simplicity and since the focus of GP 2 is graph computation, hence it is more likely a programmer will compute large graphs rather than large labels.

Concatenation of strings and lists can be done in constant time since they are stored as doubly-linked lists, and we assume accessing their pointers takes constant time. Moreover, we assume the length of lists and strings are stored as integers within a data structure, and as such take constant time to access.

Checking whether expressions are of type `int`, `char`, `string`, or `atom`, and whether an expression is a variable of such a type is also assumed to take constant time, since the information is stored as part of the data structure for expressions.

We assume that parsing an input graph, as well as outputting a host graph takes linear time. Note that this means program execution cannot take sublinear time.

3.2 The Complexity of Rule Application

A crucial step of a GP 2 program is *rule application*, which consists of matching the left-hand side L of a rule into a host graph G, and of making the relevant changes to the host graph. Matching in particular can be very inefficient in general, so implementing an efficient matching algorithm is crucial.

3.2.1 The Abstract Matching Algorithm

The goal of matching is, given a rule $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ and a host graph G, find all variable assignments α and all matches g for the instantiated rule $r^{g,\alpha}$ such that $c^{g,\alpha}$ evaluates to true. Remember matches are injective morphisms that satisfy the dangling condition. Let us first define some notions used by the matching algorithm.

Definition 3.3 (Partial Morphism and Premorphism)**.** Given two graphs G and H, *partial morphism* $g\colon G \xrightarrow{par} H$ is a pair $g = \langle g_V, g_E\rangle$ of partial functions $g_V\colon V_G \xrightarrow{par} V_H$ and $g_E\colon E_G \xrightarrow{par} E_H$ such that the defining properties of graph morphisms in Definition 2.5 hold for edges in $\mathrm{dom}(g_E)$ and nodes in $\mathrm{dom}(g_V)$.

We define a *partial premorphism* analogously. □

Note that a partial morphism g with $\text{dom}(g_E) = E_G$ and $\text{dom}(g_V) = V_G$ is a morphism.

Definition 3.4 (Extension of a Partial Premorphism)**.** Let $f, g\colon G \xrightarrow{par} H$ be partial premorphisms such that, for all nodes $v \in \text{dom}(g_V)$ we have $g_V(v) = f_V(v)$, and for all edges $e \in \text{dom}(g_E)$ we have $g_E(e) = f_E(e)$.

- We say f *extends* g by a node $v \in V_G$ if $\text{dom}(f_V) = \text{dom}(g_V) \cup \{v\}$ and $\text{dom}(f_E) = \text{dom}(g_E)$.
- We say f *extends* g by an edge $e \in E_G$ if $\text{dom}(f_E) = \text{dom}(g_E) \cup \{e\}$ and $\text{dom}(f_V) = \text{dom}(g_V)$. □

Definition 3.5 (Partial Variable Assignment)**.** A *partial variable assignment* is a variable assignment whose functions are partial functions.

By $\alpha_\emptyset$, we denote a variable assignment whose functions are defined on $\emptyset$. We call a partial variable assignment whose functions are total a *total variable assignment.* □

Definition 3.6 (Set of Roots)**.** Given a graph G, we define the *set of roots* of G as $P_G = \{v \in V_G \,|\, p_G(v) = 1\}$. □

Definition 3.7 (Edge Enumeration)**.** Given a graph L , a node $p \in V_L$, and a set of nodes $\{e_1, \dots, e_n\} \subseteq E_L$ reachable from p via undirected paths, an *edge enumeration* $e_1, \dots, e_n$ for p is a list of edges such that $s(e_1) = p$ or $t(e_1) = p$, and for each $i \geq 2$, either $s(e_i)$ or $t(e_i)$ is the source or target of some edge in $e_1, \dots, e_{i-1}$. □

Note that edges from an edge enumeration, along with their incident nodes, form an entire connected component of the graph they are contained in.

Algorithm 3.1 and its procedures, Algorithms 3.2, 3.3, and 3.4, form the basis of the GP 2 graph matching algorithm. The algorithm, along with a correctness proof can be found in [8]. Note that it is only for rules that have no condition, and only nodes reachable (via a path of edges of arbitrary direction) from a root. We explore how other rules are matched in Subsection 3.2.2.

Graph Matching (Algorithm 3.1) finds matches for a graph L in a graph G. It builds up partial premorphisms and partial variable assignments, comparing graph structure and labels, until they become injective graph morphisms and total assignments, i.e. matches. As roots are matched, they are

Algorithm 3.1: Graph Matching

```
Input: Graphs $L$ and $G$; and for each root $p \in P_L$, an edge
       enumeration $e_{p_1}, \dots, e_{p_n}$
Output: The set $A$ of all pairs of injective graph morphisms
        $L \to G$, and total assignments $\alpha$
1  A $\leftarrow \{\langle h \colon L \xrightarrow{par} G, \alpha_\emptyset \rangle \mid \mathrm{dom}(h) = \emptyset\}$
2  while there is an untagged root $p \in P_G$ do
3      $A_0 \leftarrow \{\langle h \colon L \xrightarrow{par} G, \alpha_{h'} \rangle \mid h$ is injective, and there exists
           $\langle h', \alpha_{h'} \rangle \in A$ such that $h$ extends $h'$ by $p\}$
4      tag $p$
5      Update Assignment($A_0$)    // this is Algorithm 3.2
6      for $i = 1$ to $n$ do
7          $A_i \leftarrow \{\langle h \colon L \xrightarrow{par} G, \alpha_{h'} \rangle \mid h$ is injective, and there exists
               $\langle h', \alpha_{h'} \rangle \in A_{i-1}$ such that $h$ extends $h'$ by $e_{p_i}\}$
8          if $s(e_{p_i}) \in P_L$ then tag $s(e_{p_i})$
9          if $t(e_{p_i}) \in P_L$ then tag $t(e_{p_i})$
10         Update Assignment($A_i$)
11     $A \leftarrow A_n$
12 return $A$
```

tagged. The while loop in line 2 iterates over untagged roots p. Partial premorphisms are extended with nodes and edges in order of appearance in the edge enumeration for p. If multiple extensions are possible, they are added as separate partial premorphisms. After each extension, the procedure Update Assignment is called, which extends variable assignments, compares labels, and removes partial premorphisms whose labels do not match. Since every node of L is assumed to be reachable from a root, the edge enumerations cover all of L. So in the end, premorphisms become morphisms and partial assignments become total.

The purpose of Update Assignment (Algorithm 3.2) is to remove pairs of partial premorphisms and partial assignments that make labels mismatch, and to update the remaining assignments such that labels match. The removal happens by calling the Reject procedure (Algorithm 3.3), which deletes the current pair from the set of potential matches and exits the enclosing for loop (which is always the one starting in line 2, iterating over items for which the partial premorphism is defined). Line 4 makes sure the marks match.

Comparing the labels comes next. Since labels are lists in general, we match the rule list with a host list atom by atom. Since we require labels of

Algorithm 3.2: Update Assignment

```
Input: A set $A$ of pairs of partial injective graph premorphisms and
       partial assignments
Output: A set of pairs of injective graph morphisms and partial
        assignments
1  for each $\langle h, \alpha \rangle \in A$ do
2      for each untagged item $l \in \mathrm{dom}(h_V)$ or $l \in \mathrm{dom}(h_E)$ do
3          $x \leftarrow \mathrm{label}(l)$; $y \leftarrow \mathrm{label}(h(l))$
4          if $\mathrm{mark}(x) \neq \mathrm{mark}(y)$ or ($\mathrm{mark}(x) = \texttt{any}$ and
            $\mathrm{mark}(y) = \texttt{none}$) then Reject // Reject is Algorithm 3.3
5          atom $a \leftarrow x$.first; atom $b \leftarrow y$.first
6          while $a \neq$ NULL do
7              if $a \in$ LVar then break
8              if $b =$ NULL then Reject
9              if $\neg\mathrm{Check}(a, b, \alpha)$ then Reject // Check is Algorithm 3.4
10             $a \leftarrow a$.next; $b \leftarrow b$.next
11         if $a =$ NULL then
12             if $b =$ NULL then exit else Reject
13         else
14             atom temp $\leftarrow b$
15             $a \leftarrow x$.last; $b \leftarrow y$.last
16             while $a \notin$ LVar do
17                 if $\&b = \&$temp.prev then Reject
18                 if $\neg\mathrm{Check}(a, b, \alpha)$ then Reject
19                 $a \leftarrow a$.prev; $b \leftarrow b$.prev
20             $\alpha(a) \leftarrow$ list starting with temp and ending in $b$
21             exit
22         tag $l$
```

Algorithm 3.3: Reject

```
1  $A \leftarrow A - \{\langle h, \alpha \rangle\}$
2  exit enclosing for loop
```

rules to be simple, a list can only have one list variable (LVar). List variables can match lists of arbitrary length, so having several in the same list would create ambiguity. Hence the strategy for matching lists is to first match any atoms on the left of the list variable, then any atoms on the right of the list

Algorithm 3.4: Check

```
   Input: An atomic expression $a$ in a left-hand side label, an atom $b$
          in a host graph label, and a partial assignment $\alpha$
   Output: True if $a$ and $b$ can be matched, false otherwise
1  switch $a$ do
2      case $a \in \mathbb{Z}$ do
3          if $b \in \mathbb{Z}$ then return $(a = b)$ else return false
4      case $a \in \mathrm{Char}^*$ do
5          if $b \in \mathrm{Char}^*$ then return $(a = b)$ else return false
6      case $a \in \mathrm{CVar}$ do
7          if $b \in \mathrm{Char}$ then $\alpha(a) \leftarrow b$; return true else return false
8      case $a \in \mathrm{IVar}$ do
9          if $b \in \mathbb{Z}$ then $\alpha(a) \leftarrow b$; return true else return false
10     case $a \in \mathrm{AVar}$ do
11         if $b \in \mathbb{Z} \cup \mathrm{Char}^*$ then $\alpha(a) \leftarrow b$; return true
12         else return false
13     case $a = w.s.w'$, where $s \in \mathrm{SVar})$, $w, w' \in (\mathrm{CVar} \cup \mathrm{Char})^*$ do
14         if $b \in \mathbb{Z}$ then return false
15         char $c \leftarrow a$.first; char $d \leftarrow b$.first
16         while $c \neq s$ do
17             if $d = \mathrm{NULL}$ then Reject
18             if $c \in \mathrm{CVar}$ then $\alpha(c) \leftarrow d$
19             else if $c \neq d$ then return false
20             $c \leftarrow c$.next; $d \leftarrow d$.next
21         char temp $\leftarrow d$; $c \leftarrow a$.last; $d \leftarrow b$.last
22         while $c \neq s$ do
23             if $\&d = \&$temp.prev then return false
24             if $c \in \mathrm{CVar}$ then $\alpha(c) \leftarrow d$
25             else if $c \neq d$ then return false
26             $c \leftarrow c$.prev; $d \leftarrow d$.prev
27         $\alpha(a) \leftarrow b$
28         return true
```

variable, and any remaining atoms in the host list are assigned to the list variable.

The while loop in line 6 iterates through the atoms from left to right until it finds a list variable or reaches the end of the list. Line 8 rejects if the host list is too short. Line 9 rejects if the atoms do not match, which is done by the procedure Check. This is also where assignment of non-list variables happens. By the end of the while loop, all atoms before a list variable have been matched.

In line 11, we distinguish between whether there is a list variable or not. If there is no list variable, the previous while loop must have terminated because a = NULL. If this is the case, we have successfully matched the entire list only if we have reached the end of the host list, which line 12 ensures. In the else branch, we do have a list variable. Here we match atom by atom from right to left. Line 17 makes sure the host list is not too short, which happens if the current host atom has already been matched by a rule atom to the left of the list variable.

The procedure Check (Algorithm 3.4) compares a rule atom and a host atom for whether they match. It returns true if they match, and false otherwise. If the rule atom is a constant, the procedure simply checks whether they are equal or not. If the rule atom is a variable, it checks whether the host atom is of the same type, and if so, assigns said host atom to said variable.

The case of the rule atom containing a string variable is the more complex one. Comparing strings with a string variable is similar to comparing lists with a list variable. A list variable can match a string of arbitrary length, and multiple string variables in a string are not allowed in rules. The algorithm is similar to that of list comparison with a list variable, except that we know a string variable has to be present because of the case distinction.

3.2.2 The Implementation of Matching

Let us address how the abstract algorithm in Subsection 3.2.1 is extended to work for all rules in the implementation. The abstract matching algorithm (3.1) constructs morphisms and assignments in a breadth-first manner, and returns all possible matches. The implementation (3.5) however constructs morphisms and assignments in a depth-first manner, and ends after finding the first complete match. We conjecture that, if only one match needs to be found, the abstract matching algorithm and its implementation are equivalent.

Edge enumerations are implemented using a static search plan, which is a linked list of nodes and edges of the left-hand side of a rule. The items are added in order of a depth-fist search (that ignores edge directions), starting from a root in each connected component, or from a non-root if there is no root. Search plans are essentially a concatenation of edge enumerations for different nodes that list nodes as well as edges. Note that the matching algorithm does not depend on what order edges are enumerated in. The choice of edge enumeration and how it is implemented is a matter of efficiency. For instance, the depth-first search selects outgoing edges before incoming edges, meaning the implementation needs to only check a smaller list of edges.

Algorithm 3.5: Match

```
Input: An entry of the search plan $m$, a partial premorphism $h$, a
       partial variable assignment $\alpha$, a conditional rule schema $r$,
       and a host graph $G$
Output: True if $h$ and $\alpha$ are extended to a morphism and a total
        assignment, false otherwise
1  for all items n of G that are of the same type as $m$ do
2      if $n$ is flagged as matched then continue
3      if $m$ and $n$ have mismatching marks or rootedness status, or they
         violate the dangling condition then continue
4      if $m$ matches the label of $n$ after updating variable assignments
         then
5          add $m \mapsto n$ to $h$
6          flag $n$ as matched
7          if the condition of $r$ evaluates to true or is nonexistent then
8              if Match($m$.next, $h, \alpha, r, G$) = true or is nonexistent then
9                  return true
10             else
11                 remove $m \mapsto n$ from $h$
12                 flag $n$ as unmatched
13     else remove $m \mapsto n$ from $h$
14 return false
```

Algorithm 3.5 shows the procedure Match, which is a sketch of how matching is implemented. It is a recursive procedure meant to be called on the first entry of the search plan. Match recursively calls itself on the next entry in the searchplan so it can report successful matching after all its subsequent calls report success.

It is worth noting that in line 1 we iterate over the items in host graph G that are of the same type as the current search plan entry m. If m is a root, we only iterate over the roots of G. If m is an edge, and the previous search plan entry its source, we only iterate over the outgoing edges of that source node. The iteration is analogous for incoming edges. If m is a node and the previous search plan entry is an incident edge, there is only one iteration since we know the node is either source or target of the previous edge. If m is a root node and the previous search plan entry is an incident edge, there is also only one iteration for the same reason. However, if m is a node and the previous entry is not an incident edge, i.e. the depth-first search of the search plan used m as a starting point, we have to iterate over all nodes of the host graph.

The check of whether labels match in line 4 is analogous to Update Assignment (Algorithm 3.2). In addition to comparing the labels, assignments are also updated.

3.2.3 Applying Fast Rules

There is a type of rule for which we can guarantee constant-time matching in some host graphs [8, 10]. So let us define those rules.

Definition 3.8 (Fast Rule). A rule $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ is *fast* if the following holds.

- Each node in L is in the same connected component as a root (i.e. reachable from a root via a path of edges of arbitrary direction).
- There is no string concatenation in R that is not also in L.
- There are no repeated list, string, or atom variables in either L or R.
- The application condition c contains no `edge` predicate, and no comparison $x = y$ or $x! = y$, where x and y are expressions, each of which contains a list, string, or atom variable. □

Because of the first condition, the search plan generated from L does a depth-first search starting at roots only. So when trying to match a node from such a starting point, the matching algorithm only has to iterate over roots. That makes it constantly many iterations if there are only constantly many roots in G.

String concatenation can take linear time. So in the second condition, we forbid cases where, as part of rule application, strings are concatenated.

The third condition ensures list, string, and atom variables do not have to be compared when matching labels. With big labels, this can take non-constant time. The fourth condition ensures this comparison cannot happen in the rule condition either.

Figure 3.2 shows an example of a fast rule.

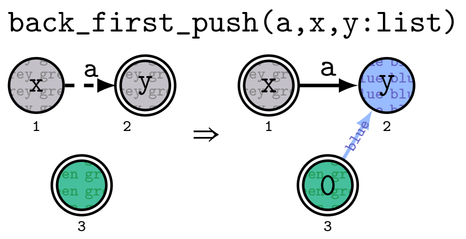


Figure 3.2: A Fast Rule

Next, we state a lemma along with its proof from [8] that shows comparing labels using the procedure Update Assignment (Algorithm 3.2) takes constant time.

Lemma 3.9 (Constant-Time Label Comparison [8])**.** *Given a fast rule $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ and a host graph G, the procedure Update Assignment compares each label in L with the corresponding label in G in constant time.*

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 constant time. There are t calls to Check, and a single assignment of the list variable to the unevaluated sublist of h. By assumption (Figure 3.1), this takes constant time. Moreover, the list and string variables do not occur anywhere else in L because r is a fast rule, so verifying consistency of the assignment also takes constant time by the complexity assumptions in Figure 3.1. Overall the running time is $O(st)$, a constant. □

The following theorem from [8] gives the runtime of the abstract matching algorithm. Host graphs need to have a bounded number of roots and a bounded node degree (number of edges adjacent to a node). The latter is because, when matching an edge starting from a node, the matching algorithm needs to iterate over the node's incident edges.

Theorem 3.10 (Fast Rule Matching [8])**.** *The abstract graph matching algorithm runs in constant time for fast rules $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ 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 and variable assignments 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 since we assume the size of programs to be fixed.

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 $r + b|A_0| + b|A_1| + \cdots + b|A_{m-1}|$. The set A_0 contains at most r morphisms, A_1 contains at most br morphisms, etc. It follows that the running time is

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

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 $bmr(r-1)$. 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 the constant

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

It remains to examine the impact of comparing labels on the runtime. 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 3.9, these executions take constant time. Therefore the total execution time is bounded from above by a constant. □

Now that we have shown fast rules are matched in constant time using the given algorithm, we extend this to applying fast rules with conditions $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$.

Corollary 3.11 (Fast Rule Application [8])**.** *Fast rules 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. Consider again a fast rule $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ with condition c and a host graph G. By Theorem 3.10, constructing a premorphism $g\colon L \hookrightarrow G$ 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 application can be executed in constant time, too.

By the definition of fast rules (Definition 3.8), 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 $x = y$ or $x != y$ where x and y 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 (Figure 3.1), 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 $g\colon L \hookrightarrow G$ can be checked by comparing the degree of each node v in $L - K$ with the degree of its image $g_V(v)$. We assume a graph representation where nodes are stored together with their indegree and outdegree. This operation then takes time of order $|VL|$, 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 time operation because there are no repeated string or list variables in the right-hand side of a fast rule. Furthermore, list concatenation can be done in constant time since we assume they are stored as doubly-linked lists. There are at most $|VK|$ relabellings, so the execution time is proportional to $|VK|$. □

3.2.4 Applying Fast Outgoing Rules

In [26], we introduce a similar result for a more general set of host graphs, but a more restricted set of rules. Neither is an improvement over the other, it is simply a trade-off.

The general idea is that, since incoming and outgoing edges of a node are stored in separate lists, we can make sure matching happens only via outgoing edges. This means host graphs only need to have bounded outdegree (number of outgoing edges of a node), but nodes in the left-hand side of rules need to be reachable from roots via a directed path. We also require there to be at most one root in each connected component. If there were two roots in a component for instance, one might not be reachable from the other via a directed path. If the search plan starts at the wrong root, it would need to traverse an incoming edge to reach the root that is not reachable via a directed path. This would render the algorithm non-constant since we do not require indegree to be bounded.

Note that in principle, this idea could be applied to incoming edges as well. However, one would need to modify the search plan generation to prioritise incoming edges over outgoing edges, instead of the other way around.

Definition 3.12 (Fast Outgoing Rule)**.** A rule $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ is *fast outgoing* if the following holds.

- Each node in L is reachable from a root via a directed path, and there is at most one node in each connected component of L.
- There is no string concatenation in R that is not also in L.
- There are no repeated list, string, or atom variables in either L or R.
- The condition c contains no `edge` predicate, and no comparison $x = y$ or $x! = y$, where x and y are expressions that both contain a list, string, or atom variable. □

Note that we need to slightly modify search plan generation for this approach to be efficient. Since the role of incoming and outgoing edges are flipped, we need to prioritise incoming edges instead of outgoing ones, i.e. when adding an edge to the enumeration, edges whose source is already covered by the enumeration are prioritised over edges whose target is covered.

Figure 3.3 shows an example of a fast outgoing rule.

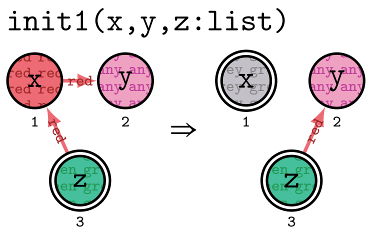


Figure 3.3: A Fast Outgoing Rule

For the proof of the theorem giving the complexity of matching fast outgoing rules, we use the assumption from Figure 3.1 that adding a node or edge mapping to a morphism takes constant time.

Theorem 3.13 (Fast Outgoing Rule Matching)**.** *Algorithm 3.5 runs in constant time for fast outgoing rules, using host graphs with a bounded outdegree that contain a bounded number of roots.*

Proof. Arguments that label comparison is constant-time are analogous to the case of fast rules since the labels and conditions fast outgoing and fast rules coincide.

For the rest this proof, we show that the number of times a partial premorphism is updated with a single node or edge is bounded. Note that L has a constant number of nodes and edges. So subsets of nodes or edges of L such as P_L and edge enumerations in L are also constant. The while loop has constantly many iterations because it terminates once every node in the bounded set P_L is tagged, which happens at least once in every iteration. The for loop has constantly many iterations because it iterates over an edge enumeration.

In line 3, a partial premorphism is added to A_0 for each match of the root p in G. Since we use host graphs with a bounded number of roots, the number of these matches is constant as well. Each of these premorphisms extends the empty premorphism by one node, so there are constantly many single-node premorphism updates.

In line 7, we extend partial premorphisms by an edge e_{p_i}, whose source is already in the extended premorphism by our assumption that edge enumerations prioritise outgoing edges and the fact that each connected component of L contains at most one root. Since G has bounded outdegree, the number of matches for e_{p_i} is bounded, and each partial premorphism can only be extended in constantly many ways. The fact that there can only be constantly many partial premorphisms can be shown by induction on $A_0, A_1, A_2, \ldots$. By the previous paragraph, A_0 has bounded size. For $k \geq 0$, if A_k has bounded size, then so does A_{k+1} since each of the partial premorphisms in A_k can only be extended in constantly many ways. So there are constantly many single-edge partial premorphism updates. Additionally, for each such update, the target of the edge may need to be added to the definition of the partial premorphism. Hence there are also constantly many single-node partial premorphism updates. □

We can again extend this result to rule application. The proof is analogous to that of Corollary 3.11, and is hence omitted.

Corollary 3.14 (Fast Outgoing Rule Application)**.** *Application of fast outgoing rules can be implemented to run in constant time, using host graphs with a bounded outdegree that contain a bounded number of roots.*

3.2.5 Applying Non-Fast Rules Efficiently

In graph programming, one often needs a kind of rule that initially introduces a root. Because of this, the left-hand side cannot contain a root, and neither the fast rule or fast outgoing rule complexity result can be used. However, there are other rules that can still match in constant time. Let us characterise one kind.

Definition 3.15 (Initialising Rule)**.** A conditional rule $\langle L \hookleftarrow K \hookrightarrow R, c\rangle$ is *initialising* if the following holds.

- L consists of a single node that is also in the interface.
- There is no string concatenation in R that is not also in L.
- There are no repeated list, string, or atom variables in either L or R.
- The condition c contains no `edge` predicate, and no comparison $x = y$ or $x != y$, where x and y are expressions that both contain a list, string, or atom variable. □

Figure 3.4 shows an example of an initialising rule.

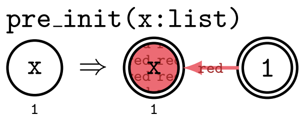


Figure 3.4: An Initialising Rule

The abstract matching algorithm (3.1) does not work for initialising roots since it can only start the matching process at roots. So instead, we argue for the complexity of the implementation (Algorithm 3.5).

Theorem 3.16 (Initialising Rule Matching)**.** *Algorithm 3.5 runs in constant time for initialising rules, using host graphs whose nodes are all a valid match for L, i.e. they are unrooted, satisfy the rule's condition, and have matching label and mark.*

Proof. The for loop in line 1 iterates over all items n of G that are nodes, which are then to be matched by node m that is the left-hand side of the rule. For the rest of this proof, we argue that the for loop terminates after its first iteration, and that the iteration takes constant time, making the execution of the whole program constant-time.

The continue in line 2 cannot be triggered since nothing has been flagged as matched yet. Neither can the continue in line 3 since m matches the label of n by assumption, they are both non-roots by assumption, and m is in the interface, the dangling condition does not apply.

Lines 2, 3, 5, and 6 take constant time because of the assumptions in Figure 3.1 that flagging operations on items, accessing marks, accessing rootedness status, and adding a mapping to a map take constant time.

Checking if the label matches in line 4 takes constant time by Lemma 3.9. The program moves on to the then branch because the label of m is assumed to match any node label in the host graph.

Evaluating the if condition in line 7 takes constant time since the rule condition is nonexistent. The same argument applies to the if in line 8.

Since the rule has no condition, and there is no additional node in its left-hand side, the algorithm then returns true, ending the execution. □

We can again extend this result to rule application. The proof is analogous to that of Corollary 3.11, and is hence omitted.

Corollary 3.17 (Initialising Rule Application). *Application of initialising rules can be implemented to run in constant time, using host graphs whose nodes are all a valid match for L, i.e. they are unrooted, satisfy the rule's condition, and have matching label and mark.*

3.2.6 Applying Edgeless Rules

One of the worst-case time complexities for matching happens when the left-hand side of a rule has no edges, and each of its nodes only has one match in the host graph. So let us explore the time complexity of that.

Theorem 3.18 (Edgeless Rule Matching). *For a rule of left-hand side L such that $E_L = \emptyset$, and host graph G, Algorithm 3.5 runs in worst-case time $O(|V_G|^{|V_L|})$ given the complexity assumptions on the compiler.*

Proof. Due to assumptions we have made on the efficiency of compiler operations, executing any line of the matching algorithm takes constant time. So the bottleneck is the number of for loop iterations in line 1. In the worst case, the for loop has to iterate over all nodes of G to find a match, since the search plan consists of nodes only. So a single call of the for loop has at most $|V_G|$ iterations. Match is called recursively for each item in the search plan, i.e. $|V_L|$ times. So the total number of for loop iterations is at most $|V_G|^{|V_L|}$. □

Once again, this result can be extended to rule application. The proof is analogous to that of Corollary 3.11, and is hence omitted.

Corollary 3.19 (Edgeless Rule Application). *Application of edgeless rules can be implemented to run in worst-case time* $\mathrm{O}(|V_G|^{|V_L|})$.

3.3 The Complexity of GP 2 Programs

Now that we have seen various conditions in which rule matching is constant-time, we look at the bigger picture and see how we can analyse the complexity of an entire GP 2 program given a set of inputs. We call a set of programs together with a set of inputs a *model*. The GP 2 model is the set of all programs with the set of all inputs.

We aim to characterise models that are subsets of GP 2. One obstacle is time-intensive rule matching. A second one is that when *critical subprograms* (bodies of loops and conditions of `if` and `try`) fail, they need to be undone, which can be handled in different ways. The current implementation of GP 2 reverses the relevant rule applications. The formal semantics uses a stack of graphs to keep track of the state of the host graph before entering a critical subprogram. Reversion can then be done by a simple pop operation. The downside to this is that the host graph has to be duplicated every time a loop or a branching statement is entered, which can add a polynomial factor to the time complexity of the overall program. It is also space-intensive, by a polynomial factor since only finitely many loop and branching statements can be nested in a command sequence. To avoid undoing altogether, one wants failed critical subprograms to be *null*, i.e. they do not change the graph state, meaning they do not need to be undone.

Definition 3.20 (Critical Subprogram). A GP 2 subprogram `C` is *critical* if it can be extended to (i.e. it is contained within) one of the following subprograms: `C!`, `if C then A else B`, or `try C then A else B`, where `A` and `B` are subprograms, and `then` and `else` branches are optional.

A subprogram is *null* on a graph state S if its execution on S results in S. □

Models that overcome the aforementioned obstacles can be implemented to match rules in constant time, which allows us to assign unit time cost to that operation. They also make undoing redundant, allowing us to charge no time cost for that.

Another obstacle is that matching a set of rules is nondeterministic, and backtracking that nondeterminism can be costly in time or space. This can be addressed at the implementation level. The current GP 2 compiler for instance picks the first match it finds and does not backtrack. A more general approach however is to show that within the scope of running a program on an input graph, every rule and rule set (nondeterministic call of a list of rules) can only have at most one match, effectively making the program deterministic.

Alternatively, the programmer can make sure that, whatever nondeterministic path rule matching has led to, the output falls within the programmer's expectations. When searching for minimum spanning trees for instance, nondeterminism can give different results, but they are all minimum spanning trees.

Definition 3.21 (Efficient Model)**.** We define an *efficient model* as $\mathcal{M} = \langle \mathcal{P}, \mathcal{I} \rangle$, where $\mathcal{P}$ is a set of GP 2 programs and $\mathcal{I}$ a set of GP 2 graphs such that the following two properties are satisfied within the scope of derivation sequences starting with any $\langle P, I \rangle \in \mathcal{M}$.

(1) *Constant Matching*: Every rule matches in constant time.

(2) *Critical Subprogram*: Every critical subprogram that fails is null on every graph state it is called on during execution. □

As a space measure, we use graph size, as defined in Definition 3.1, which counts nodes, edges, and sizes of labels. This is also how we measure input size for the sake of time and space complexity.

Definition 3.22 (Program Space Complexity)**.** Given a set of input graphs $\mathcal{I}$, we call the *(worst-case) space complexity* of a terminating program P the function

$$s_P(n) = \max\{|H| \mid G \in \mathcal{I},\ |G| = n,\ H \in [\![P]\!]G \text{ or } \langle P, [G] \rangle \to^* \langle H, S \rangle, S \in \mathcal{S}\}.$$

□

Now for efficient models, it makes sense to measure time using the number of rule applications that happen during execution, whether they are successful or not.

Definition 3.23 (Time Measure)**.** The *(worst-case) time measure* $t(P, G)$ of a terminating program P executed on a graph G is defined as the maximum number of rule applications in all semantic transition sequences starting with P and G. □

Given this time measure for the execution of programs, we can analyse the asymptotic complexity of a program with respect to a set of input graphs. Note that we do not generally give the time complexity function, but its asymptotic complexity class.

Definition 3.24 (Program Time Complexity)**.** Let $\langle \mathcal{P}, \mathcal{I} \rangle$ be an efficient model. We say the *(worst-case) time complexity* of a terminating program $P \in \mathcal{P}$ is the function $t_P(n) = \max\{t(P, G) \mid G \in \mathcal{I},\ |G| = n\}$. □

Note that sometimes, a single rule in a program takes non-constant time, but is only called once. We account for these in a program's time complexity by adding that rule's matching complexity, for instance derived from Corollary 3.19.

3.4 Related Work: Matching

In this section, we take a look at alternate approaches to graph matching. As far as we are aware no other graph transformation language uses fast rules.

3.4.1 Incremental Matching

Incremental matching is an approach to graph matching that stores partial matches throughout program execution, and updates them whenever the host graph changes. This eliminates the need to recompute previous matches, saving time.

The RETE matching algorithm [38] is an example of incremental matching. There is an implementation for graph grammars [16], and there are benchmarks [14] showing its efficiency.

Rules are compiled into a RETE-network, which, instead of storing the left-hand sides of all rules, stores graphs that occur as subgraphs in one or more left-hand sides, alongside all of its matches in the host graph. This eliminates the redundancy of storing both a left-hand side and an identical subgraph of another left-hand side, and speeds up matching.

We do not use this approach in GP 2 because the conditions under which it yields a performance gain are quite restrictive. In fact, its time complexity is the same as that of a naive solution in the worst case, especially when counting the time it takes to construct RETE-networks.

3.4.2 Dynamic Search Plans

The search plan in the current GP 2 implementation is generated statically, i.e. it only depends on the program specification, specifically the rule. A dynamic search plan also depends on the host graph and how it changes during executions. Independently of the language, an implementation of dynamic search plans needs to keep track of the host graph. This means a large structure has to be maintained, which can create a significant time overhead. Whether it actually improves or worsens time efficiency strongly depends on the problem that needs to be solved. Because it involves complex structures that do not guarantee an efficiency gain, GP 2 does not use dynamic search plans, even though they seem feasible to implement.

GrGen achieves fast matching without roots [40]. The host graph structure is analysed in advance. This is updated as rules are applied. The search plan takes this host graph structure into account, which is what makes it dynamic.

In [87], multiple search plans are generated in advance, but an optimal one is selected dynamically during program execution. The paper [45] expands on this by giving a framework that allows cost to be attributed to a variety of search plans.

4 Case Studies in Time Complexity

In this chapter, we present several case studies where we analyse the time complexity of GP 2 programs. We use the complexity models and results from Chapter 3.

The programs in this chapter are efficient because they are able to traverse a graph in linear time. Since GP 2 does not grant the programmer access to a graph's internal data structure, this is not trivial. Graph traversal is implemented using depth-first search, which takes linear time on connected graphs of bounded degree thanks to fast rule matching (Subsection 3.2.3).

An alternate approach to create efficient GP 2 programs is to write programs that reduce the size of the host graph during their execution. Having fewer nodes and edges reduces the number of potential matches for a rule, speeding up matching. However, this is not suitable for programs that output a structure, such as topological sortings or minimum spanning trees, since one wants to preserve the structure of the input graph. But it is a useful technique for programs that check whether a graph has a given property, such as recognising trees or binary directed acyclic graphs (DAGs) [19].

Note that these programs are only efficient for connected host graphs of bounded degree. Disconnected host graphs pose a problem when one component has a linear size. Matching an arbitrary node in another component can take linear time, since in the worst case, the matching algorithm has to consider every single node of the large component. Bounded degree is necessary for many rules to match in constant time. With an unbounded degree, finding an adjacent edge that matches can take linear time.

Our case studies are a program that recognises whether a graph is connected in Section 4.1, a program producing a topological sorting while recognising directed acyclic graphs in Section 4.2, and a program finding a minimum spanning tree of a weighted graph in Section 4.3. In each section, we give the program, some correctness argument, and evidence for its time complexity.

In Section 4.4, we take a look at time complexity case studies and benchmarks of other graph programs and algorithms in the literature.

4.1 Recognising Connected Graphs

This program introduces a useful technique for creating efficient GP 2 programs, namely linear-time graph traversal using depth-first search.

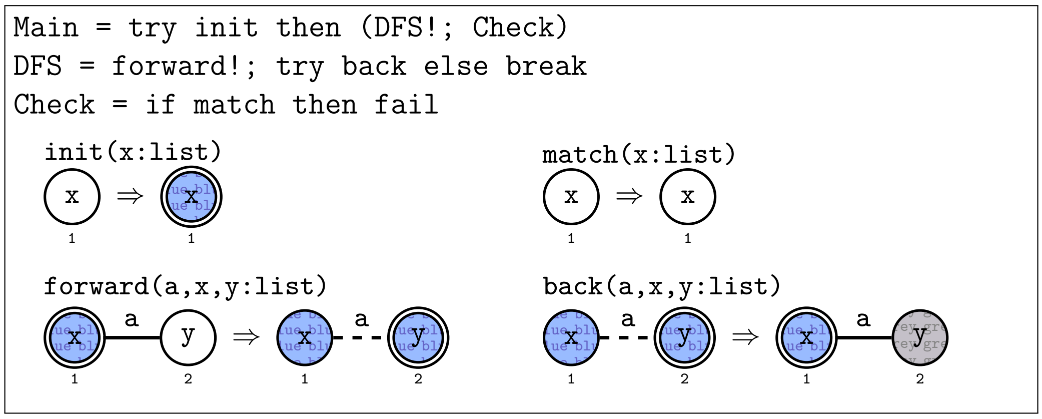
Input: An arbitrary labelled host graph such that every node and edge is unmarked, and every node is unrooted.

Output: Fail if and only if the input graph is not connected.

Throughout this section, when we refer to *specified input graph*, we mean a graph as described by the input specification.

4.1.1 The Program `is-connected`

The program `is-connected` (Figure 4.1) can detect the connectedness of a graph. It fulfils the specification in that it fails if and only if its input graph is not connected. This is achieved by conducting a depth-first search (DFS) that turns unmarked nodes into grey ones. Since the DFS cannot propagate beyond the connected component it started in, the presence of an unmarked node indicates that the host graph is not connected.



```
Main = try init then (DFS!; Check)
DFS = forward!; try back else break
Check = if match then fail
```

Figure 4.1: The program `is-connected`

4.1.2 Correctness of `is-connected`

In order to show correctness, we first exhibit a host graph property that is invariant throughout the program execution.

Lemma 4.1 (Invariant of `is-connected`). *Throughout the execution of the program `is-connected` on a specified input graph, all marked nodes in the host graph are in the same connected component.*

Proof. The rule `init` is only called at the start of the program, and turns an unmarked node into a blue one. Subsequently, only applications of `forward` can turn unmarked nodes into grey ones, and only if `init` was successfully applied. The rules `back` and `match` do not modify the number of marked nodes. Every rule preserves the structure of the host graph up to marks and rootedness. Let us inductively show that the invariant is satisfied.

If `init` is applied at the start of the program, it introduces a single blue node into a specified input graph, which does not violate the invariant. If `init` fails, `forward` is never called, and the invariant is trivially satisfied due to a lack of successful rule applications.

Assume by induction that the invariant holds on the current host graph. An application of `forward` turns node `2` in the rule's left hand side blue. However, `forward` can only be applied if said node is adjacent to an existing blue node `1`. Hence it shares a connected component with the other marked nodes. □

Lemma 4.2 (Termination of `is-connected`). *On any host graph, the program `is-connected` terminates.*

Proof. Since the loop body of `forward!` consists of a single rule, `forward` either applies and reduces the number of unmarked nodes in the host graph, or fails to find a match and terminates the loop. At some point, since the host graph is finite, there are no unmarked nodes left, and `forward` cannot match, terminating the loop.

For the termination of the loop `DFS!`, consider a couple $\#(G) = \langle a, b \rangle$ consisting of the number a of unmarked nodes of a host graph G, and the number b of dashed edges of G. By *reducing* the measure # we mean that after changing a host graph G to a graph H, we have $\#(G) > \#(H)$ with respect to the lexicographical ordering, i.e. $\langle a, b \rangle < \langle c, d \rangle$ if either $a < c$ or both $a = c$ and $b < d$.

When calling `DFS` on a host graph G, because of `try back else break`, either `back` is applied, or the loop terminates. When `back` is applied, the measure # is reduced. Indeed, if `forward` is applied at least once the number of unmarked nodes is reduced (`back` does not modify the number of unmarked nodes). And if `forward` is not applied, the number of unmarked nodes remains the same, but the number of dashed edges decreases when `back` applies. If `back` does not succeed, the loop terminates anyway.

Due to host graphs being finite, # cannot be reduced anymore at some point, which means `back` cannot be applied. Hence `break` is invoked and the loop terminates. □

Lemma 4.3 (Existence of a Marked Connected Component)**.** *In the output graph of* ***`try init then DFS!`*** *executed on a specified input graph, there is a connected component consisting only of marked nodes.*

Proof. The lemma is trivially true for an empty specified input graph. In the case of an input graph consisting of a single node, `init` marks the entire graph blue, satisfying the lemma. So we can assume the input contains at least two nodes.

Assume for the sake of a contradiction that all connected components have at least one unmarked node in the output graph. Since the input is nonempty, `init` is applied. Consider the connected component of the node `init` was applied to. Let u and v be adjacent marked and unmarked nodes of the output graph, respectively. They exist because they share a connected component that has at least one marked node (application of `init`) and at least one unmarked node (assumption). We aim to show that u and v are matched by `forward` during the execution, contradicting our assumption.

If u is grey, it must have been matched by `back`. Right before that happened, u must have been a blue root, and `forward` cannot have been applicable to it (note that there is at most one root in the host graph at any given time). However, since v is unmarked and adjacent to u, `forward` must have been applicable, which is a contradiction. So we may assume u is blue.

Since u is blue in the output graph, it must have been matched by either `init` or `forward` at some point. Either way, after that rule application, u is a blue root, and the program is executing the loop `forward!`. Since `forward` can be applied to u and v, u must have an unmarked neighbour w different from v, otherwise v would be marked blue.

For the next argument, let us take a look at the data structure the program creates. The dashed edges form a path of blue nodes, where a node at an end is rooted. This can be seen as a stack of blue nodes, where the root represents the top. Indeed, `init` initialises the stack, `forward` implements the *push* operation, and `back` the *pop* operation. Note however that `back` leaves the popped node with a grey mark, meaning it cannot be pushed again. This prevents the path from becoming a cycle, and also means that throughout the execution of `DFS!`, the number of unmarked nodes is reduced. Since the host graph has finitely many nodes, `forward` is not applicable anymore at some point, due to unmarked nodes reachable by `forward` being gone. So eventually, only `back` can be applied, popping the top of the stack until the loop terminates.

Coming back to u, v, and w, this reasoning can be applied to the loop `DFS!` from the point where u is first rooted onward, in the subgraph of the nodes reachable from w without going through u. By that reasoning, at some point, the top of the stack is popped until u is the top again. When this happens, `back` is applied and the loop `DFS!` enters its next iteration, which starts with `forward!`. This leaves us in the same situation as previously, where we must assume u has another unmarked neighbour, distinct from v and w. However at some point, there will be no unmarked neighbours to apply the previous argument to (since host graphs are finite), so v will have to matched by `forward`, contradicting our assumption. □

Theorem 4.4 (Correctness of `is-connected`). *Running `is-connected` on a specified input graph results in fail if and only if the input graph has more than one connected component.*

Proof. Termination follows from Lemma 4.2. For correctness, first assume input graph G has no connected components, i.e. G is the empty graph. Then `init` cannot be applied, and the procedure `Check` is called. The rule `match` cannot be applied either, so the program terminates without failing.

Assume G has exactly one connected component. We know by Lemma 4.2 that `DFS!` terminates. Furthermore, by Lemma 4.3, the output H of `try init then DFS!` has a connected component whose nodes are blue. Since no rule adds or deletes nodes or edges, H is isomorphic to G ignoring marks and roots. Hence the marked connected component must be the entirety of H. The procedure `Check` is called, and `match` cannot find a match in a graph containing only marked nodes. Hence `is-connected` does not fail.

Assume G has more than one connected component. The loop `DFS!` still terminates by Lemma 4.2. Furthermore, by Lemma 4.3, the output H of `try init then DFS!` has a connected component C with marked nodes. Since by Lemma 4.1, all marked nodes share the same connected component, and since H consists of more than one connected component, there is an unmarked node in $H - C$. Hence the rule `match` matches and the program fails. □

4.1.3 Time Complexity of `is-connected`

Before we examine the complexity of `is-connected`, let us show the critical subprogram property and hence that backtracking is not necessary.

Proposition 4.5 (Critical Subprogram Property for `is-connected`). *When executing `is-connected` on a specified input graph, all critical subprograms that fail are null.*

Proof. This is trivially true for critical subprograms to consist of only a single rule. The only one that does not is `DFS`, which cannot fail since loops cannot fail and the `try` statement cannot fail since `break` cannot fail. □

Theorem 4.6 (Time Complexity of `is-connected`). *On a class of bounded degree specified input graphs, the program* `is-connected` *(Figure 4.1) terminates in linear time with respect to the size of its input.*

Proof. To prove this theorem, we show that for each rule, the number of applications during an execution multiplied with its matching complexity is linear.

The rule `match` has a linear matching complexity by Corollary 3.19. It is also only called once, so we can add its time complexity to the program's. It remains to show that the rest of the program has linear time complexity.

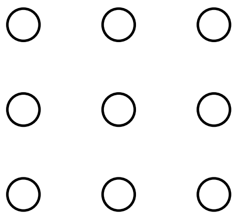
In order for Corollary 3.11 to be applicable, there can only be a constant number of roots in the host graph. This is indeed the case. The only rule that does not preserve the number of roots is `init`, which is only called once. So the host graph can have at most one root at any time.

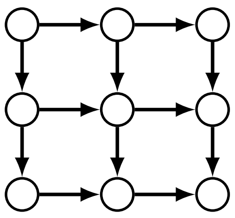
The rules `forward` and `back` are fast rules, and hence apply in constant time by Corollary 3.11. Additionally, `init` matches in constant time by Corollary 3.17, and is only applied once.

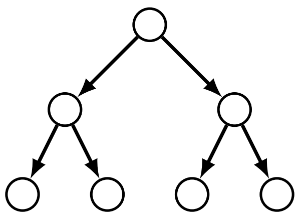
Let us show that `back` is applied a linear number of times. The number of applications is the number of successful applications plus the number of unsuccessful applications. Since the loop `DFS!` terminates after `back` is successfully matched (see Lemma 4.2), we know that `back` succeeds at each call except for the final one, i.e. it has a constant number of unsuccessful applications. So it is enough to show the linearity of the number of successful applications of the rule. It is easy to see that `back` increases the number of grey nodes, while all other rules preserve it. Since `back` cannot match a grey node, it can only be applied a linear number of times.

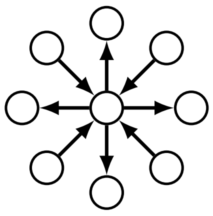
Let us show that `forward` is applied a linear number of times. By the previous paragraph, `back` is applied a linear number of times, meaning the loop `DFS!` has a linear number of iterations. During each of these iterations, `forward` fails exactly once (termination of `forward!` by Lemma 4.2), so there is a linear amount of unsuccessful applications. The rule `forward` decreases the number of unmarked nodes, while the other rules called in `DFS!` preserve it. Since it needs an unmarked node to match successfully, it can only do so a linear number of times. Hence `forward` is only called a linear number of times. □

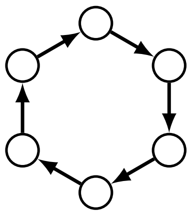
Finally, we have collected empirical timing results, supporting our claim that the program runs in linear time on graph classes of bounded degree, but

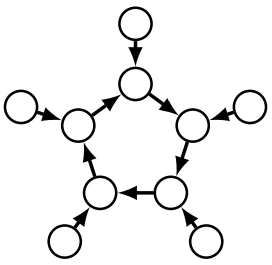

(a) Discrete graph

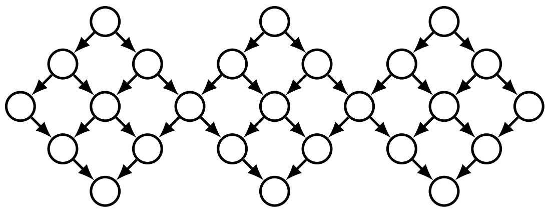

(b) Grid graph


(c) Binary tree


(d) Star graph


(e) Cycle graph


(f) Sun graph


(g) Grid chain

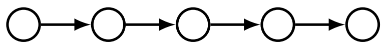

(h) Linked list

Figure 4.2: Input graph classes for time measurements

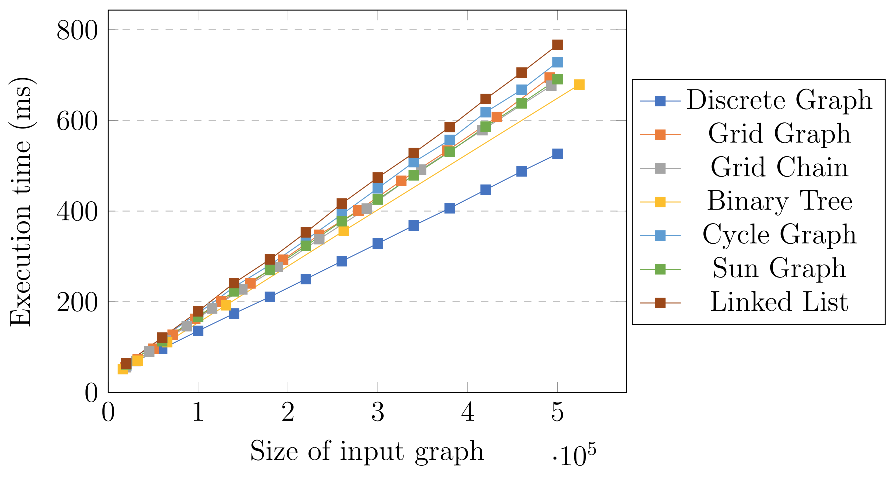


Figure 4.3: Measured performance of `is-connected` on graphs of bounded degree

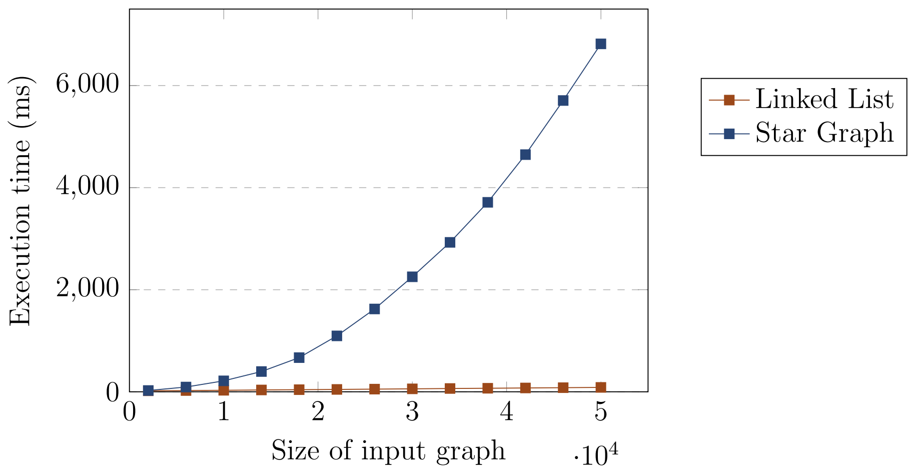


Figure 4.4: Measured performance of `is-connected` on graphs of unbounded degree

not necessarily on those that do not have bounded degree.

The input graphs for our measurements are depicted in Figure 4.2. Only star graphs have unbounded degree, as they grow by increasing the number of nodes adjacent to the central one. Discrete, grid, binary tree, cycle, and linked list graphs grow in the intuitive way. Sun graphs grow by adding nodes to the central cycle and completing the pattern. Grid chains grow by concatenating additional grids and expanding the size of each grid.

The timings can be found in Figures 4.3 and 4.4. These time measurements and figures are made by co-author Graham Campbell as part of [19].

Note that program execution never takes constant time because the measured time includes input graph parsing, which takes linear time.

4.2 Topological Sorting and DAG Recognition

In this section, we give a program that recognises directed acyclic graphs (DAGs) and produces a topological sorting.

Input: An arbitrary labelled connected host graph G such that every node is marked grey, every edge is unmarked, and every node is unrooted.

Output: Fail if the input is not a DAG, and G equipped with a topological sorting otherwise.

Note that the input graph having grey nodes is a cosmetic decision. It allows us to use the `any` mark to match an unvisited node. With unmarked input nodes, rules with the any mark can be replaced a rule set call that covers the absence of a mark in addition to the `any` mark. Throughout this section, when we refer to *specified input graph*, we mean a graph as described by the input specification.

4.2.1 The Program `top-sort`

The GP 2 program `top-sort` (Figures 4.5, 4.6, 4.7) presented in this section has two purposes: recognising whether its connected specified input graph is a DAG (directed acyclic graph) and if it is, producing a topological sorting of said graph. An example execution can be found in Figure 4.8.

The class of *DAGs (directed acyclic graphs)* consists of all graphs that do not contain a directed cycle as a subgraph. A *topological sorting* of a DAG G is a total order (an antisymmetric, transitive, reflexive, and strongly connected binary relation) $\leq$ on the set of nodes of G, such that for each edge of source u and target v, $u \leq v$ (*topological property*). Topological sortings cannot exist for graphs containing directed cycles, since there is no way to define a total order on the nodes of a cycle such that the topological property is satisfied.

Algorithm 4.6 and Procedure 4.7 form the pseudocode for an imperative implementation of topological sorting [23]. The program uses depth-first search to traverse the nodes. Unvisited nodes are marked red, nodes currently being used in the DFS grey, and finished nodes blue. When a node is finished,

```
Main = try init then
       (StackNodes!; unroot; LoopNodes!; if flag then fail)
StackNodes = {forward1, forward2}!; try back else break
```

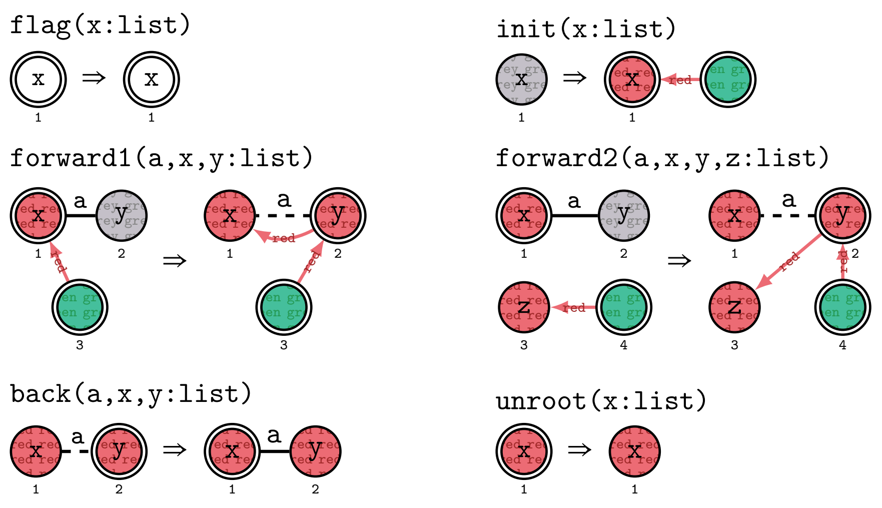


Figure 4.5: The program `top-sort` and procedure `StackNodes`

```
LoopNodes =
  if flag then break;
  try skip1 else (
    try skip2 else (
      try init1 then (
        SortNodes!
      ) else (
        try init2 then (
          SortNodes!
        ) else (
          break
        )
      )
    )
  )
```

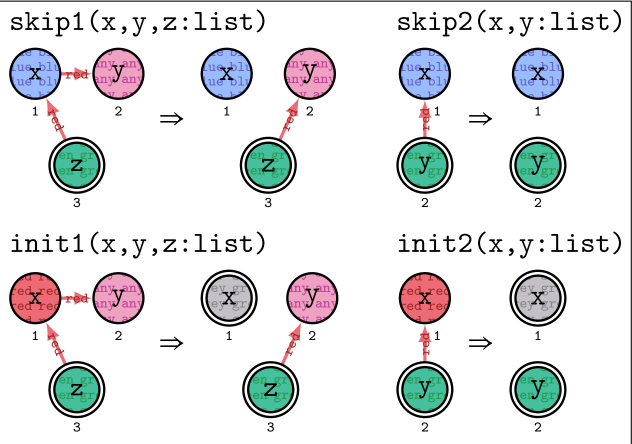


Figure 4.6: The procedure `LoopNodes`

```
SortNodes =
  forward!;
  if {loop, two_cycle, back_edge} then (
    set_flag;
    break
  );
  try back_push else (
    try back_first_push else (
      try grey_push else (
        grey_first_push
      );
      break
    )
  )
```

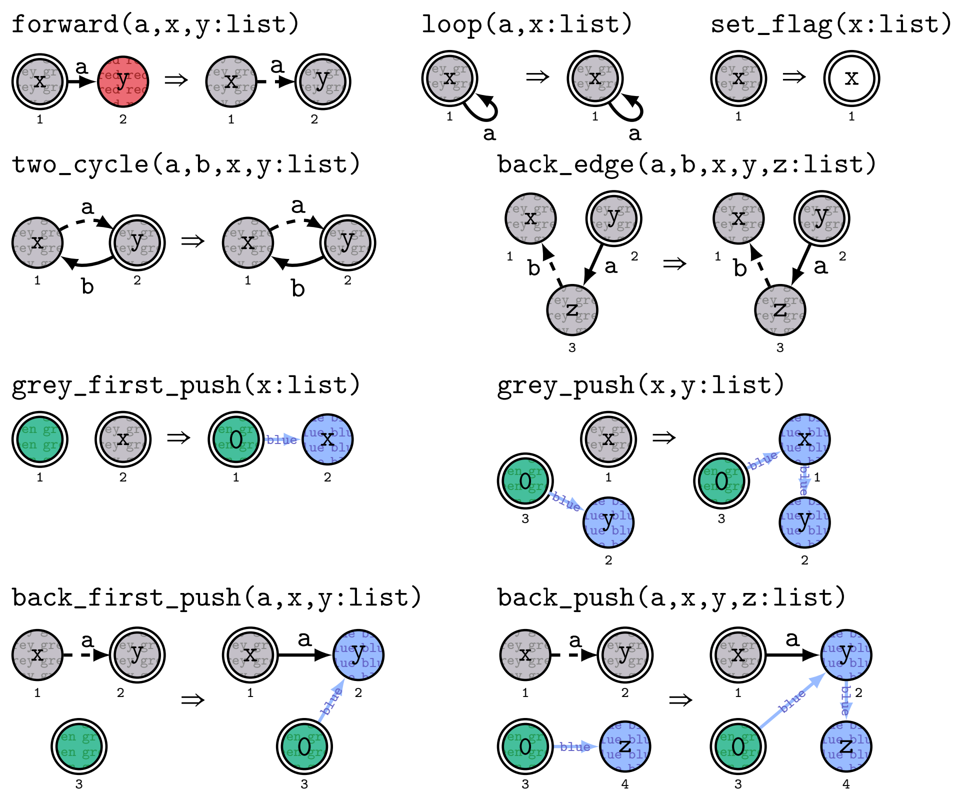


Figure 4.7: The procedure `SortNodes`

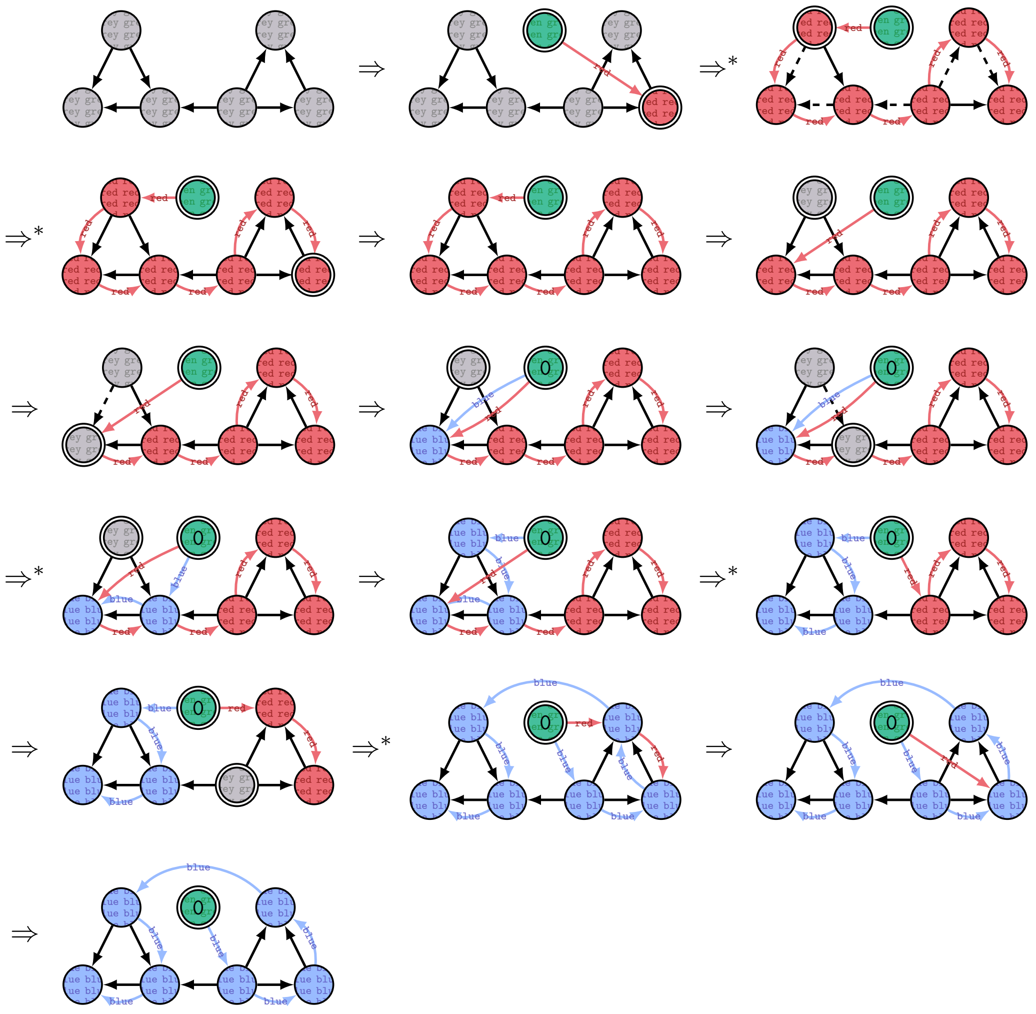


Figure 4.8: Example derivation sequence of `top-sort`

it is inserted onto the front of a linked list. When the program terminates, that linked list contains the nodes in a topological order.

The GP 2 program `top-sort` works in a similar fashion. One difference is that, since (by design) we do not have access to the graph data structure in GP 2, we cannot directly loop over all nodes in the host graph. So as a first step, we use DFS to construct a stack of red edges containing all nodes, enabling us to iterate through them. Another difference is that the GP 2 program has the secondary purpose of recognising DAGs. A topological sorting is defined and only makes sense on DAGs. So instead of leaving the behaviour of `top-sort` undefined on non-DAGs, we make the program fail on them. This is achieved by setting a flag if structures that signify the existence of cycles are found during the topological sorting.

Algorithm 4.6: Topological-Sort(G)

```
1 $L \leftarrow$ empty linked list
2 for each node $u$ of $G$ do $u$.colour $\leftarrow$ red
3 for each node $u$ of $G$ do
4     if $u$.colour = red then DFS-Visit($G, u, L$)
5 return $L$
```

Procedure 4.7: DFS-Visit(G,u,L)

```
1 $u$.colour $\leftarrow$ grey
2 for each node $v$ adjacent to $u$ do
3     if $v$.colour = red then DFS-Visit($G, v, L$)
4 $u$.colour $\leftarrow$ blue
5 insert $u$ onto the front of $L$
```

So the GP 2 program `top-sort` uses depth-first search to traverse the host graph in linear time while testing whether it is a DAG and constructing a path of blue edges that define a topological sorting. Moreover, it terminates in linear time on inputs of bounded node degree.

Note that while the program is only correct on connected graphs, it can be modified to work on arbitrary graphs too, but at a cost. Not only does the program become more complex, but the linear time complexity result fails also, due to there being no way to iterate all the connected components in linear time. This is because the matching algorithm finding an unvisited node in a new connected component needs to check the nodes of the already visited components first in the worst case. If there is an unbounded number of connected components, the repetition of this process would take quadratic time.

4.2.2 Correctness of `top-sort`

The first step is to show termination of `top-sort`.

Lemma 4.7 (Termination of `top-sort`). *On any host graph, the program `top-sort` terminates.*

Proof. Consider the loop {`forward1, forward2`}`!` called in the procedure `StackNodes` (Figure 4.5). In each iteration, either `forward1` is applied, `forward2` is applied, or both fail and the loop terminates. Whenever one of these rules is applied, the number of grey nodes in the host graph is reduced. Due to host graphs being finite, there are no grey nodes left eventually, and neither rule can match, terminating the loop.

Next, consider the loop `StackNodes!` (Figure 4.5), a measure # consisting of the number of grey nodes of a host graph paired with the number of dashed edges, and a lexicographical ordering on said pairs. In each iteration, either `back` is applied or the loop terminates due to `break`. If `back` is applied, either the number of grey nodes remains the same and the number of dashed edges is reduced (i.e. neither `forward1` nor `forward2` are applied), or the number of grey nodes is reduced (i.e. `forward1` or `forward2` have been applied at least once). In either case, # is reduced. Since host graphs are finite, # cannot be reduced anymore at some point, hence `back` is not applicable. Then `break` is invoked and the loop terminates.

Now consider the loop `forward!` in the procedure `SortNodes` (Figure 4.7). In each iteration, either `forward` is applied, or the loop terminates. Applying `forward` reduces the number of red nodes in the host graphs. Since there are only finitely many, there will be no red nodes left for `forward` to match eventually. Hence the loop has to terminate.

Next, consider the loop `SortNodes!` (Figure 4.7). In each iteration, either the rule `back_push` applies, the rule `back_first_push` applies, or `break` is invoked. We claim that each iteration either lexicographically reduces the number of red nodes in the host graph paired with the number of grey nodes (let us call this measure #), or terminates the loop. The rules `set_flag`, `grey_push`, and `grey_first_push` only get called in an iteration that terminates the loop, so we do not need to consider them for the purpose of reducing #. Similarly, as rules called in the condition of an `if` statement do not modify the host graph, we do not need to consider `loop`, `two_cycle`, and `back_edge` for reduction purposes either. So consider an iteration where either either `back_push` or `back_first_push` applies. If `forward` is applied at least once this iteration, the number of red nodes is reduced, which reduces #. If `forward` is not applied at all, the number of red nodes remains the same while either `back_push` or `back_first_push` reduces the number of grey nodes, reducing #. So eventually, # cannot be reduced any further in the finite host graph, meaning neither `back_push` nor `back_first_push` can be applied, causing the loop to terminate.

Finally, consider the loop `LoopNodes!` (Figure 4.6). In each iteration, either `break` is invoked, or one of the rules `skip1`, `skip2`, `init1`, and `init2` is applied. Each of these rules reduces the number of red edges in the host graph. So eventually, since host graphs are finite, there are no red edges that can be matched by these rules anymore, so all of them will fail, meaning `break` is invoked due to the structure of the nested `try` statements, and the loop terminates. □

The command `init; StackNodes!` creates a stack of nodes in order to navigate between strongly connected components while doing a depth-first search that travels in the direction of the edges. It is equivalent to `try init then StackNodes!` if `init` is applied successfully. We shall define stacks via their implementation.

A *stack* is a finite set of red nodes connected by red edges such that the red edges form a path that does not self-intersect. The node in the path that has no incoming red edge from another path node is called the *top* of the stack. Additionally, there is an unlabelled green root node called the *pointer* with only one adjacent edge, namely an outgoing red edge whose target is the top of the stack.

Note that such a stack can also be defined with blue nodes and edges instead of red ones, in which case we shall call it a *blue stack*. In fact, during the execution of `LoopNodes!`, a red and a blue stack coexist using the same green root.

Lemma 4.8 (Correctness of `init; StackNodes!`)**.** *The command sequence `init; StackNodes!` is totally correct with respect to the specification:*

Input: *A specified input graph G.*
Output: *G where all its nodes are in a red stack.*

Proof. Termination follows from Lemma 4.7.

The proof that all nodes of G are marked red is analogous to that of Lemma 4.3.

To show that a correctly encoded stack is formed, let us proceed by induction. The rule `init` creates a valid stack containing a single node. Now assume a valid stack is encoded in the host graph. Let us argue that after applying `StackNodes`, that is still the case. Whenever a red edge is created, the target is the top (since it is adjacent to the green root), and the source is a grey node (and hence not part of the path already, since a specified input graph has grey nodes), extending the non-self-intersecting path. The green root now points toward the newly added node, making it the new top. □

Let us now define how to represent a topological sorting in the context of GP 2.

Definition 4.9 (Topological sorting as a graph structure)**.** Consider a graph G and the set of its blue nodes B. Define a binary relation on B by $u \leq v$ if there is a blue edge from u to v, or if $u = v$. G contains a *topological sorting* if the transitive closure of $\leq$ is a topological sorting of the subgraph of G induced by the blue nodes B and the unmarked edges. $\square$

With such a structure, one can test whether two nodes are in a topological order by checking whether there is a path of blue edges connecting them.

Lemma 4.10 (Correctness of `LoopNodes!`). *On a graph whose nodes are all in a red stack, and whose subgraph induced by its unmarked edges is a DAG,* `LoopNodes!` *outputs a graph G that contains an unmarked root or a topological sorting.*

Proof. Termination follows from Lemma 4.7.

Let us assume G does not contain an unmarked root, and show that it does contain a topological sorting. In fact, since no rules called in `LoopNodes!` can mark or unroot an unmarked node, we can assume that no unmarked root is introduced at any point, i.e. `set_flag` is not applied.

Consider the binary relation $\leq$ on the set of blue nodes defined by blue edges as in Definition 4.9, and let us show it is a topological sorting. It is transitive since it is defined as a transitive closure.

Antisymmetry follows from the fact that $\leq$ is reflexive and the fact the subgraph H of G induced by blue edges does not contain directed cycles. Indeed, H behaves like a stack of blue nodes and edges with a green root pointing towards the top with a blue edge. When the green root is unlabelled, the stack is initialised as a single blue node, and the green root is labelled `0`. Once that label has been established, non-blue nodes are pushed. At no point does the program pop a blue node, or change the mark of a blue node. Hence no blue cycle can be introduced.

Strong Connectedness follows from the fact that every node is eventually marked blue, i.e. pushed. By using arguments analogous to those in Lemma 4.3, we can conclude that after `SortNodes!` is applied, the grey root and all nodes reachable from it are marked blue (which we can only conclude because the `if` statement does not change the host graph since we assume `set-flag` is not applied). This difference is because the steps of the depth-first search are sensitive to edge direction. In order to sort through remaining nodes, `LoopNodes!` skips over blue, i.e. already sorted nodes in the red stack with `skip1` and `skip2`, until it reaches a red, i.e. unsorted node, which is then initialised as grey root with `init1` or `init2`. Then `StackNodes!` is applied on that grey root. Since all nodes of the input graph are in the red stack by Lemma 4.8, all nodes are eventually marked blue.

It remains to show that $\leq$ satisfies the topological property, namely that for each unmarked edge from u to v in the input, $u \leq v$, i.e. there is a blue path from u to v in the output graph. Since the blue edges form a stack of all nodes, it is enough to show that u is pushed after v. Consider the iteration of `SortNodes!` that pushes u onto the blue stack with one of the push rules (`grey_first_push`, `grey_push`, `back_first_push`, and `back_push`). The node u has no outgoing unmarked edge with a red node as the source because then `forward!` would have had at least one more iteration, and this would not be the iteration of `SortNodes!` that pushes u. So v is not red. It cannot be grey either because then there would be a path of dashed edges from v to u (since the grey nodes are in a path of dashed edges, and the root, u, is the final node of the path), which would mean there was a cycle of unmarked edges in the input. So v must be blue, i.e. it is pushed before u. □

Now let us show the total correctness of `top-sort`. Note that we include the empty graph in the definition of DAGs. If one wishes to exclude it from the class of DAGs, it suffices to add the `else fail` to the `try` statement in `Main`, since `init` fails on the empty graph.

Theorem 4.11 (Correctness of `top-sort`). *The program `top-sort` (Figures 4.5, 4.6, 4.7) is totally correct with respect to the specification:*

Input: *A specified input graph.*
Output: *Fail if the input is not a DAG, and G equipped with a topological sorting otherwise.*

Proof. If G is the empty graph, `init` is not successfully applied, and the output is G, which defines a valid topological sorting of the empty DAG.

Termination follows from Lemma 4.7.

If G is a DAG, and no unmarked root is introduced in `LoopNodes!`, it follows from the same lemmata and the fact that no rule of `top-sort` changes the structure of the underlying graph of unmarked edges, that the output is G containing a topological sorting. So we need to show that, if G is a DAG, `LoopNodes!` does not introduce an unmarked root. Conversely, if G is not a DAG, we need to show that an unmarked root is introduced in `LoopNodes` because matching `flag` is the only way for `Main` to fail (`unroot` always matches since `SteckNodes` leaves a red root in the host graph).

The only rule that introduces an unmarked root is `set_flag`, which is only called during the `if` statement in `SortNodes`, if the condition is satisfied. So it is enough to show that G is a DAG if and only if neither `loop`, `two_cycle`, nor `back_edge` matches.

As argued in the proof of Lemma 4.10, every non-pointer node is pushed onto the blue stack with one of the push rules. So the `if` statement called right before the push rules are invoked for each non-pointer node while it is a grey root.

If G is a DAG, `loop` and `two_cycle` cannot match since they need a 1-cycle or 2-cycle respectively to be present in G. The rule `back_edge` cannot match either. It contains a path from node 2 to node 1. As the target of a dashed edge, node 1 is in the stack of dashed edges, so there is a path to node 1 to node 2, the top of the stack. This means there is a cycle.

Conversely, assume that G is not a DAG. If it contains a 1- or 2-cycle, either `loop` or `two_cycle` matches. So assume G contains a cycle of length at least 3. Consider the first time a node of that cycle becomes a grey root due to `forward`. Eventually, `forward` is applied to make the next node in the cycle the grey root. We can repeat this argument until the last node in the cycle is the grey root (in the cycle, all edges but one are dashed, all nodes are grey, and the node with an outgoing unmarked edge is rooted). Then `back_edge` can match. □

4.2.3 Time Complexity of `top-sort`

Before we examine the complexity of `top-sort`, let us show the critical subprogram property and hence that backtracking is not necessary.

Proposition 4.12 (Critical Subprogram Property for `top-sort`). *When executing `is-connected` on a specified input graph, all critical subprograms that fail are null.*

Proof. This is trivially true for critical subprograms consisting of only a single rule, or a single rule set.

The loop body `StackNodes` cannot fail because a loop cannot fail, and the `try` statement cannot fail since `break` cannot fail.

In the loop body `LoopNodes`, nothing can fail since it is a combination of branching statements, a loop, and `break` statements.

Using the same reasoning on the loop body `SortNodes`, we can conclude that `set_flag` and `grey_first_push` are the only rules that can cause failure. The rule `set_flag` cannot fail because it is immediately preceded by an application of either `forward`, `init1`, or `init2`, which all leave behind a grey root. Similarly, these preceding applications guarantee the presence of a grey root for `grey_first_push`. The green root is there because it is created by `init`. If it had a label, it would also have a blue edge pointing to a blue node, which is apparent from all the rules that modify the green root's label and its outgoing blue edge. Therefore, `grey_push` in the `try` condition would have matched and `grey_first_push` would never have been called in the first place. So we can assume the green root has no label, and `grey_first_push` cannot fail. □

Theorem 4.13 (Complexity of `top-sort`). *On a class of bounded degree specified input graphs, the program `top-sort` (Figures 4.5, 4.6, 4.7) terminates in linear time with respect to the size of its input.*

Proof. To prove this theorem, let us now show that for each rule, the number of applications during an execution multiplied with its matching complexity is linear.

First consider `init`. It is applied once, and every node of the input graph is a valid match, so its matching complexity is constant by Corollary 3.17.

Let us now argue that there is a constant number of roots at any given time, so that we can apply Corollary 3.11. The rule `init` introduces a green root. In all other rules, the rootedness and mark no green nodes get modified, and no green nodes get introduced. The rule `init` also introduces a red root. In `StackNodes!`, no rules modify the number of red roots. Then with `unroot`, the one red root is removed. The rules `init1` and `init2` introduce a grey root. Whenever one of them is applied, `SortNodes!` is called. Let us show that `SortNodes!` removes the grey root. This loop can only terminate when `break` is invoked (termination itself is shown in Lemma 4.8), or when `grey_first_push` does not find a match. In the latter case, either the grey root has already been removed (which is what we want), or the green root has a label. If the green root as a label, `grey_push` or `back_push` would have been applied, and `grey_first_push` never called. Now consider the case where `break` is invoked. This must be preceded by a successful application of `set_flag`, `grey_push`, or `grey_first_push`. In the latter two cases, the grey root is unrooted. In the former case, the grey root is unmarked, and various `break` and `fail` statements are invoked and the program terminates. In either case, the number of roots remains constant.

Each rule except for `init`, is a fast rule. So by Corollary 3.11, they have constant matching complexity. So it remains to show that the number of applications is at most linear.

The rules `unroot` and `flag` are applied at most once.

Note that {`forward1, forward2`} is only successfully applied a linear number of times since it reduces the number of grey nodes, of which there can be only linearly many, and no other rule in `StackNodes!` introduces grey nodes. So `back` can only be successfully applied a linear number of times. And the number of times `back` is unsuccessfully applied is once, since in that case `break` is invoked. Hence`back` is applied a linear number of times. This also means that the number of iterations of `StackNodes!` is linear. So the number of times `forward1` and `forward2` are unsuccessfully applied can only be linear as well. Hence `forward1` and `forward2` are applied a linear number of times.

Therefore there can only be linearly many red edges. In each successful iteration of `LoopNodes!`, one of `skip1`, `skip2`, `init1`, `init2` has to be applied, reducing the number of red edges. So there can only be linearly many iterations of `LoopNodes`, making the number of applications of `skip1`, `skip2`, `init1`, and `init2` linear.

The rule `forward` can only be successfully applied a linear number of times since it reduces the number of red nodes, and no other rule in the loop `LoopNodes!` modifies that number. So there can only be linearly many dashed edges, meaning that combined, `back_first_push` and `back_push` can only be applied a linear number of times. Hence there can only be a linear number of applications of `SortNodes`. So each rule `r` called only once in `SortNodes` has only a linear number of total applications. That also means that the number of times `forward` is unsuccessfully applied is also linear, so `forward` is applied a linear number of times. □

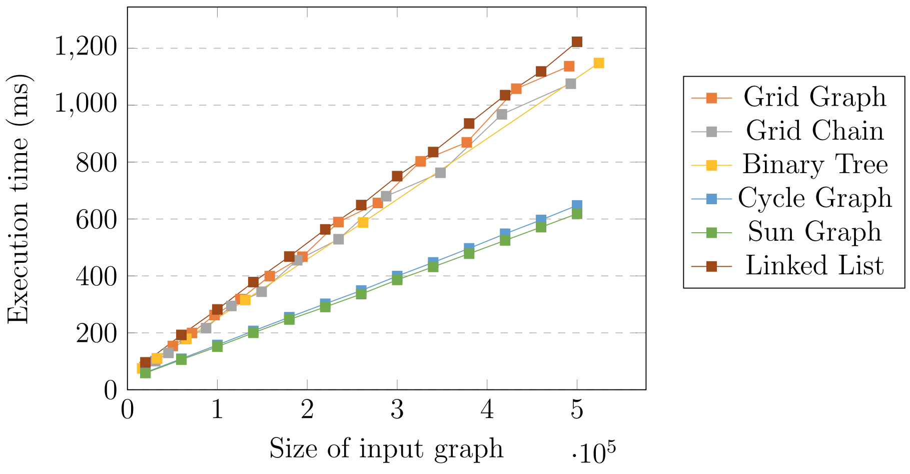


Figure 4.9: Measured performance of `top-sort`

We have collected empirical timing results, supporting our claim that the program runs in linear time on classes of connected graphs of bounded degree (Figure 4.9). The input graph classes are the same as the ones in Subsection 4.1.3. These time measurements and figures are made by co-author Graham Campbell as part of [19].

4.3 Minimum Spanning Trees

The program in this section finds a minimum spanning tree, that is a tree of minimum edge weight that contains all nodes of a graph. The correctness of this program is based on its similarity to the algorithm it implements, as well as testing on various small examples.

Input: A connected host graph G such that every node is marked grey, every edge is unmarked, and every node is unrooted, nodes have arbitrary labels, and edges have positive integer labels.

Output: Graph G where a minimum spanning tree is highlighted with blue edges.

4.3.1 Boruvka's Algorithm

Prim's, Kruskal's, and Boruvka's algorithms for computing MSTs can all be implemented to run in $\mathrm{O}(m \log n)$ time, where m is the number of edges, and n the number of nodes. However Prim's algorithm needs binary heaps to achieve it, and Kruskal's algorithm the union find data structure [12]. The advantage of Boruvka's algorithm is that it only needs very simple data structures that are clean to implement in GP 2 to reach that time complexity bound [76]. GP 2 has no predefined data structures except for the host graph that it transforms. Any additional data structures need to be encoded in the host graph itself, which can make a program tricky to read. Hence we choose to implement Boruvka's algorithm in GP 2.

Algorithm 4.8 shows pseudocode for Boruvka's algorithm. Although it cannot translate directly into GP 2, it is a suitable starting point for the development of a GP 2 program. A C implementation of this algorithm can be found in [74, 75].

Algorithm 4.8: Boruvka's MST algorithm on an input graph G

```
1 Preprocess: initialise the spanning forest $F$ to be the nodes of $G$;
2 while $F$ consists of more than one tree do
3     for each tree $T$ in $F$ do
4         FindEdge: select a minimum weight edge between $T$ and
            $G - T$, prioritising already selected edges if they are
            minimum
5     GrowForest: add the selected edges to $F$
```

The idea of Boruvka's algorithm is to initialise a forest as the nodes of the input graph without any edges, and to grow that forest by adding minimum-weight edges from between its connected components until it becomes a minimum spanning tree of the input graph.

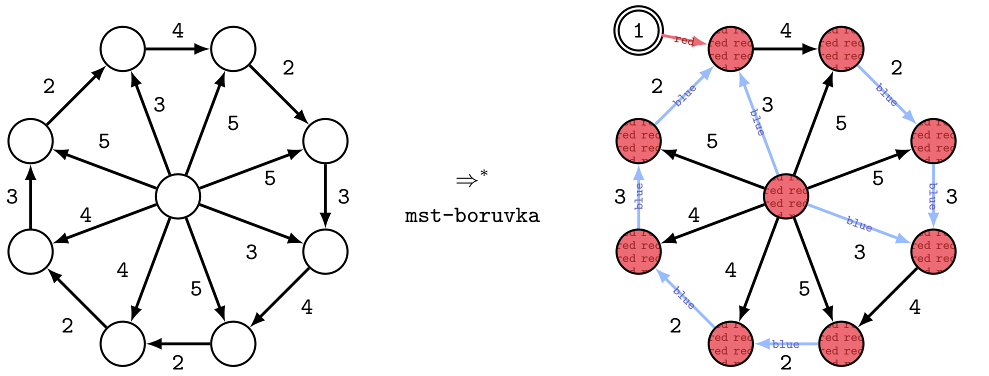


Figure 4.10: Example input and output of `mst-boruvka`

As illustrated in Figure 4.10, the input of `mst-boruvka` is a connected graph with unmarked nodes and edges. Nodes are unlabelled, and edges have integer labels. In the output, the subgraph induced by the blue edges are a minimum spanning tree of the input. The additional root with label 1 is an auxiliary construct used in the execution of the program (which could be removed in constant time).

4.3.2 Example Execution

Throughout the execution of `mst-boruvka`, the graph induced by the blue edges is a subgraph of the minimum spanning tree highlighted in the output. We shall call this forest F, and its connected components its *trees*. Let us explore how `mst-boruvka` executes using the example in Figure 4.11, and compare it to the pseudocode in Algorithm 4.8. The `Main` procedure of `mst-boruvka` is depicted in Figure 4.12.

The procedure `Preprocess` initialises the forest F to be just the nodes of the input(see line 1 of the pseudocode). It also sets up a linked list of red edges and red nodes that helps the program loop over the trees of F efficiently. Each tree of F is represented by exactly one of its nodes being an entry in the linked list. Additionally, there is a pointer in the form of an unmarked root node with an outgoing red edge towards the "current" node in the linked list. The pointer also stores the number of trees the forest has in order to efficiently check whether only one tree is left, terminating the main loop (see line 2 of the pseudocode).

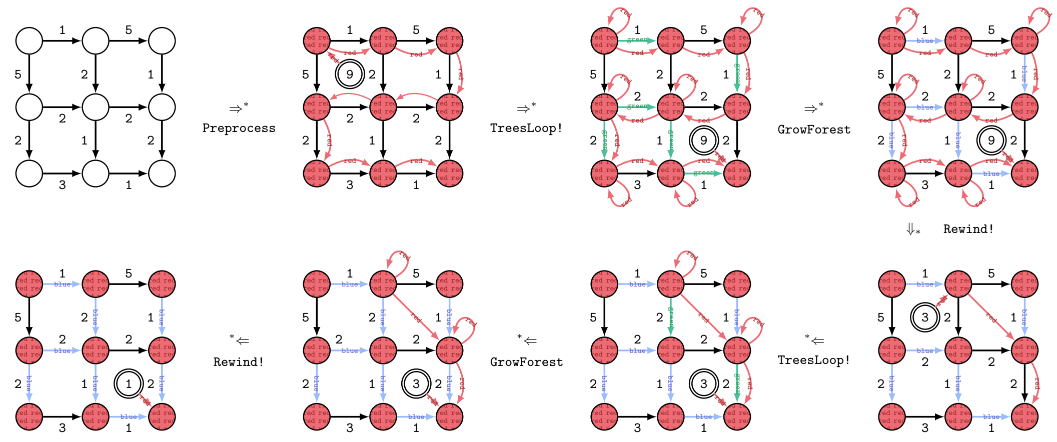


Figure 4.11: Example derivation sequence of `mst-boruvka`

The loop `TreesLoop!` moves the pointer through the nodes of the linked list, effectively looping over the trees of F (see line 3 of the pseudocode). On each tree T, the procedure `FindEdge` is called, which selects a minimum weight edge between T and its complement in the host graph by marking it green (see line 4 of the pseudocode). If there is already an adjacent green edge with minimum weight, no new edge is selected since that could introduce a cycle into F. To ensure that only one node of each tree is part of the list, the current tree gets marked for deletion from the list using a red loop under certain conditions. Subsection 4.3.7 elaborates on this.

The procedure `GrowForest` adds the selected edges to F by turning green edges into blue ones (see line 5 of the pseudocode).

The loop `Rewind!` serves to maintain the linked list. It moves the pointer back to the beginning of the list. On the way, it removes nodes that have been marked for deletion with a red loop. It also decrements the pointer's label each time it encounters such a node since that node's tree has been merged with another tree.

4.3.3 The Program `mst-boruvka`

The program `mst-boruvka` is depicted in Figure 4.12. Most of it is explained by the example execution in Subsection 4.3.2. Let us now examine the loop `TreesLoop!`.

The purpose of the loop `TreesLoop!` is to find a minimum weight edge from each tree to its complement and mark it green. It initialises by rooting the node the pointer points to. Then that node's tree is marked blue with the procedure `ColourBlue` so it can easily be distinguished from the rest

```
Main = Preprocess; Loop!
Loop = if one_tree then break else Body
Body = TreesLoop!; GrowForest; Rewind!
TreesLoop = root_current; TraverseTree; MarkForDeletion;
            CleanUp; try next_tree else break
TraverseTree = ColourBlue; FindEdge
CleanUp = ColourRed; unroot_red!
```

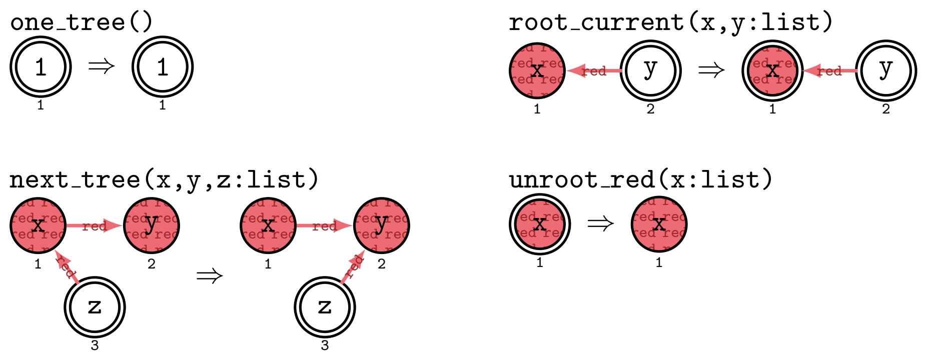


Figure 4.12: The program `mst-boruvka`

of the graph. `FindEdge` then finds the minimum edge from the tree to its complement. The procedure `MarkForDeletion` marks the tree for deletion if it will be merged with another one. The procedure `ColourRed` makes the nodes of the tree be red again. The command `unroot_red!` unroots any red roots. The rule `next_tree` then moves the pointer to the next entry in the linked list.

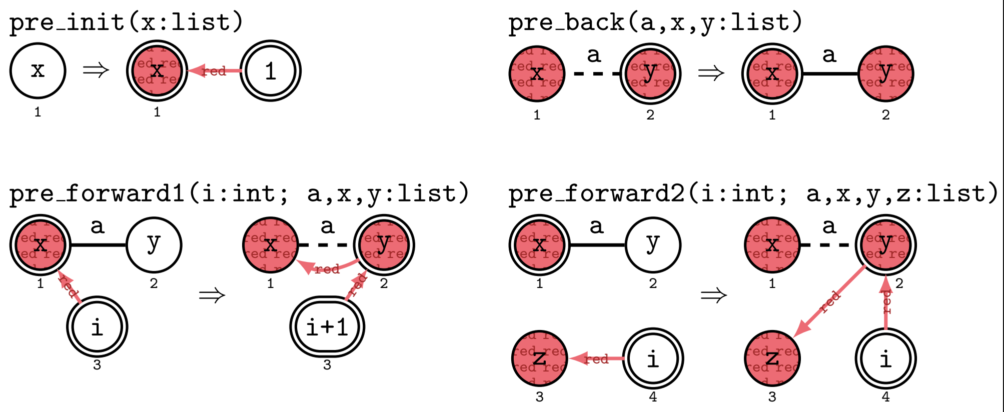
4.3.4 The Procedure `Preprocess`

The procedure `Preprocess` depicted in Figure 4.13 uses depth-first search (DFS) to construct the linked list and the pointer. An example of its input and output can be seen in Figure 4.11.

The rule `pre_init` initialises some node of the input to be the starting point of the DFS, and constructs the pointer. Since initially each node is its own tree, the pointer's label will count the number of nodes encountered during the DFS. Red nodes are considered to be discovered by the DFS, and unmarked nodes undiscovered.

The rules `pre_forward1` and `pre_forward2` are called nondeterministically. They both move the red root to an adjacent unmarked node. The rules contain *bidirectional edges* (without arrowheads) that can be matched

```
Main = Preprocess; Loop!
Loop = if one_tree then break else Body
Body = TreesLoop!; GrowForest; Rewind!
TreesLoop = root_current; TraverseTree; MarkForDeletion;
            CleanUp; try next_tree else break
TraverseTree = ColourBlue; FindEdge
CleanUp = ColourRed; unroot_red!
```



pre_init(x:list)
pre_back(a,x,y:list)
pre_forward1(i:int; a,x,y:list)
pre_forward2(i:int; a,x,y,z:list)

Figure 4.13: The procedure Preprocess

in either orientation. The rule is a shorthand for a non-deterministic call of copies of the same rule whose bidirectional edges have been replaced with directed edges in all possible combinations of orientation. The dashed edge serves as a way to keep track of the path the DFS has taken, which is back-tracked by the rule pre_back. The backtracking enables the “forward” rules to find new undiscovered nodes again.

The rules pre_forward1 and pre_forward2 also increment the counter and construct the linked list of red edges. The reason we need both rules is to cover both cases of whether the newest entry of the list is also the current red root or not.

4.3.5 The Procedure FindEdge

The procedure FindEdge, depicted in Figures 4.15 and 4.16, serves to find a minimum-weight edge between the current tree (blue nodes) to the rest of the graph (red nodes) using DFS, and to mark it green. If among said minimum edges is an already selected (green) one, it will stay selected, and no additional edge is selected for the current tree. If this were not the case, the selected edges would form a cycle on a 3-cycle whose edges have equal weight for instance, causing the output MST not to be a tree.

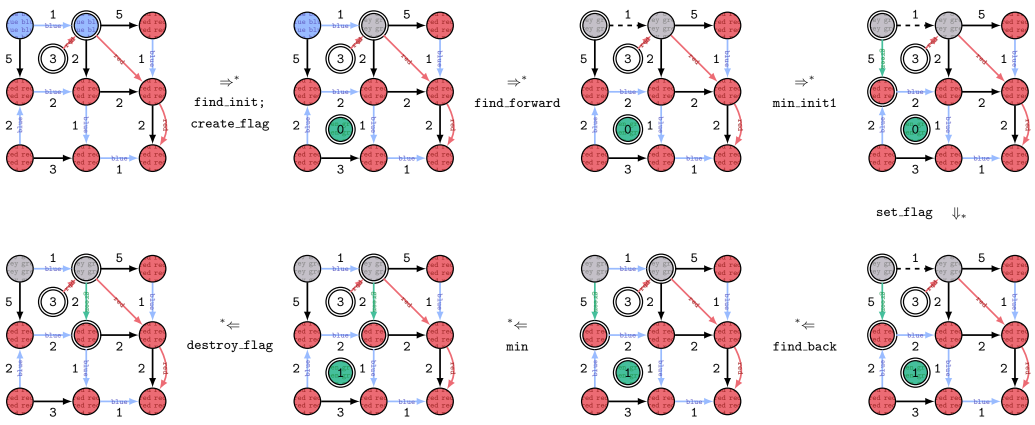


Figure 4.14: Example derivation sequence of `FindEdge`

Let us examine the example execution of `FindEdge` in Figure 4.14. It is part of the transition from the fifth to the sixth graph labelled `TreesLoop!` in the example execution of `mst-boruvka` in Figure 4.11. We start with a graph where the current tree has blue nodes to distinguish it from the rest of the graph. This was done using the procedure `ColourBlue`, which is always called before `FindEdge` as defined in the procedure `TraverseTree` in Figure 4.12. The nodes of the tree are turned grey, but are still distinguishable from the the rest of the graph, which has red nodes.

The procedure `FindEdge` starts by turning the blue root grey, and creating a green root which serves as a flag indicating whether the minimum edge has been initialised yet. The flag is `1` if initialisation has already happened, and `0` otherwise.

We enter the loop `FindLoop!` and apply `find_forward` to move the root along in the current tree in a depth-first fashion. The flag is not yet set to `1`, so we call `MinSetup` to initialise the minimum edge using `min_init1`. The rule `min_init2` exists in case the grey root's only incident edge has already been selected (marked green) when the procedure `FindEdge` was applied to a different tree. An edge selected from both this tree and another tree is represented with a label that is a list consisting of the edge weight followed by a `0`. The currently selected minimum edge of the current tree is represented by a green edge incident to a grey as well as a red root.

We then enter the procedure `Success` which minimises the weight among the unmarked edges incident to the current grey root using the procedure

```
FindEdge = find_init; create_flag; FindLoop!; destroy_flag
FindLoop = find_forward!; if flag then Minimise! else
           (try MinSetup); try find_back else break
MinSetup = try min_init2 then Success else
           (try min_init1 then Success)
Success = MinWithS!; set_flag
Minimise = try MinWithN else MinWithoutN
MinWithS = {min_s, min_sn, min_sp, min_snp, min1_st, min2_st}
MinWithN = {min_n, min_np, min_sn, min_snp, min_tn, min_tnp}
MinWithoutN = {min, min_p, min_s, min_sp, min_t, min_tp,
               min1_st, min2_st}
```

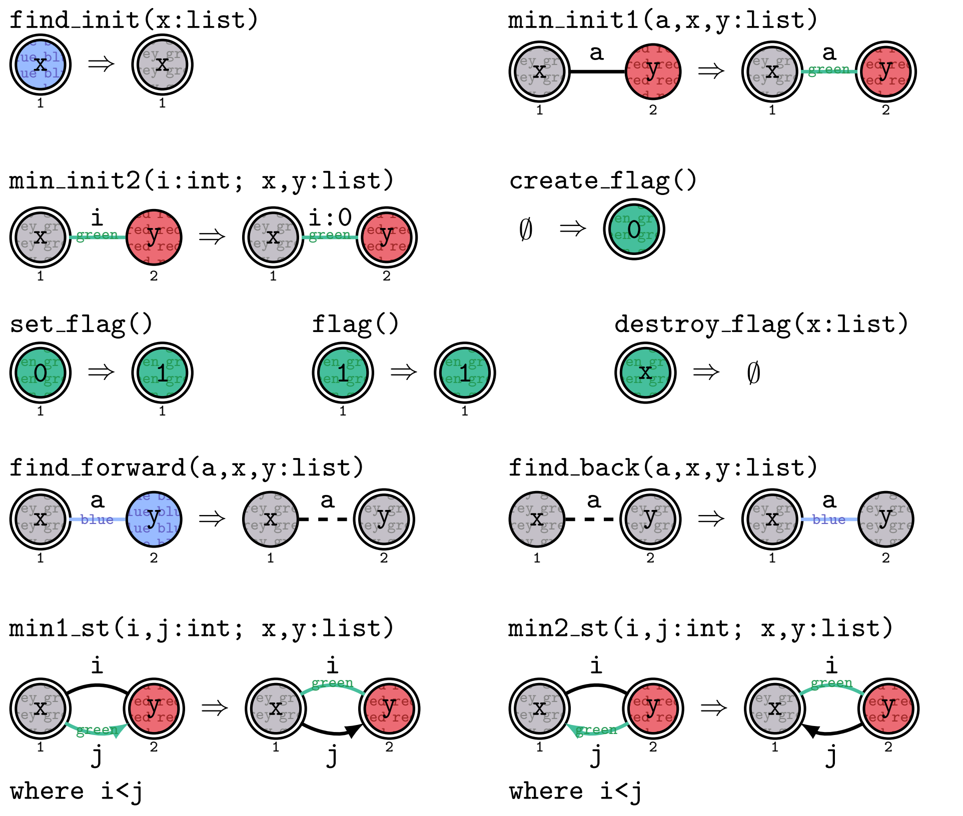


Figure 4.15: The procedure FindEdge

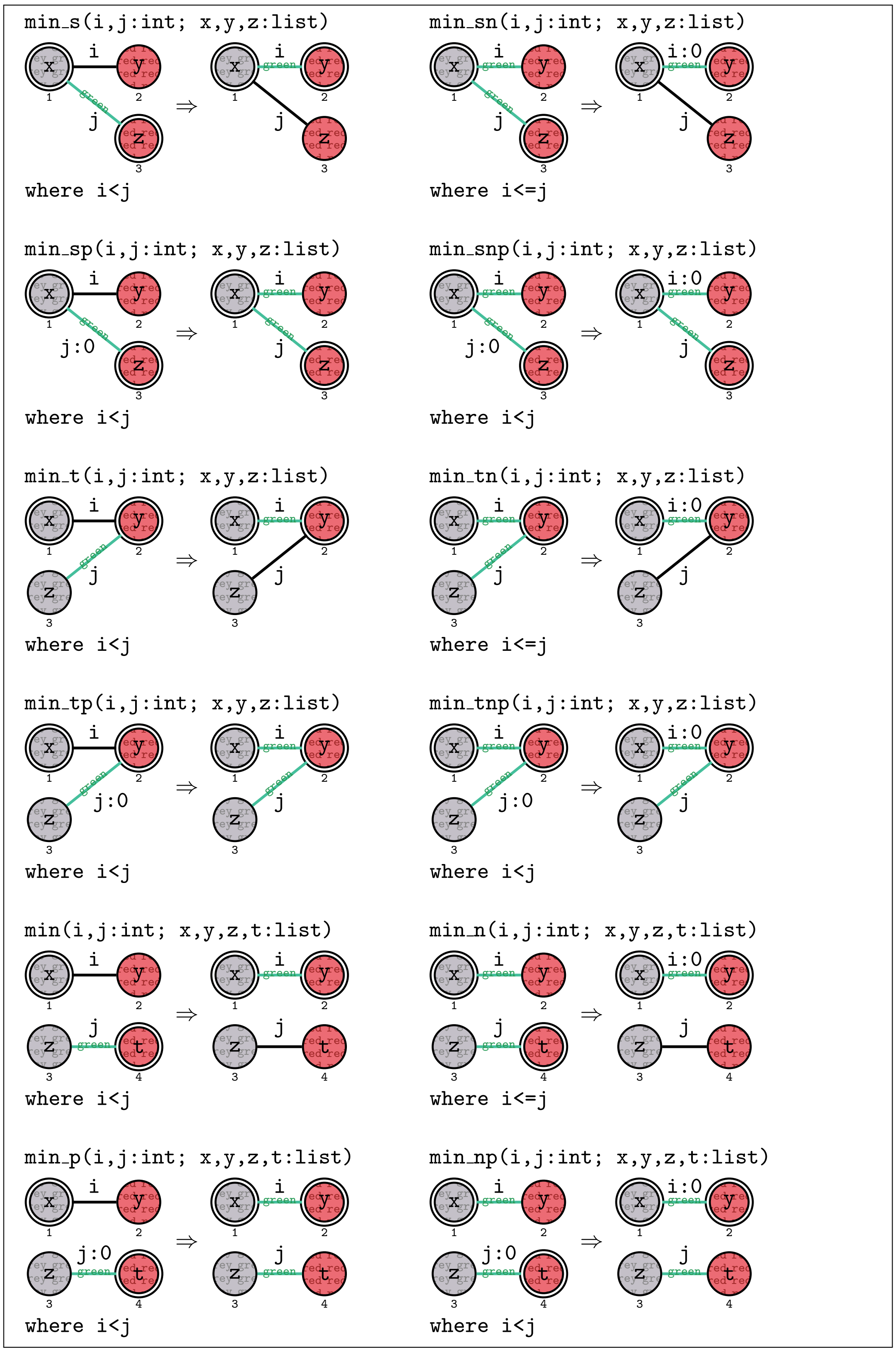

min_s(i,j:int; x,y,z:list)
where i<j
min_sn(i,j:int; x,y,z:list)
where i<=j
min_sp(i,j:int; x,y,z:list)
where i<j
min_snp(i,j:int; x,y,z:list)
where i<j
min_t(i,j:int; x,y,z:list)
where i<j
min_tn(i,j:int; x,y,z:list)
where i<=j
min_tp(i,j:int; x,y,z:list)
where i<j
min_tnp(i,j:int; x,y,z:list)
where i<j
min(i,j:int; x,y,z,t:list)
where i<j
min_n(i,j:int; x,y,z,t:list)
where i<=j
min_p(i,j:int; x,y,z,t:list)
where i<j
min_np(i,j:int; x,y,z,t:list)
where i<j
i
j
i:0
j:0
green
⇒

Figure 4.16: More "min" rules

MinWithS (which only calls rules that minimise edges incident to the current grey root), and then applies set_flag to indicate that the initialisation of the minimum edge is complete.

Next, the rule find_back moves the grey root back through the tree in depth-first fashion. We then enter the next iteration of FindLoop!. The rule find_forward cannot be applied, so we continue with the loop Minimise! since the flag has already been set.

The purpose of the loop Minimise! is to find an edge incident to the grey root with a smaller weight than the currently selected edge. There are 14 different cases we have to distinguish with the rules that update the minimum edge. They can be seen as combinations of the presence or absence of four flags s, t, n, and p, present in the rule names. The flag s is present if the new and previous minimum edge share their "source", i.e. the incident grey node in the current tree. The flag t is present if the new and previous minimum edge share their "target", i.e. the incident red node outside of the current tree. The flag n denotes that the new minimum edge is also a selected minimum edge of a different tree from a previous call of FindEdge. The presence of flag p indicates that the previous minimum edge has already been selected for a different tree. These edges are denoted by a 0 being appended to their label. They need to be distinguished since their green mark needs to be preserved in order for the program to work correctly.

The "min" rules with both the s and t flags, i.e. the ones minimising parallel edges, are a special case. We omit the cases that involve previously selected edges (flags n or p) since such an edge would have already been minimised over its parallel edges by previous applications of min1_st and min2_st. We use two rules with directed edges labelled j instead of one rule with a bidirectional edge labelled j due to parallel bidirectional edges being disallowed by GP 2. This is because, if the parallel bidirectional edges are indistinguishable in the left hand side of a rule, the result of the rule application is not necessarily unique up to isomorphism, since it could leave the host graph with an edge in one of two possible directions.

In order to prioritise edges that have already been selected for different trees, we call the rules of the procedure MinWithN first. They consist of the rules with flag n. We can then call the rest of the rules using MinWithoutN. Note that min_sn, min_tn, and min_n (i.e. "min" rules with n but not p) are the only rules that can be applied if the weights are equal. This is because they are the only ones selecting a previously selected edge, which we prioritise. Making the other rules applicable on equal weights can lead to non-termination. In our example, min is applied.

Finally, the DFS terminates and the rule destroy_flag deletes the temporary flag needed for this procedure. The flag could have been implemented as an additional list entry of the unmarked root, but is chosen to be its own green root for the sake of semantic clarity.

4.3.6 The Procedure GrowForest

The procedure `GrowForest` depicted in Figure 4.17 serves to turn the edges selected by `FindEdge!` (green mark) into edges of the forest (blue mark), thus merging some of the trees. Graphs 6 and 7 in the example execution of `mst-boruvka` in Figure 4.11 exemplify input and output graph of `GrowForest`.

```
GrowForest = grow_init; GrowLoop!; GrowClean!; unroot_red
GrowLoop = GrowTree!  try next_root else break
GrowTree = down!; add_edge!; try up else break
GrowClean = try ColourRed; try previous_root else break
```

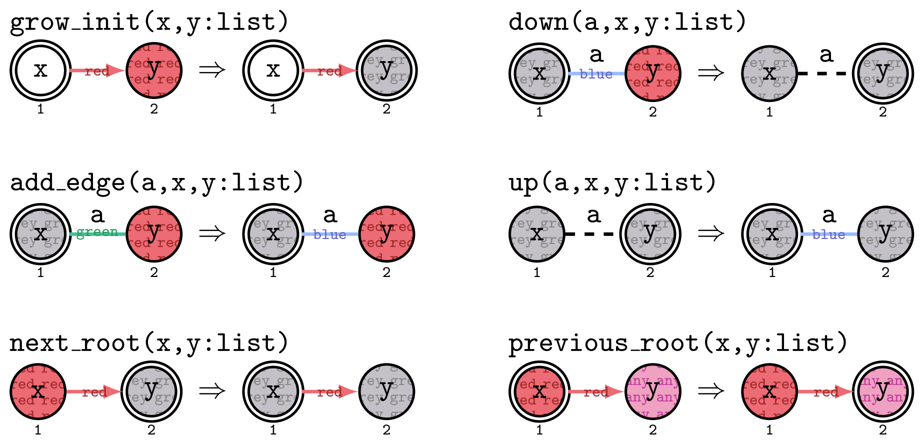


Figure 4.17: The procedure `GrowForest`

The procedure `GrowForest` traverses the graph by iterating through the list of trees, and conducting a DFS on each tree. `next_root` helps iterate through the list in the direction opposite to the orientation of the red edges.

The rules `down` and `up` play the roles of `forward` and `back` in a DFS. They use blue edges to ensure only the current tree is traversed. `add_edge!` is called right before `up` to turn all green edges adjacent to the grey root blue. After the `up` rule is applied to a grey root, it is not visited again by the DFS, ensuring the new blue edges will not be traversed. Future DFSs will also not traverse these edges since one of its adjacent nodes is grey.

The loop `CleanUp!` iterates through the list of trees in the direction opposite to `GrowTree!` and calls `ColourRed` on each tree to mark the nodes red again.

4.3.7 Other Procedures

The program `mst-boruvka` calls several procedures to maintain the list data structure or to prepare the graph for the next step. In this subsection, we describe, `ColourBlue` in Figure 4.18, `ColourRed` in Figure 4.19, `MarkForDeletion` in Figure 4.20, and `Rewind` in Figure 4.21.

```
ColourBlue = blue_init; BlueLoop!
BlueLoop = blue_forward!; try blue_back else break
```

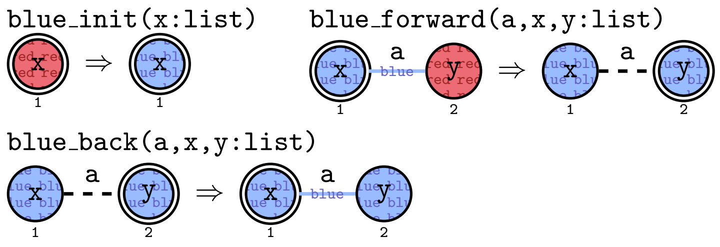


Figure 4.18: The procedure `ColourBlue`

```
ColourRed = red_init; RedLoop!
RedLoop = red_forward!; try red_back else break
```

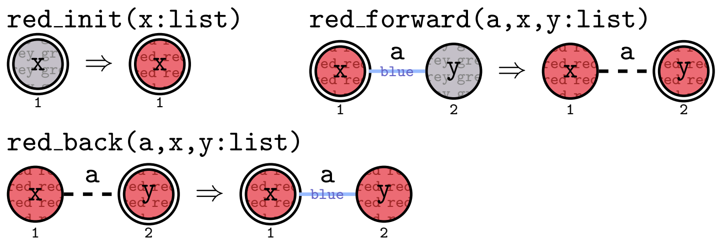


Figure 4.19: The procedure `ColourRed`

The procedure `ColourBlue` uses DFS to turn the nodes of a tree from red to blue, and the procedure `ColourRed` to turn the nodes of a tree from grey to red.

The procedure `MarkForDeletion` determines whether the current tree needs to be removed from the list of trees or not. This needs to be done

```
MarkForDeletion = try clean else Mark; unroot_red
Mark = if red_loop then skip else add_loop
```

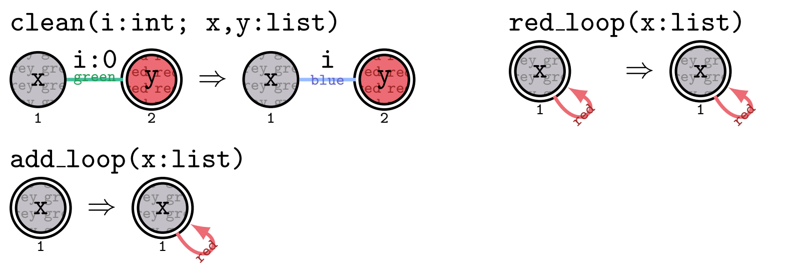


Figure 4.20: The procedure `MarkForDeletion`

```
Rewind = try remove_mid else RemoveEnd
RemoveEnd = try {remove_top, remove_bottom} else keep
```

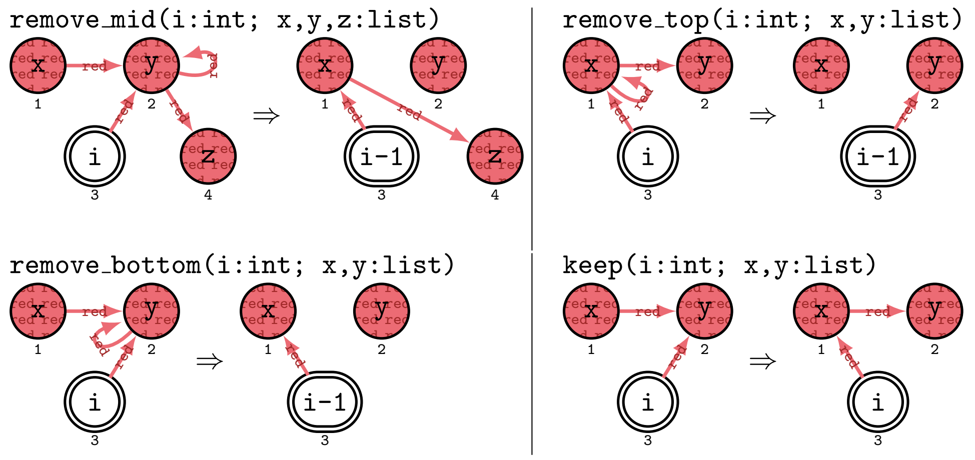

remove_mid(i:int; x,y,z:list)
remove_top(i:int; x,y:list)
remove_bottom(i:int; x,y:list)
keep(i:int; x,y:list)
red
i
i-1

Figure 4.21: The procedure `Rewind`

when the current tree is being merged with another tree in the procedure `GrowForest`. However, in a set of trees that are merged into one tree, exactly one of them needs to be kept as an entry in the list. This is done by exploiting the fact that exactly one of the green edges used to merge that set of trees must have been selected by two different trees. If none of the edges fulfil that condition, the merging would introduce a cycle. If multiple edges fulfil it, the trees are merged into a forest, and not a single tree. Hence the trees that select a previously selected edge are kept as an entry in the list. The rule `clean` easily detects these edges since their label is a list of their edge weight followed by a `0`.

The procedure `Rewind` returns the pointer to the beginning of the list of trees. On the way, it removes list entries marked for deletion with a red loop, and updates the pointer's label which represents the number of trees in the list.

4.3.8 Empirical Performance Results

On the graph classes we tested, time measurements as illustrated in Figures 4.22 and 4.23 show subquadratic growth on square grids and fixed degree wheels, and polynomial growth on unbounded degree wheels. These figures contain error bars, an addition by this thesis' author since previous performance measurements and visualisations are made by a paper's co-author.

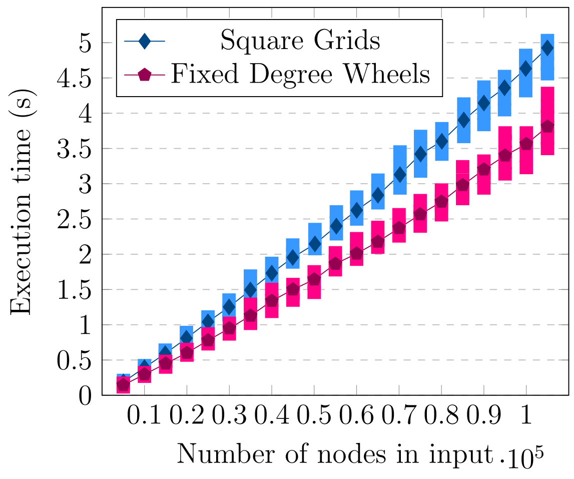


Figure 4.22: Measured performance of `mst-boruvka` on graphs of bounded degree

The execution time of the program `mst-boruvka` is measured on square grids, fixed degree wheels, and unbounded degree wheels. The k^{th} *square grid* is a $k \times k$ grid graph as depicted in Figure 4.11. Figure 4.10 depicts a wheel graph with 8 spokes. The k^{th} *fixed degree wheel* is a wheel graph with 16 spokes, each of which consist of a path graph with k edges. The k^{th} *unbounded degree wheel* is a wheel graph with k spokes.

The edge weights of the input graphs are randomly generated integers between 1 and 1000. The number of nodes of the square grids and fixed degree wheels ranges up to over 100000, and that of the unbounded degree wheels to almost 35000. For each graph of a given size, the execution time depicted with shapes is the average execution time of `mst-boruvka` on copies of that graph with at least 20 random weight distributions. The bars around those

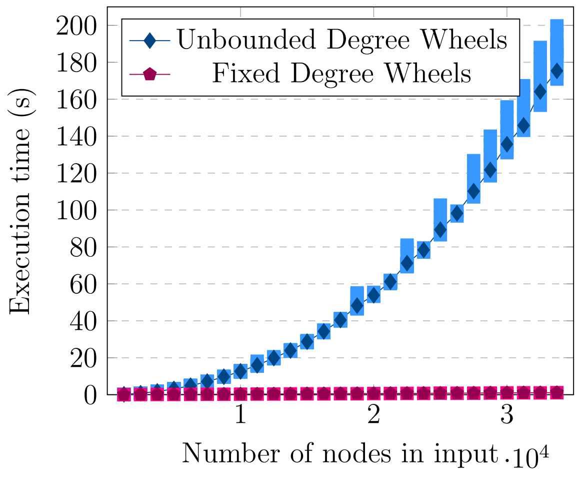


Figure 4.23: Measured performance of `mst-boruvka` on unboundeed degree wheels

data points show the range between the minimum and maximum measured execution time for that graph. The extent of that range can be attributed to differing random weight distributions used for each time measurement. With a fixed weight distribution, that range is much smaller.

Figure 4.22 shows that `mst-boruvka` is subquadratic and close to linear on fixed degree wheels and square grids. We expect the time complexity to be $O(m \log n)$, where m is the number of edges and n the number of nodes, akin to those of standard implementations of Boruvka's MST algorithm [76]. Note that, in order to reach this time complexity in GP 2, the use of root nodes is necessary.

In Figure 4.23, `mst-boruvka` is seen to be of an order worse than $m \log n$ on unbounded degree wheels. In fact, we conjecture it to be quadratic. GP 2 programs that are non-destructive in that they preserve the input graph seem to require at least quadratic time on unbounded degree input graphs. For example, consider `MinWithN!` seen in Figures 4.15 and 4.16. In each case, it has to match a root (say u) and an adjacent non-root (say v) as long as possible. Assume that in the host graph, a root that is a valid match for u has a linear number of adjacent nodes, all of which are a valid match for v. Assume that the first time `MinWithN` is called, the node with the highest edge weight is matched as v. The program only needs to check one node since every node is a valid match. Then assume that the second time `MinWithN` is called, the node with the next highest edge weight is matched. In the worst case, two nodes have to be checked for a valid match. Summing up the number of nodes that are checked if we continue this pattern, we get a sum of consecutive integers with a linear number of terms, which is quadratic. Hence the quadratic time complexity.

Furthermore, procedures that are based on depth-first search and preserve their input, such as the procedure `ColourBlue` in Figure 4.18, have quadratic time complexity on unbounded degree graphs. The rule `blue_back` looks for a dashed edge around the blue root. Assume the blue root has degree that is linear in the number of edges of a graph class the input graph belongs to. Only one of its adjacent edges can be dashed since the dashed edges form a path from the blue root to the origin of the DFS. Since there's only one valid match for the dashed edge, the rule application takes linear time. Every node of the input has to play the role of the blue root in `blue_back` at some point.

The execution time on square grids is slower than that on fixed degree wheels by a constant factor. This is likely due to the fact that a large part of fixed degree wheels consists of path graphs, in which separate trees often share a minimum edge. So `MinWithN` is applied more often in fixed degree wheels than in square grids. Hence more rules (those of `MinWithN`) generally have to be called in square grids.

4.4 Related Work: Case Studies and Benchmarking

Benchmarking for graph transformation is done in [89]. The authors establish a systematic way to compare the efficiency of different graph transformation tools on a variety of test programs.

The paper [83] is a case study on the generation of Sierpinski triangles that compares the efficiency of different graph transformation languages. No real complexity analysis is done, but benchmarks are given.

There are several case studies on a concrete time bound on GP 2 programs that implement well-known algorithms. These include minimising finite automata [61], 2-colouring [11], as well as transitive closure, node colouring, cycle checking, and series-parallel graphs in [59]. As far as we are aware, there are no such case studies in other rule-based graph transformation languages.

However, there is research into the complexity of graph-related algorithms that decide membership, such as graph grammar parsing. When it comes to parsing hyperedge replacement grammars and extensions thereof for instance, it is shown to be NP-complete in some cases [51, 31]. Parsing these grammars can also be accomplished using graph transformation rules. The time complexities of efficient algorithms like that is shown for subsets of hyperedge replacement grammars [32], as well as contextual hyperedge replacement grammars [33].

5 Space Compression

Schönhage's storage modification machines (SMMs) and Kolmogorov-Uspenskii machines (KUMs) are graph-based computation models that do not use graph transformation rules. They have the remarkable feature of being able to simulate Turing machines using less space with only quadratic time overhead, using a uniform space measure [85]. Although these are graph-based models, the literature on rule-based graph transformation so far seems to have ignored the work of Schönhage [69] and Kolmogorov [49], and in particular the results of van Emde Boas [85] and Luginbuhl and Loui [52].

One fundamental difference between SMMs and KUMs on the one hand, and GP 2 on the other hand is that the inputs and outputs of SMMs and KUMs are strings, whereas GP 2 programs compute relations on graphs. Moreover, SMMs and KUMs differ from modern graph transformation languages in that they use low-level pointer instructions instead of pattern-based rules where the programmer can specify a subgraph and how it is transformed, allowing for a more natural implementation of graph algorithms.

An advantage of structured programming languages such as GP 2 is the absence of "go to" statements which are considered to be harmful [27]. Furthermore, GP 2 is based on the double-pushout approach for graph transformation, which comes with a large amount of theoretical results. We are not aware of any comparable theory for SMMs and KUMs.

The fact that the above mentioned space compression property applies to a high-level language is not obvious. Even so, we show in this chapter that GP 2 exhibits the same space compression feature. Specifically, we show that a Turing machine using $\mathrm{O}(s(n) \log s(n))$ space can be simulated in $\mathrm{O}(s(n))$ space with quadratic time overhead, where $s(n)$ is an arbitrary function. The simulation is asymptotically more efficient than the original (for non-constant $s(n)$). Rather than using full GP 2, we show that a subset of the language suffices to establish the compression result.

Note that another way to show space compression is to efficiently simulate SMMs in general. This would also mean other properties are applicable to

GP 2, such as the real-time simulation of Turing machines shown by Luginbuhl and Loui [52]. This chapter's approach however comes with a concrete class of programs that show in much detail that the simulation is possible. In particular, we show *how* to establish a result akin to van Emde Boas' in a rule-based graph transformation language such as GP 2. Moreover, the van Emde Boas paper lacks a lot of technical details, which we provide in our setting.

In Section 5.1, we define the subset of GP 2 sufficient to get our results. We also show how this subset finds matches for rules efficiently.

We show how we simulate Turing machines in Section 5.2. First, we define what Turing machines we are simulating in Subsection 5.2.1. We then outline the basic idea of how we achieve space compression in Subsection 5.2.2. In Subsection 5.2.3, we describe exactly how Turing machine configurations are represented as graphs. Next, we give and describe the class of programs that simulate Turing machines in Subsection 5.2.4. We then provide an example of a Turing machine simulation in Subsection 5.2.5. Finally, we show the correctness of the simulation in Subsection 5.2.6.

In Section 5.3 we show that our simulation is in a class of efficient models based on GP 2, which comes with a time complexity measure.

Next, in Section 5.4, we show the aforementioned time and space complexities of the simulation, and discuss the use of logarithmic versus uniform space measures.

To end this chapter, in Section 5.5, we talk about the relation between space compression and the invariance thesis in the literature.

5.1 Small GP 2

Throughout this chapter, we use a subset of GP 2 called SGP 2 (Small GP 2), which suffices to establish the simulation and compression result. This subset uses simpler host graphs. Specifically labels are limited to being either an integer or one of three characters, and lists of these are not allowed. Rules are simpler as well. They use neither label expressions with variables, nor application conditions.

Graphs are defined as before, but with labels generated by the simpler grammar in Figure 5.1

Both nodes and edges are labelled either with an integer, with the special characters `"L"`, `"R"`, `"I"`, or with the constant `empty`. Figure 5.1 defines the labels that may occur in both rules and host graphs. In this chapter, we only use integers 0, 1, and 2 in labels.

Label	::=	Atom [Mark]
Atom	::=	Integer | `"L"` | `"R"` | `"I"` | `empty`
Mark	::=	red | green | blue | grey | dashed

Figure 5.1: Abstract syntax of host and rule graph labels

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

Figure 5.2: Abstract syntax of Small GP 2 programs

Rules, graph morphisms, and rule applications are as previously defined for GP 2. SGP 2 programs follow the syntax described in Figure 5.2, which is a subset of the GP 2 semantics. The structural operational semantics of programs is as defined for GP 2.

5.2 Simulating Turing Machines

We describe Turing machines and how we simulate them in this section.

5.2.1 Off-Line Turing Machines

In this chapter, we consider deterministic off-line Turing machines in order to exhibit that space compression can happen with sublinear space complexities.

The input tape is finite and uses a binary encoding, i.e. the input alphabet is $\Sigma = \{0, 1\}$. It is read-only (symbols cannot be modified) with a tape head pointing towards the tape square being read. The input tape contains the input of the Turing machine, and its length is the length of the input.

Off-line Turing machines additionally have a working tape, a one-way infinite tape that can be read and written on (symbols can be modified). The tape heads on both tapes move simultaneously, but independently. We

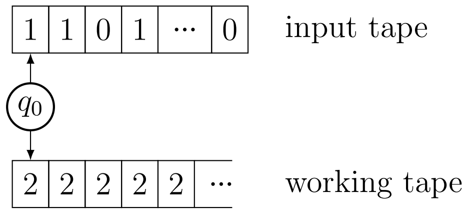


Figure 5.3: Initial configuration of an off-line Turing machine

allow for an additional symbol 2 as a blank symbol or separator, i.e. the working tape alphabet is $\Gamma = \{0, 1, 2\}$. We use 2 instead of another symbol because it is useful for the working tape to be encoded in base 3 for the purposes of our simulation. Initially, the working tape contains only blank symbols, i.e. the symbol 2. We define *working tape contents* to be the section of the working tape starting at the beginning of the tape, and ending with the final nonblank (not a 2) tape square that is followed by only blanks (only 2's).

We consider the finite state set Q to consist of integers. There is an initial state q_0, an accepting state h_a, and a transition function $\delta\colon Q \times \Sigma \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\} \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$. Consider the transition $\delta(q, a, b) = \langle q', b', D_1, D_2\rangle$. This means in state q, with a on the input tape, and b on the working tape, the machine goes into state q', replaces the working tape symbol with b', moves the input tape head in direction D_1, and moves the working tape head in direction D_2. Transitions and computations are defined as usual.

The time complexity of a Turing machine is the function which, to an integer n, associates the maximum number of transitions (or length of computation) starting from an input of size n. The space complexity is the function which associates to n the maximum number of used working tape squares for an input of size n.

5.2.2 The Basic Idea of the Simulation

We simulate a Turing machine M with an SGP 2 program called $\texttt{Sim}(M)$, as specified in Figures 5.6 to 5.11. Input symbols, tape symbols, and state names are encoded as integers. The compression is achieved by representing blocks of Turing machine tape squares as single nodes. This idea is shared with van Emde Boas' representation [85]. By representing graphs in SGP 2, we can have an edge that provides a direct link to a distant list element, which is impossible in a Turing machine.

The space compression comes from how we represent the working tape, which is illustrated in Figure 5.4 (note that the text not contained within

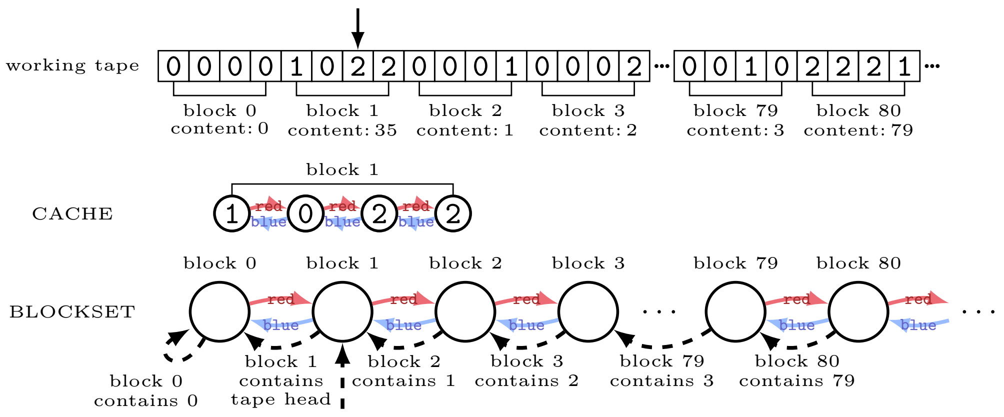


Figure 5.4: Graph representation of the working tape of a Turing machine.

nodes or squares is explanatory). The working tape is divided into blocks of size $c = 4$. This value is chosen for the purpose of illustration. The actual simulation starts with $c = 2$ and expands as needed. Each block can be interpreted as a 4-digit number in base 3. Block 1 for instance contains the ternary number 1022. In base 10, that is $1{\cdot}3^3{+}0{\cdot}3^2{+}2{\cdot}3^1{+}2{\cdot}3^0 = 27{+}6{+}2 = 35$. Hence we say the content of block 1 is 35. The graph CACHE represents the block currently containing the working tape head, which is block 1 in this case. We call this block the *active block*. CACHE is a doubly-linked list whose 4 nodes are labelled with the symbol of the corresponding tape square. The graph BLOCKSET is a less direct representation of the working tape. Each of its nodes represents a block of working tape squares instead of a single square. Instead of storing the content of a block as a spacious label, we store it as a single dashed edge. (This is the crucial idea making compression possible.) Block 79 for instance has content 3, so there is a dashed edge from node “block 79” to node “block 3”. The content of a block is converted into the index of a node in BLOCKSET (seen as a doubly-linked list). We call the nodes of BLOCKSET *block nodes*.

In general, if block i has content j, there is dashed edge in BLOCKSET from node “block i” to node “block j”. The only exception is the node representing the active block, in this case node “block 1”, which has a dashed edge pointing towards node “block 0”. This is by convention for the following reason. The content of “block 1” is stored in CACHE, so storing it in BLOCKSET as well would be redundant. So instead of using the dashed edge to store the content of “block 1”, the dashed edge is directed to the leftmost node (“block 0”) by convention. Since that dashed edge now holds no information, when labels in CACHE are changed, the dashed edge does not need to be updated. Once the tape head moves to a different block, and the active block changes, that dashed edge is redirected to represent to content of its block again, and the outgoing dashed edge of the new active block node is redirected to node "block 0".

The only part of the tape we need to represent is the working tape contents, since only blanks follow. For a given working tape, let b be the minimum number of blocks that contain the working tape contents. So we only need b nodes in BLOCKSET. Remember $c = 4$ is the number of tape squares per block. Since we are converting the content of a block into a list index in BLOCKSET, the list needs to be long enough to have that index. In other words, the number of nodes b in BLOCKSET needs to be greater than the largest possible block content. A 4-digit ternary number can have $3^c = 3^4 = 81$ different values, so we need $b \geq 3^c = 81$. Since our aim is compressing space, we want to use the lowest number of nodes, so we pick $b = 3^c = 81$. If the tape contents exceed the size b blocks, we extend our representation by incrementing c, and updating $b = 3^c$. These new values become $c = 5$ and $b = 243$.

After incrementation, the number of nodes in CACHE is $c = 5$, and the number of edges $2 \cdot (c - 1) = 8$, so 13 in total. The number of nodes in BLOCKSET is $b = 243$, and the number of edges $b + 2 \cdot (b - 1) = 727$, so 970 in total. However the number of squares on the working tape is $b \cdot c = 19683$, which is significantly larger. The graph representation uses less space for $c \geq 5$. Let us describe this behaviour asymptotically. We have $c = \log_3 b$. The number of outgoing edges of a node is bounded by 3, hence the number of nodes and edges is bounded by $4(b + c)$, which is $\mathrm{O}(b + \log_3 b) = \mathrm{O}(b)$. The number of working tape squares on the other hand is $\mathrm{O}(b \log_3 b)$.

5.2.3 Turing Machine Configurations as Graphs

Figure 5.3 shows the initial configuration of a Turing machine with its input and working tapes and tape heads, as well as the initial state q_0. The corresponding graph can be seen in Figure 5.5.

Let $\mathrm{O}(s(n) \log s(n))$ be the space complexity of a Turing machine, where n is the size of the input tape, and s some function. For the rest of this chapter, we omit the base of the logarithm since $\mathrm{O}(\log_3 n) = \mathrm{O}(\log_c n)$ for any constant $c > 1$. If we choose the number of blocks to be $b = \mathrm{O}(s(n))$, we can represent $\mathrm{O}(s(n) \log s(n))$ tape squares in space $\mathrm{O}(s(n))$, as outlined by the previous subsection. This is proved in Theorem 5.6.

The graph representation consists of the *central node*, labelled by the initial state q_0 as represented by some integer, and the subgraphs INPUT, BLOCKSET, and CACHE. We represent 'left' and 'right' positioning by blue

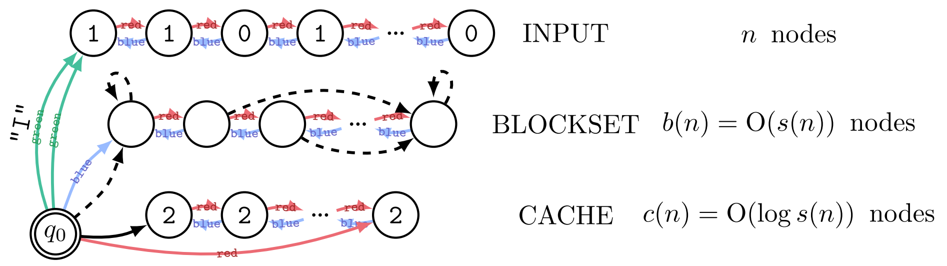

1
1
0
1
...
0
INPUT
n nodes
"I"
green
green
blue
BLOCKSET
$b(n) = \mathrm{O}(s(n))$ nodes
2
2
2
CACHE
$c(n) = \mathrm{O}(\log s(n))$ nodes
q_0
red
blue

Figure 5.5: Graph representation of the initial configuration of a Turing machine of space complexity $\mathrm{O}(s(n) \log s(n))$.

and red edges respectively. These edges will not be modified by the program. The unmarked and unlabelled green edges mark the positions of the tape heads. Dashed edges serve to encode the state of BLOCKSET and how it relates to CACHE, as described in the previous subsection. The dashed edge from the central node points towards the block node whose block currently contains the working tape head (the active block node). By convention, there is a dashed edge from the active block node to the first block node, as stated in the previous subsection. Hence there is always a path of dashed edges from the central node to the active block node to the first block node.

The central node is a root that is never moved. Parts of the graph such as tape head positions and tape ends can be accessed efficiently via the central node's outgoing edges. The only other time roots are used in the simulation is when extending the tape. We use roots to keep track of the end of the tape, and later to traverse the tape. For a more in-depth explanation of how the roots move, see Subsection 5.2.4.

INPUT represents the input tape of the Turing machine, and the position of the tape head is represented by an unmarked green edge from the central node. The green edge labelled `"I"` always points to the leftmost square of the input tape. The working tape is represented by BLOCKSET and CACHE. These subgraphs are implemented as doubly linked lists in order to enable fast rule matching.

Each node in BLOCKSET represents a block of c nodes of the working tape. The blue edge from the central node points towards the leftmost node, and the dashed edge towards the block that contains the position of the tape head.

CACHE is the block of the working tape that contains the tape head. Its position within BLOCKSET is marked with a dashed edge from the central node. The red edge from the central node points towards the rightmost node of the cache, and the unmarked edge towards the current position of the tape head. The node labels represent the content of each square.

The operations of the Turing machine happen within CACHE. If the tape head moves out of bounds of CACHE, we encode the content of CACHE within BLOCKSET, move on to another block, and decode it into CACHE.

Initially, the tape heads are on the first square of their tapes, and the working tape is blank (i.e. filled with 2's). Remember CACHE has c nodes and BLOCKSET 3^c. We start the simulation with $c = 2$ because it is the smallest nontrivial value for c. If $c = 1$, then each block would represent a single tape square, and no space would be gained. If we run out of nodes in BLOCKSET, c is incremented and BLOCKSET adjusted accordingly.

5.2.4 The Simulation `Sim`(M)

In this subsection, we assume we have a Turing machine M and describe `Sim`(M), the SGP 2 program that simulates M. The program `Sim`(M) takes inputs that consist of graphs such as INPUT together with the central node and green edges as represented in Figure 5.5. The overall behaviour of the program is that it starts the simulation with a small BLOCKSET and CACHE. If we run out of tape squares, we reset to the initial configuration, extend BLOCKSET and CACHE, and restart the simulation.

Figure 5.6 contains the main control sequence of `Sim`(M). First the rule `setup` is called. It matches the root and constructs CACHE with two nodes and BLOCKSET with nine nodes, as seen in Figure 5.5. We omit the definition of that rule since it is straightforward.

Next we have the loop `Simulate!`, which means the procedure `Simulate` is applied until no longer possible. The loop body first calls `Transitions` (Figure 5.7), a procedure that applies the transition function to the current state of M. If `Transitions` cannot apply the transition function, the procedure results in failure, and the loop terminates. Next, `try MoveLeft` is called, which means we attempt to apply `MoveLeft` and if it fails, we skip this instruction. The rules in `MoveLeft` detect a label `"L"` on the unmarked edge adjacent to the central node, which is created in `Transitions` if the working tape head needs to move to the left. It then executes that move in CACHE. Analogously, `try MoveRight` attempts to move the working tape head to the right. These rule sets (which are nondeterministic calls of the rules they contain) will fail however if the tape head moves out of bounds of CACHE, which is detected by `Left` or `Right` since the continued presence of the `"L"` or `"R"` label indicates that tape head movement still needs to be done. In that case, `PrevBlock` or `NextBlock` (Figure 5.8) are called in order to move towards the relevant block.

Figure 5.7 contains `Transitions`, the rule set that encodes the transition functions of M. For each entry in the transition table, there is a corre-

```
Main = setup; Simulate!
Simulate = Transitions; try MoveLeft; try MoveRight;
           try Left then PrevBlock; try Right then NextBlock; try Flag then Restart
```

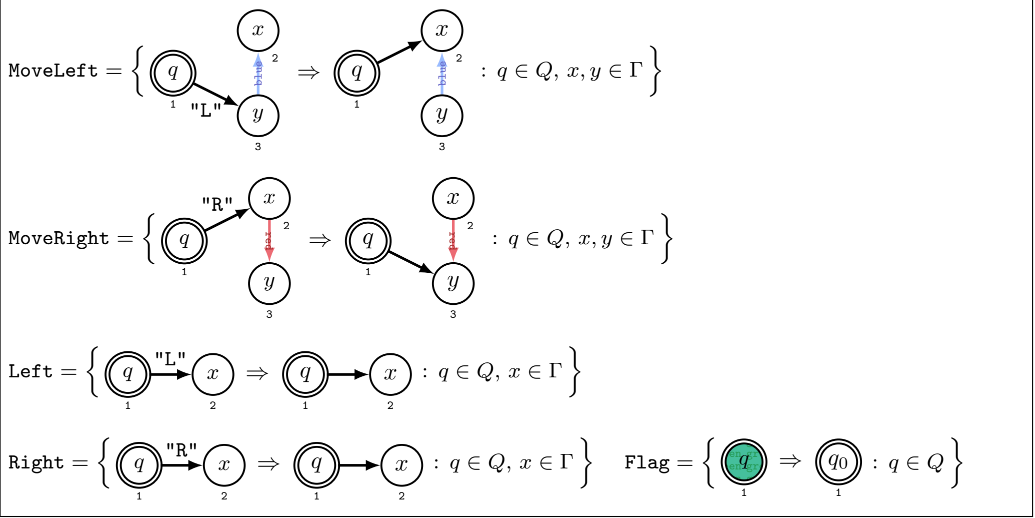

MoveLeft =
"L"
blu
$: q \in Q,\ x, y \in \Gamma$
MoveRight =
"R"
red
$: q \in Q,\ x, y \in \Gamma$
Left =
"L"
$: q \in Q,\ x \in \Gamma$
Right =
"R"
$: q \in Q,\ x \in \Gamma$
Flag =
q_0
$: q \in Q$

Figure 5.6: SGP 2 program `Sim`(M) that simulates Turing machine M.

sponding rule in `Transitions`. The rules are divided into nine categories `transitionXY`, where `X` represents the movement of the input tape head, and `Y` that of the working tape head. Each rule implements the change of labels and the movement of the input tape head directly. For the movement of the working tape head, a label is left behind on the unmarked edge, and the actual movement is handled by `Simulate` in Figure 5.6.

In Figure 5.8, we show the procedures `NextBlock` and `PrevBlock`, which handle movement to another block. We call the node pointed at by a dashed edge from the central node the *active block node*. The procedure `Encode` (Figure 5.9) saves the information from CACHE into BLOCKSET. This process is further described in the example in Subsection 5.2.5. The procedure `CacheInc` increments CACHE as a ternary number. First the rightmost digit is turned into a red root. Then the digit is attempted to be incremented with the rule set `Inc`. If the digit is a `0` or a `1`, incrementation succeeds, the process is marked as finished by turning the red root into an unmarked node, and `CacheInc` terminates. If the digit is a `2` however, `Inc` fails, the `2` is turned into a `0`, and the red root is moved along to the next digit which is attempted to be incremented. If `Increment!` ends without `Inc` ever succeeding, it means that all digits were `2`, and that the red root is still present. In that case, `Reset!` turns all digits into `0`.

The procedure `Decode` in Figure 5.10 decodes the active block in BLOCKSET and stores the information in CACHE. It is analogous to `Encode` (note

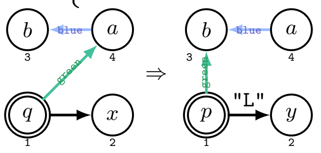
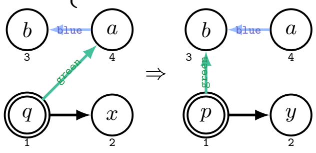
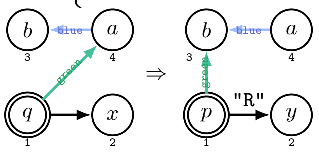
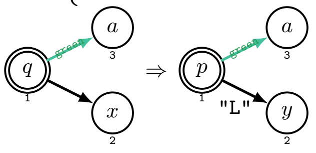
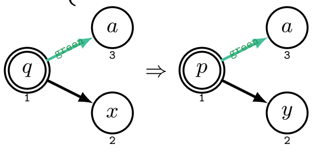
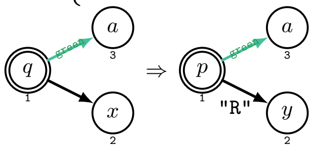
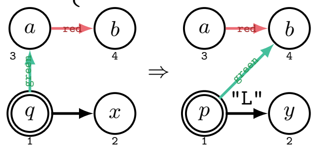
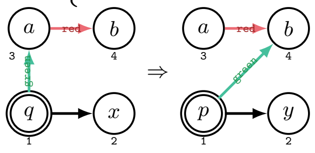
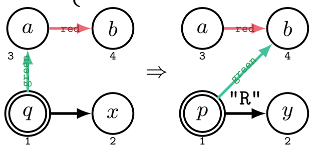
`Transitions` $= \bigcup_{X,Y\in\{\mathrm{L,S,R}\}} XY$		
LL $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{L},\mathrm{L}), b\in\Sigma\Big\}$	LS $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{L},\mathrm{S}), b\in\Sigma\Big\}$	LR $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{L},\mathrm{R}), b\in\Sigma\Big\}$
SL $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{S},\mathrm{L}), b\in\Sigma\Big\}$	SS $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{S},\mathrm{S}), b\in\Sigma\Big\}$	SR $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{S},\mathrm{R}), b\in\Sigma\Big\}$
RL $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{R},\mathrm{L}), b\in\Sigma\Big\}$	RS $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{R},\mathrm{S}), b\in\Sigma\Big\}$	RR $= \Big\{$  $: \delta(q,a,x) = (p,y,\mathrm{R},\mathrm{R}), b\in\Sigma\Big\}$

Figure 5.7: Rule set `Transitions` that models the transitions of a Turing Machine.

```
NextBlock = Encode; try Next then (HeadLeft!; Decode) else (SetFlag; break)
PrevBlock = Encode; Prev; HeadRight!; Decode
```

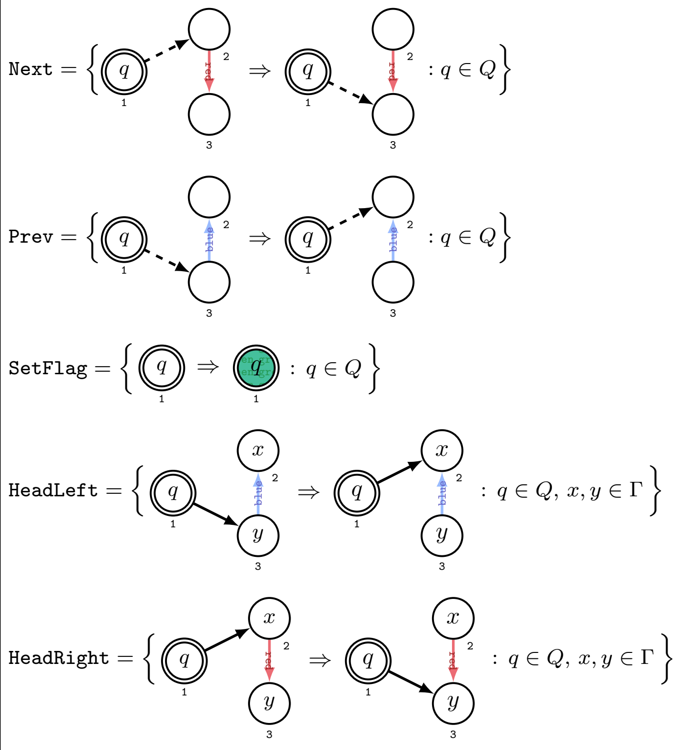

Next = { q ⇒ q : q ∈ Q }
red
Prev = { q ⇒ q : q ∈ Q }
blue
SetFlag = { q ⇒ q : q ∈ Q }
HeadLeft = { q x y ⇒ q x y : q ∈ Q, x, y ∈ Γ }
HeadRight = { q x y ⇒ q x y : q ∈ Q, x, y ∈ Γ }
1
2
3

Figure 5.8: Procedures `NextBlock` and `PrevBlock` that change the active block.

```
Encode = EncodeInit; Encoding!; Update
Encoding = CacheDec; try Finish then break; try next_value
CacheDec = CacheInit; Decrement!; if Unfinished then Reset!
Decrement = try Dec then (Finish; break)
                    else (underflow; try CacheNext else break)
Reset = overflow; try CachePrev else break
```

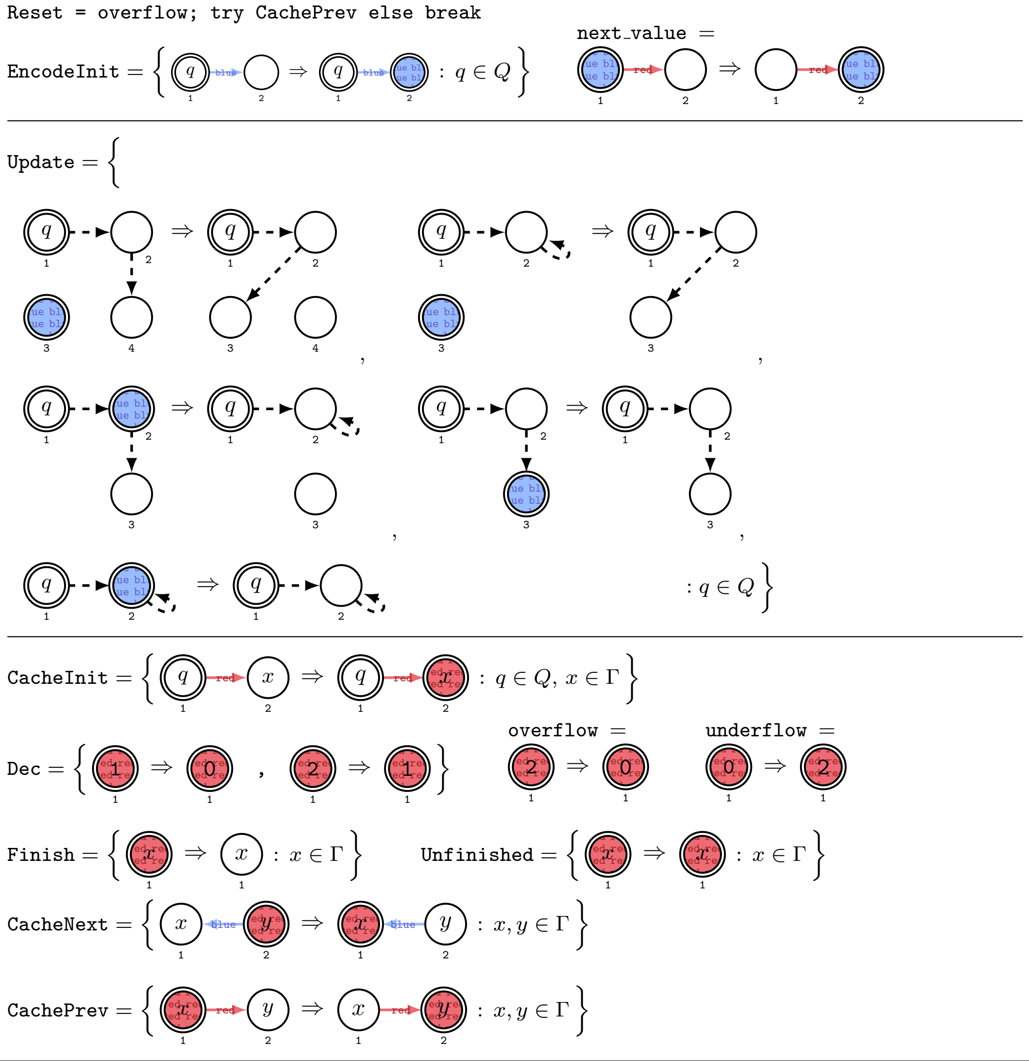


Figure 5.9: Procedure `Encode` that encodes the current block into BLOCK-SET.

```
Decode = DecodeInit; Decoding!; Update
Decoding = try prev_value else break; CacheInc
CacheInc = CacheInit; Increment!; if Unfinished then (Reset!; Finish)
Increment = try Inc then (Finish; break)
                        else (overflow; try CacheNext else break)
```

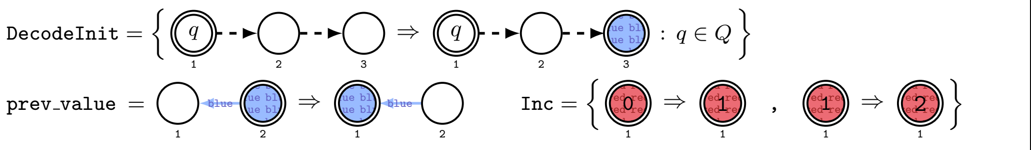


Figure 5.10: Procedure `Decode` that decodes the current block into CACHE.

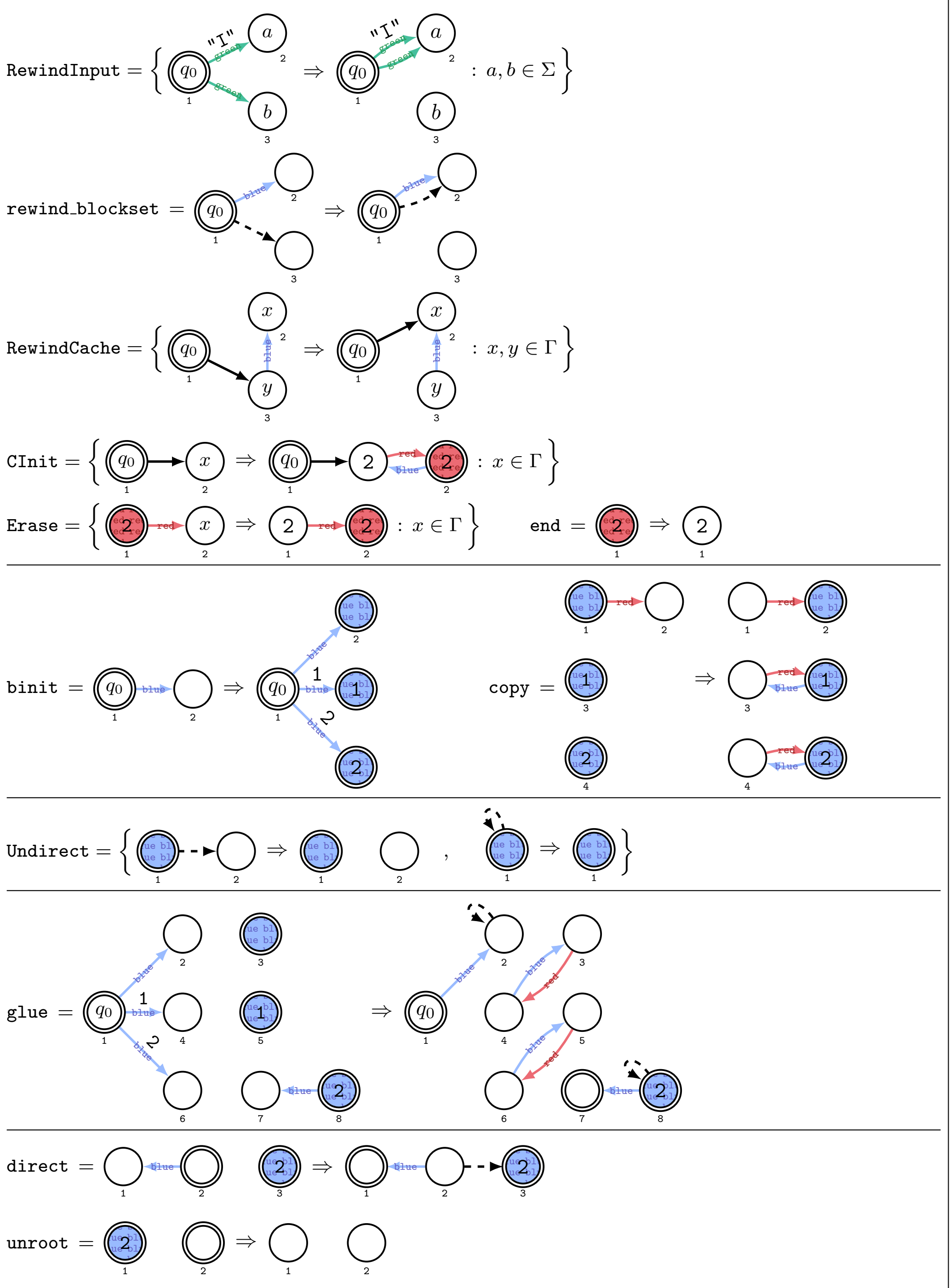
that some of `Decode`'s rules and procedures appear in Figure 5.9). Similarly to `CacheInc`, `CacheDec` decrements CACHE as a ternary number if that number is not 0, and keeps it as 0 if it already is.

Figure 5.11 shows the procedure `Restart` which, when the machine runs out of tape, resets the simulation and extends the tape. CACHE and hence the size of a block is extended by one square, and the size of BLOCKSET is tripled. The procedure `RewindTapes` rewinds the tape heads to the beginning. Next, `ResetCache` resets the content of CACHE back to 2's (the blank symbol). Then, `ResetBlockset` extends BLOCKSET and directs dashed edges to their initial state. In rules `binit` and `copy`, an unlabelled blue root traverses the existing BLOCKSET, while blue roots with labels `1` and `2` traverse two copies of that BLOCKSET as they are being created. In the meantime, `Undirect` deletes dashed edges that are outgoing from block nodes. The rule `glue` appends the copies to the original BLOCKSET, tripling its length. The blue nodes are unrooted except for the final node, which remains a blue root labelled `2`. The penultimate node becomes an unmarked root, which is used in `direct` to traverse BLOCKSET from right to left, creating dashed edges where needed. The rule `unroot` then removes the blue and unmarked roots.

5.2.5 Example

In this subsection, we give an example of a Turing machine simulation, and show how we move from one block to another. Consider a Turing machine that takes as input the number n represented in unary, and writes n in binary on its working tape n times. It is reasonable to assume the machine has a space complexity of $O(n \log n)$. If $n = 6$, the machine uses 18 squares, which are filled with 6 copies of the string `110` (6 in binary). A CACHE size of 2 and a BLOCKSET size of 9 are enough to represent $2 \cdot 9 = 18$ tape squares. Their representation in the simulation has only $2 + 9 = 11$ nodes, which is

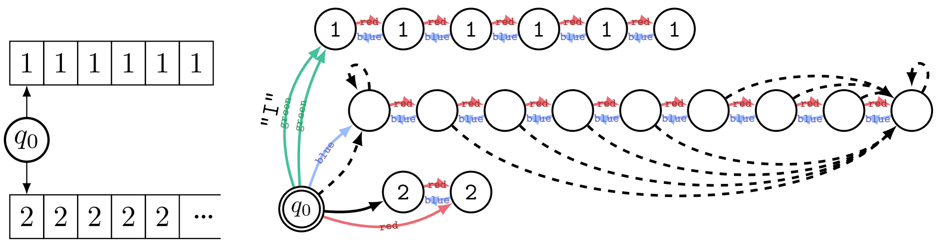
```
Restart = RewindTapes; ResetCache; ResetBlockset
RewindTapes = try RewindInput; try rewind_blockset; try RewindCache!
ResetCache = CInit; Erase!; end
ResetBlockset = binit; try Undirect; (copy; try Undirect)!; glue; direct!; unroot
```



RewindInput = { ⇒ : $a, b \in \Sigma$ }
"I"
green
rewind_blockset = ⇒
blue
RewindCache = { ⇒ : $x, y \in \Gamma$ }
CInit = { ⇒ : $x \in \Gamma$ }
red
blue
Erase = { ⇒ : $x \in \Gamma$ }
end = ⇒
binit = ⇒
copy = ⇒
Undirect = { ⇒ , ⇒ }
glue = ⇒
direct = ⇒
unroot = ⇒

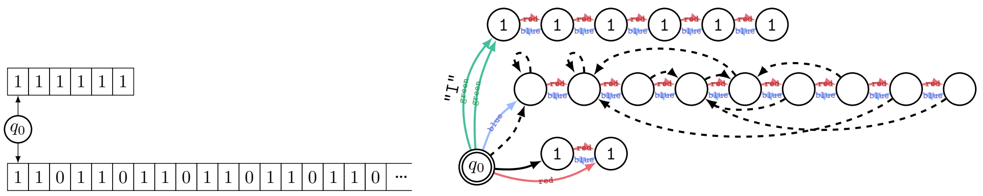
Figure 5.11: Procedure `Restart` that resets the simulation and enlarges the tape.

$O(n + \log n) = O(n)$. The initial state of the machine on input 6 is shown in Figure 5.12, and the final one in Figure 5.13.



(a) Turing machine (b) Simulation

Figure 5.12: Initial state of the example machine



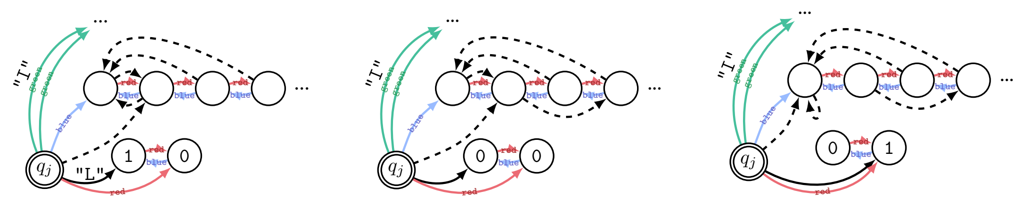
(a) Turing machine (b) Simulation

Figure 5.13: Final state of the example machine

Initially, the working tape is blank (all 2's). Hence most blocks consist of two 2's, i.e. their content is 8 (22 in ternary is 8 in base 10). So most block nodes point towards the block node of index 8 (final block node). The exception is the first node, which points to itself. This is because it represents the block that contains the tape head, which is set to index 0 by convention. In the final state, the working tape is represented in the same manner. Note that even though the working tape content looks binary, 2's (blank symbols) could be present, and hence tape blocks are considered a ternary number. For instance consider the third block (of size 2). It contains `10`, so the third bock node points towards the block node of index 3 (fourth block node).

Let us sketch the behaviour of the machine. First, the input is copied in unary onto the working tape for use as a counter. Then, a binary number to the right of the counter is incremented while traversing the input. The previous step is repeated while decrementing the counter until it reaches 0. Tape contents need to be shifted. The symbol 2 can be used as a separator.

Now let us show what happens when the block has to be changed. Consider the situation where `0100` are the first 4 squares of the working tape and



(a) Symbol "1" written into CACHE.

(b) CACHE loaded into second block.

(c) First block loaded into CACHE.

Figure 5.14: Changing blocks

the tape head is on the third square. Let the next transition write symbol `1` and move the tape head to the left, and change state q_i to state q_j.

Figure 5.14 shows the process of changing the active block. In Subfigure 5.14a, a rule from `Transitions` has just been applied, labelling the left node of CACHE `1`, and the unmarked edge "`L`". The working tape head is yet to be moved. Next, in `Simulate`, `Left` matches and `PrevBlock` is called. The procedure `Encode` loads the content of CACHE into a dashed edge within BLOCKSET. While CACHE is decremented as a ternary number, the outgoing dashed edge of the current (second) BLOCKSET node is shifted to the right. This produces the graph from Subfigure 5.14b. CACHE does not represent the content of the second block anymore, but the dashed edge from the second node in BLOCKSET to the fourth does. Next we move one block to the left with `Prev`, and reposition the tape head in CACHE with `HeadRight!`. We then load the new block into CACHE. While moving the outgoing dashed edge of the current (first) BLOCKSET node to the left, the ternary number in CACHE is incremented. This results in the graph from Subfigure 5.14c. We have now loaded the content of the first block into CACHE, so there is no need to store it with a dashed edge anymore. Hence the dahed edge goes from the first node in BLOCKSET to itself by convention.

5.2.6 Correctness

We define the *configuration* of a Turing machine to consist of the input and working tapes, the position of the tape heads on those tapes, and the current state.

Let us define what graphs programs in `Sim` operate on. We call graphs as in Figure 5.5 *configuration graphs*. These graphs can vary from the depicted graph in the following ways. The targets of the unlabelled green edge, dashed edges, and the unmarked edge can be any node in INPUT, BLOCKSET, and

CACHE respectively. An exception to that is the outgoing dashed edge of the active node in BLOCKSET (target of dashed edge from the root), which is targeted towards the leftmost node in BLOCKSET since the content of that block is currently stored in CACHE. This is a convention that allows the graph representation of a configuration to be unique. The labels of nodes in INPUT, CACHE, and the root node can be any element from the input alphabet, tape alphabet, or set of states respectively, encoded as SGP 2 labels.

Note that the graphs' capacity to represent tape squares is limited by the sizes of BLOCKSET and CACHE. Hence for a non-negative integer k, we define a function enc_k that encodes the configuration S of a Turing machine as the configuration graph $\mathrm{enc}_k(S)$ with 3^{k+2} nodes in BLOCKSET and $k+2$ in CACHE. The integer k corresponds to the number of times the tape is extended by the procedure `Restart`. For a given value of k that is large enough, $\mathrm{enc}_k(S)$ exists and is unique.

For a Turing machine M we denote a single transition from configuration S to S' by $S \Rightarrow_M S'$, and the transitive reflexive closure of that relation using $\Rightarrow_M^*$. Similarly, for an SGP 2 rule `r` we use $S \Rightarrow_{\mathtt{r}} S'$ to denote a change of configuration, and $S \Rightarrow_{\mathtt{P}}^* S'$ for a program `P`.

During the execution of `Sim`(M), graphs that are not configuration graphs are generated. They only differ in terms of edge labels, node marks, roots, and dashed edge targets. However, these variations are temporary. Once the procedure `Simulate` has successfully terminated, the host graph is once again a configuration graph.

We say an SGP 2 program `P!` (i.e. a loop where procedure `P` is applied as long as possible) *simulates* a Turing machine M of initial configuration I if, for each two configurations S and S' of M such that $I \Rightarrow_M^* S \Rightarrow_M S'$, we have $\mathrm{enc}_0(I) \Rightarrow_{\mathtt{P!}}^* \mathrm{enc}_k(S) \Rightarrow_{\mathtt{P}}^* \mathrm{enc}_k(S')$ for some integer k.

The following theorem shows that this is indeed the case for the class of programs presented in this chapter.

Theorem 5.1 (Correctness)**.** *Let M be a Turing machine and `Sim`(M) the corresponding SGP 2 program. Then the subprocedure `Simulate!` simulates M.*

Proof. Let S and S' be configurations of M such that $I \Rightarrow_M^* S \Rightarrow_M S'$. Then some transition $\delta(q, a, x) = (p, y, X, Y)$ happened, where $p, q \in Q$, $a \in \Sigma$, $x, y \in \Gamma$, and $X, Y \in \{\mathrm{L}, \mathrm{S}, \mathrm{R}\}$. So the only changes from S to S' are that q and x have been updated to p and y, and that the input and working tape heads have been shifted into direction X and Y respectively.

We proceed by induction on the length of $I \Rightarrow_M^* S$. For the base case we only need to show $\mathrm{enc}_k(S) \Rightarrow_{\mathtt{P}}^* \mathrm{enc}_k(S')$ for some integer k. This also happens to be what we need to show for the induction step since $\mathrm{enc}_0(I) \Rightarrow^*_{\mathtt{P!}} \mathrm{enc}_k(S)$ is given by the induction hypothesis, and the existence of a large enough k by part of the proof of Theorem 5.6.

Now it remains to show that $\mathrm{enc}_k(S) \Rightarrow^*_{\mathtt{P}} \mathrm{enc}_k(S')$. The procedure `Reset` can be ignored because it is never called since we already have a large enough k.

When running `Simulate` on the graph $\mathrm{enc}_k(S)$, the first call is the rule set `Transitions`. A rule in that set that is guaranteed to be applicable is the one corresponding to the aforementioned Turing machine transition. It correctly updates the q and x to be p and y to match $\mathrm{enc}_k(S')$. It also moves the input tape head to the correct position. For the rest of this proof, we will show that the working tape head moves to the correct position.

If $Y = \mathrm{S}$, the working tape head is not moved, corresponding to $\mathrm{enc}_k(S)'$. Due to lack of an unmarked edge label, none of the `try` conditions in `Simulate` are applicable, and we terminate with $\mathrm{enc}_k(S')$.

The cases for $Y = \mathrm{L}$ and $Y = \mathrm{R}$ are analogous, so we will only argue for the former. When attempting to apply `MoveLeft`, if the target of the unmarked edge is not the leftmost node of CACHE, the rule set succeeds, the unmarked edge is in the correct position, and no other conditions of `try` statements in `Simulate` are applicable. Hence we terminate with $\mathrm{enc}_k(S')$.

If the target of the unmarked edge is the leftmost node of CACHE however, the only `try` condition that can match is `Left`. This removes the edge label and calls `PrevBlock`. So it remains to argue that `PrevBlock` turns the host graph into $\mathrm{enc}_k(S')$.

The commands `Prev; HeadRight!` correctly position the working tape head on the rightmost node of the previous block. It remains to show that `Encode` and `Decode` preserve the working tape content according to the encoding enc_k.

The procedure `Encode` contains a loop that, whenever it decrements the content of CACHE, it increments the content of the active block (represented by an outgoing dashed edge). This happens until the CACHE content reaches 0, meaning the content of CACHE is correctly stored in BLOCKSET. After moving to the new block, the procedure `Decode` is called. It contains a loop that, whenever it decrements the content of the active block, increments the content of CACHE. This terminates once the content of the active block has reached 0, meaning the content of the active block has been correctly stored in CACHE. Hence the host graph is now $\mathrm{enc}_k(S')$. □

5.3 Complexity Measures

We use the time complexity measure established in Section 3.3. For that, we have to show that our simulation forms an efficient model. Remember that we define an *efficient model* as $\mathcal{M} = \langle \mathcal{P}, \mathcal{I} \rangle$, where $\mathcal{P}$ is a set of GP 2 programs and $\mathcal{I}$ a set of GP 2 graphs such that the following two properties are satisfied within the scope of derivation sequences starting with any $\langle P, I \rangle \in \mathcal{M}$.

(1) *Constant Matching*: Every rule matches in constant time.

(2) *Critical Subprogram*: Every critical subprogram that fails is null.

Let us define the *graph space* measure of a graph as the number of nodes plus the number of edges. We do not consider the size of labels in this chapter since all labels are of constant size. This measure is uniform in that it gives unit cost to each node and edge. A discussion about this uniformity can be found at the end of Section 5.4.

Remember that for an efficient model $\mathcal{M}$, we define the time complexity of a program P of $\mathcal{M}$ as the maximum number of rule applications in terminating derivation sequences starting with P on graphs of a given size.

We show in this section that $\langle P, I \rangle$, where $P = \{\texttt{Sim}(M) \mid M$ is a Turing machine$\}$ and $I = \{\mathrm{CG} \mid \mathrm{CG}$ is a configuration graph$\}$, is an efficient model. The constant matching property is given by Theorem 3.13, which holds because all rules are fast, input graphs have bounded outdegree and a bounded number of roots, and the programs preserve this boundedness. Moreover, when extending the tape with `Restart`, outgoing dashed edges of existing block nodes are removed, and one outgoing dashed edge is added for each block node, preserving bounded outdegree. The critical subprogram property is shown by Proposition 5.3, which needs Lemma 5.2.

Lemma 5.2. *Let `P` be a critical subprogram in a graph state S. Assume either `P` cannot fail from state S, or `P` can only fail from state S due its first component being a rule or rule set call that fails to match. Then if `P` fails from state S, `P` is null.*

Proof. If `P` cannot fail from state S, the lemma is trivially satisfied. Now assume `P` fails from S due to its first component being a rule or rule set failing to match. Since it failed to match, the first component cannot have changed the host graph, and since it is the first, no other component can have changed the host graph. □

Proposition 5.3 (Critical Subprogram Property). *In `Sim`(M), given configuration graphs as inputs, every critical subprogram that fails is null.*

Proof. We will argue for each critical subprogram of `Sim` that Lemma 5.2 applies.

The conditions of all `if` and `try` statements are either rules or rule set calls, satisfying Lemma 5.2.

Let us now argue for the loop bodies. The procedures `Erase`, `RewindCache`, `HeadLeft` as well as `HeadRight` are rule sets, and `rewind_blockset` and `direct` are rules, satisfying Lemma 5.2.

`Reset`: The rule `overflow` can fail, but the other component cannot since it is a `try` statement whose branches only contain `break` which cannot fail.

`Increment`: This consists of a `try` statement, so only the branches can fail. `Finish` cannot fail because `CacheInit` is always called before this loop, providing a match. The rule `overflow` cannot fail because it is only applied when `Inc` fails and the labels of nodes in CACHE are either `0`, `1`, or `2`. The `try` statement cannot fail because its branches cannot fail.

`Decrement`: The reasoning is analogous to that in the previous paragraph.

`Decoding`: The `try` statement cannot fail because its branches cannot fail. For the remainder of this paragraph, we argue that `CacheInc` cannot fail either. `Increment!` cannot fail because it is a loop. `CacheInit` always succeeds because the target of the red edge originating from the unmarked root is unmarked because of the structure of the input graph, and because previous applications of `CacheInc` and `CacheDec` turn the only marked rooted node in this part of the graph unmarked and unrooted with `Finish`.

`Encoding`: The reasoning is analogous to that in the previous paragraph.

The body of the loop `(copy; try Undirect)!` contained in the procedure `ResetBlockset`: The rule `copy` is allowed to fail, and `try Undirect` cannot fail since it is a `try` statement.

`Simulate`: `Transition` is a rule set at the start of the body and hence allowed to fail. The rest consists of `try` statements whose branches call `PrevBlock` and `NextBlock`. `Prev` and `Next` cannot fail because we assume that BLOCKSET in the input graph is large enough to accommodate the execution of the Turing machine. `Encode` and `Decode` do not fail, which we will argue for the rest of this paragraph. `EncodeInit` and `DecodeInit` find a match because of the structure of the input graph and because previous calls of `Encode` and `Decode` leave BLOCKSET unmarked and unrooted. `Encoding!` and `Decoding!` are loops and hence cannot fail. `Update` has a match regardless of the blue root's location within BLOCKSET. □

Proposition 5.4 shows our simulation is deterministic, just like Turing machine it simulates.

Proposition 5.4 (Unique Matches). *In `Sim`(M), given configuration graphs as inputs, whenever a rule or rule set is applied, there is at most one match.*

Proof. In each rule set, if the left-hand side of two rules have the same structure, their labels are different. In `Transitions` in particular, no two rules have the same left-hand side since we consider deterministic Turing machines. Among rules with different labels but the same structure, at most one can match, namely the one that has the same labels as the host graph. Among rules with different structures and the right labels, at most one can match because they differ by an edge or node that is unique in host graphs. Furthermore, it is easy to check that each rule can only have at most one match in the host graph. Each rule contains a root with a unique combination of mark and label. And for each node, outgoing edges can be distinguished in the same way. □

5.4 Results on Time and Space Complexity

In this section, we present theorems on the time and space complexities of the simulation.

Theorem 5.5 (Time Complexity). *Every Turing machine M of time complexity $t(n)$ and of space complexity* $\mathrm{O}(s(n) \log s(n))$ *is simulated by* `Sim`(M) *in time* $\mathrm{O}(t^2(n))$*, where n is the size of the input.*

Proof. Given the discussion in Section 5.3, we can assign unit time to rule and rule set calls and argue time complexity to be the number of such calls.

First, we will show that simulating one step of M (not counting restarting the simulation) takes $\mathrm{sim}(n) = \mathrm{O}(s(n))$ time. We consider the size of CACHE from the final simulation since it provides an upper bound. The only sources of non-constant time are `PrevBlock` and `NextBlock`. Their complexity is the worst of the loops `Encoding!`, `Decoding!`, `HeadLeft!`, and `HeadRight!`. The latter two simply traverse CACHE, which takes $\mathrm{O}(s(n))$ time. The former two decrement/increment CACHE as a ternary counter. Their time complexity is proportional to the number of digit operations it takes to decrement the counter from the number $s(n)$ all the way to 0. The rightmost digit is modified $s(n)$ times, the next one $\frac{1}{3}s(n)$ times, the one after $\frac{1}{3} \cdot \frac{1}{3}s(n)$ times, and so on. So the total number of digit operations is $\sum_{k=0}^{\log s(n)} s(n) \left(\frac{1}{3}\right)^k$. Using properties of the geometric series, one can see that this is $\mathrm{O}(s(n))$.

In this paragraph, we will show that resetting the simulation and extending the tape takes $r(n) = \mathrm{O}(s(n))$ time. Consider the loops of `Restart`. Both `RewindCache!` and `Erase!` traverse CACHE and hence take time $\mathrm{O}(\log s(n))$. The loop `(copy; try Undirect)!` in `ResetBlockset` traverses

BLOCKSET and thus takes $\mathrm{O}(s(n))$ time. And `direct!` traverses the extended BLOCKSET in $\mathrm{O}(3 \cdot s(n)) = \mathrm{O}(s(n))$ time.

Next, we will show that the number of times the simulation is restarted is $l(n) = \mathrm{O}(\log s(n))$. The final size of the tape of M is $\mathrm{O}(s(n) \log s(n))$. Since the number of represented squares is tripled in each step, the number of steps is $\mathrm{O}(\log(s(n) \log s(n)))$. Using the formula for the logarithm of a product, one can simplify this to $\mathrm{O}(\log s(n))$.

The total time complexity can be bounded by $\mathrm{reset}(n) + \mathrm{simulation}(n)$, where $\mathrm{reset}(n) = l(n) \cdot r(n)$ is the total cost of all resets, and $\mathrm{simulation}(n) = l(n) \cdot t(n) \cdot \mathrm{sim}(n)$ the total cost of simulating M across all resets. Using results from previous paragraphs, we get $\mathrm{reset}(n) = \mathrm{O}(s(n) \log s(n))$ and $\mathrm{simulation}(n) = \mathrm{O}(t(n) \cdot s(n) \log s(n))$. Since space complexity $s(n) \log s(n)$ can be bounded by time complexity $t(n)$, the entire simulation takes $\mathrm{O}(t^2(n))$ time. □

Theorem 5.6 (Space Complexity)**.** *Every Turing machine M of space complexity $\mathrm{O}(s(n) \log s(n))$ is simulated by `Sim`(M) in graph space $O\left(s(n)\right)$, where n is the size of the input.*

Proof. During the execution of `Sim`(M), nodes and edges are only created by `setup` and `Restart`, and none are ever deleted. The numbers of nodes and edges only differ by a constant factor since the outdegree is bounded, so we will argue for space complexity using number of nodes only.

Initially, after application of `setup`, BLOCKSET has $b(n) = 3^2$ nodes, and CACHE $c(n) = 2$. Then, each application of `Restart` adds one to $c(n)$ and triples $b(n)$. So after k iterations, we have $b(n) = 3^{2+k}$ and $c(n) = 2 + k$.

The Turing machine needs $S(n) = \mathrm{O}(s(n) \log s(n))$ tape squares. This means that there are positive integers n_0 and c such that $S(n) \leq c\, s(n) \log s(n)$ for all $n \geq n_0$. So for all n we can say $S(n) \leq c\, s(n) \log s(n) + m$, where $m = \max_{n \in \{0,\ldots,n_0\}} S(n)$, a constant.

Assume `Restart` is called $k = \log s(n) - 2 + d$ many times, where $d = \max(m, \log c)$. For that value of k, we have $c(n)b(n) = 3^d s(n) \log s(n) + 3^d s(n)\, d$. By definition we have $d \geq m$ and $3^d \geq c$. Furthermore, we have $3^d s(n) \geq 1$. Hence we get $c(n) \cdot b(n) \geq c\, s(n) \log s(n) + m \geq S(n)$. So for this value of k, the graph can store enough tape squares to execute M.

With the aforementioned value of k, the number of nodes in this graph is $c(n) + b(n) = 3^d s(n) + \log s(n) + d$, which is $\mathrm{O}(\log s(n) + s(n)) = \mathrm{O}(s(n))$.

Hence the number of nodes of the graph that is created is bounded by $c(n) + b(n)$, and hence in $\mathrm{O}(s(n))$. □

One might wonder why this space compression is not possible on random access machines. GP 2 does have a C implementation after all. The reason for this is related to how graphs are represented in random access machines (RAMs), and how much space that takes. Graph edges are usually implemented as pointers. However the size of pointer addresses grows logarithmically with the number of nodes, since these addresses are usually stored as binary numbers. So in the context of RAMs, it does not seem very accurate to assign unit cost to edges. To take this into account, one can use *logarithmic space*, in which graphs of $s(n)$ nodes are assigned a cost of $s(n) \log s(n)$.

If we use logarithmic space on our model, space compression is nullified since the simulation then has the same asymptotic space complexity as the machine it simulates. This puts into question whether uniform or logarithmic space should be used, which is discussed by van Emde Boas [85]. One may want to charge more than unit space since in RAMs, edges are represented by pointers whose size grows with the number of nodes. A related issue can be found in the time measure of RAM models when programs have to deal with large integers.

5.5 Related Work: Invariance Thesis

Van Emde-Boas shows in [85] the storage modification machines and Kolmogorov-Uspenskii machines can simulate Turing machines in less space, with only a quadratic time overhead. Luguinbuhl and Loui improve on this in [52] by showing this can be done in real-time, using a complete binary tree that encodes the contents of a tape section. This tape section can be seen as a binary number. Depending on whether a digit is 0 or 1, a left or right child is chosen. In this representation, the leaves correspond to all possible tape section contents. Edges towards these leaves are used to store these tape section contents. Having a tree structure allows changing the content of a tape section to happen in more efficient time since it is only a matter of traversing the depth of the tree twice in the worst case. This representation is then implemented into a machine that uses concurrent processes that update tape sections in the background and allow for a real-time simulation.

As described in [77, 86], the *invariance thesis* is as follows. There is a standard class of machine models (including Turing machines and RAMs) that simulate each other with a constant space overhead and a polynomial time overhead.

The space compression results show that the invariance thesis does not hold for uniform space measures (unit cost for edges). One has to either adopt the more cumbersome logarithmic space measure (size of pointer addresses representing edges taken into account) or accept that the invariance thesis only holds for a limited class of machine models.

6 Conclusions and Future Work

We end this thesis by drawing a few conclusions from it, and by giving some interesting items of future work.

6.1 Conclusions

This thesis shows that rule-based graph programs, specifically in GP 2, can be efficient in both time and space. We provide a small-step semantics and a formal complexity framework that allows us to give formal arguments about the complexity of these programs. We also present a collection of efficient graph programs that achieve the same asymptotic complexity as equivalent imperative implementations, using linear-time depth-first search and rules that can be matched in constant time under mild conditions. Finally, we demonstrate that GP 2 shares the space compression property with storage modification machines and Kolmogorov-Uspenskii machines, meaning that a class of programs in a subset of GP 2 can simulate Turing machines using less space and only quadratic time overhead. Overall, this thesis contributes to the theoretical understanding of rule-based graph programs and their time and space complexity.

When comparing fast GP 2 programs (like finding minimum spanning trees in Section 4.3) to ones that have not been optimised for time complexity (like the ones in [59]), it is apparent that there is a trade-off. If one wants to make a fast GP 2 program, one often sacrifices readability, by which we mean it takes more effort to intuit the program's behaviour. This is because the program needs to create and maintain structures on top of host graphs, which adds complication. However, it is not the case for all algorithms. Connectedness checking in Section 4.1 and 2-colouring in [10] are easy to read. So are the ones based on reducing graph size in [19] since they do not rely on additional structures to be efficient.

6.2 Future Work

In this section, we bring up interesting avenues for future research. This includes improvements to the GP 2 language and compiler, performance measuring, and complexity framework, as well as finding applications for GP 2.

6.2.1 Changing GP 2 or Its Implementation to Enhance Efficiency

There are multiple ways in which the implementation of GP 2 and the language itself could evolve to enable faster programs in more cases.

Limiting search space for edges. In the current compiler, the adjacent edges of a node are stored in distinct lists based on whether they are outgoing or incoming. One could additionally separate them based on marks. With the right programming approach, this circumvents the need to restrict ourselves to graphs of bounded degree or even outdegree. Assigning a different mark to previously traversed edges can limit the search space for subsequent edge traversals and improve the program's efficiency. This would enable a linear-time depth-first search on graphs of unbounded degree for instance, making the bounded degree restriction on the case studies in this thesis superfluous.

Limiting search space for nodes. Similarly to edges, nodes can also be separated based on marks. Currently, nodes are accessed through a linked list of pointers. However, by creating separate linked lists for each mark, the search space for finding a node that is not adjacent to an already matched edge can be reduced. Essentially, nodes of each mark can be accessed as if they were roots, which is especially beneficial when moving on to another connected component. Assigning a distinct mark to traversed nodes allows for highly efficient discovery of untraversed nodes. This approach enables a linear-time depth-first search on graphs with an unbounded number of connected components for instance. One could implement the 2-colouring algorithm to run on graphs with multiple connected components in linear time this way.

Adding constructs for efficient graph traversal. Another approach is to extend the language with built-in functions that take advantage of internal data structures for greater efficiency. This would allow the programmer to call these functions without needing to delve into the implementation details. For example, one possible built-in function could serve as an iterator over all nodes, taking linear time to execute. During each iteration, a different node would act as the root to which rules can be applied. Another function could add a root to each connected component in linear time, which would allow for algorithms to be more efficient on graphs with an unbounded number of connected components. This would simplify programs such as the one finding a minimum spanning tree by eliminating the need for depth-first search while making it efficient on input graphs with an unbounded number of connected components.

Extending to parallel computing. One could explore adapting GP 2 for parallel computation, which could allow multiple procedures to run concurrently and potentially speed up programs. For example, as shown for storage modification machines and Kolmogorov-Uspenskii machines in Luginbuhl and Loui's paper [52], parallel computation can enable real-time simulation of Turing machines using less space. Additionally, parallelisation could be utilised to speed up matching by constructing multiple potential matches at the same time. It is worth noting, however, that this approach would offer empirical efficiency gains and would not impact the asymptotic complexity of rule matching.

6.2.2 Exploring Techniques for Measuring Performance

So far, time measurements for GP 2 programs have been conducted on various graph classes, but could be broadened to include any valid input graph for the program being evaluated. Achieving this would require a method for generating random graphs of a specific size.

One possible approach at the implementation level is to create a tool that collects efficiency data during the execution of a GP 2 program. The compiled C program could keep track of the number of times each rule is applied and measure the time taken in both worst and average case. For measuring space, the tool could monitor changes in the number of nodes, edges, and label sizes throughout the program's execution. Recording the amount of memory used during execution would provide empirical space measurements.

6.2.3 Extending the Complexity Framework

The formal framework for expressing the complexity of GP 2 programs could be expanded to include arbitrary terminating programs and inputs. Such an extension would require accounting for the time needed for rule application, which can be very inefficient.

Recent efforts in formalising GP 2 in a proof assistant [78] present an opportunity to automate complexity proofs. Another promising approach to automate complexity analysis is to follow the approach proposed in [54], which is about term rewriting, i.e. similar to graph transformation. The main advantage of this approach is that programs do not need to terminate for the analysis to take place.

6.2.4 Finding Applications for GP 2

With the improved understanding of how to make GP 2 programs efficient, it is worth exploring potential applications of the graph programming language.

Extension to string and term rewriting. The fact that GP 2 applies its rules to graphs is not a fundamental aspect of its underlying theory, and the same semantics could be adapted for string rewriting [15] or term rewriting [7]. The established theoretical results for graph transformation could be leveraged, and a careful implementation could allow for similar approaches to complexity analysis.

Real-world applications. One could apply GP 2 to problems that are related to graph transformation, like modelling social networks [13], chemical reactions [3], or epidemics [90]. Another potential area of application is in graph databases. GP 2 could be used to implement something similar to the Gremlin graph traversal query language [66]. Queries could be expressed using rules, or even using depth-first search to find the queried graph pattern.

Bibliography

[1] Aditya Agrawal, Gabor Karsai, Sandeep Neema, Feng Shi & Attila Vizhanyo (2006): *The Design of a Language for Model Transformations. Software and System Modeling* 5(3), pp. 261–288, doi:10.1007/s10270-006-0027-7.

[2] Alfred V. Aho, John E. Hopcroft & Jeffrey D. Ullman (1974): *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

[3] Jakob Lykke Andersen, Christoph Flamm, Daniel Merkle & Peter F. Stadler (2017): *Chemical Graph Transformation with Stereo-Information.* In Juan de Lara & Detlef Plump, editors: *Proc. 10th International Conference on Graph Transformation (ICGT 2017)*, *Lecture Notes in Computer Science* 10373, Springer, pp. 54–69, doi:10.1007/978-3-319-61470-0_4.

[4] Krzysztof R. Apt (1984): *Ten Years of Hoare's Logic: A Survey. Part II: Nondeterminism. Theoretical Computer Science* 28, pp. 83–109, doi:10.1016/0304-3975(83)90066-X.

[5] Krzysztof R. Apt (1996): *From Logic Programming to Prolog.* Prentice Hall.

[6] Nina Aschenbrenner & Leif Geiger (2008): *Transforming Scene Graphs Using Triple Graph Grammars – A Practice Report.* In A. Schürr, M. Nagl & A. Zündorf, editors: *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, *Lecture Notes in Computer Science* 5088, Springer, pp. 32–43, doi:10.1007/978-3-540-89020-1_3.

[7] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That.* Cambridge University Press, doi:10.1017/CBO9781139172752.

[8] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language.* Ph.D. thesis, Department of Computer Science, University of York. Available at `http://etheses.whiterose.ac.uk/12586/`.

[9] Christopher Bak, Glyn Faulkner, Detlef Plump & Colin Runciman (2015): *A Reference Interpreter for the Graph Programming Language GP 2. Electronic Proceedings in Theoretical Computer Science* 181, pp. 48–64, doi:10.4204/eptcs.181.4.

[10] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs.* In: *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54, p. 12, doi:10.14279/tuj.eceasst.54.780.

[11] Christopher Bak & Detlef Plump (2016): *Compiling Graph Programs to C.* In Rachid Echahed & Mark Minas, editors: *Graph Transformation*, *Lecture Notes in Computer Science (LNCS)* 9761, Springer, pp. 102–117, doi:10.1007/978-3-319-40530-8_7.

[12] Cüneyt F. Bazlamaçcı & Khalil S. Hindi (2001): *Minimum-weight spanning tree algorithms A survey and empirical study. Computers & Operations Research* 28(8), pp. 767–785, doi:10.1016/S0305-0548(00)00007-1.

[13] Nicolas Behr, Bello Shehu Bello, Sebastian Ehmes & Reiko Heckel (2021): *Stochastic Graph Transformation For Social Network Modeling.* In Berthold Hoffmann & Mark Minas, editors: *Proc. 12th International Workshop on Graph Computational Models (GCM 2021)*, *EPTCS* 350, pp. 35–50, doi:10.4204/EPTCS.350.3. Revised Selected Papers.

[14] Gábor Bergmann, Ákos Horváth, István Ráth & Dániel Varró (2008): *A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation.* In: *Proc. International Conference on Graph Transformation (ICGT 2008)*, *Lecture Notes in Computer Science* 5214, Springer, pp. 396–410, doi:10.1007/978-3-540-87405-8_27.

[15] Ronald V. Book & Friedrich Otto (1993): *String-Rewriting Systems.* Springer, doi:10.1007/978-1-4613-9771-7.

[16] Horst Bunke, Thomas Glauser & T.-H. Tran (1991): *An Efficient Implementation of Graph Grammars Based on the RETE-Matching Algorithm.* In: *Proc. Graph Grammars and Their Application to Computer Science and Biology*, *LNCS* 532, Springer-Verlag, pp. 174–189, doi:10.1007/BFb0017389.

[17] Graham Campbell (2019): *Efficient Graph Rewriting.* Bachelor thesis, Department of Computer Science, University of York, doi:10.48550/arXiv.1906.05170.

[18] Graham Campbell, Brian Courtehoute & Detlef Plump (2019): *Linear-Time Graph Algorithms in GP 2.* In Markus Roggenbach & Ana Sokolova, editors: *Proc. 8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, *Leibniz International Proceedings in Informatics (LIPICS)* 139, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, pp. 16:1–16:23, doi:10.4230/LIPIcs.CALCO.2019.16.

[19] Graham Campbell, Brian Courtehoute & Detlef Plump (2022): *Fast Rule-Based Graph Programs. Science of Computer Programming* 214, doi:10.1016/j.scico.2021.102727.

[20] Graham Campbell, Jack Romo & Detlef Plump (2020): *The Improved GP 2 Compiler.* In: *Pre-Proc. 11th International Workshop on Graph Computation Models (GCM 2020)*, pp. 206–217. Available at `https://eprints.whiterose.ac.uk/176054/`.

[21] Graham Campbell, Jack Romo & Detlef Plump (2020): *Improving the GP 2 Compiler*, doi:10.48550/arXiv.2002.02914.

[22] William Clinger (1984): *The Scheme 311 Compiler an Exercise in Denotational Semantics.* In: *Proc. ACM Symposium on LISP and Functional Programming (LFP 1984)*, pp. 356–364, doi:10.1145/800055.802052.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 3rd edition. The MIT Press.

[24] Brian Courtehoute & Detlef Plump (2020): *A Fast Graph Program for Computing Minimum Spanning Trees.* In: *Proc. 11th International Workshop on Graph Computation Models (GCM 2020)*, *Electronic Proceedings in Theoretical Computer Science* 330, pp. 163–180, doi:10.4204/eptcs.330.10. Revised Selected Papers.

[25] Brian Courtehoute & Detlef Plump (2021): *A Small-Step Operational Semantics for GP 2.* In: *Proc. 12th International Workshop on Graph Computation Models (GCM 2021)*, *Electronic Proceedings in Theoretical Computer Science* 350, pp. 89–110, doi:10.4204/eptcs.350.6. Revised Selected Papers.

[26] Brian Courtehoute & Detlef Plump (2022): *Time and Space Measures for a Complete Graph Computation Model.* In: *Proc. 13th International Workshop on Graph Computation Models (GCM 2022), Electronic Proceedings in Theoretical Computer Science* 374, pp. 23–44, doi:10.4204/eptcs.374.4. Revised Selected Papers.

[27] Edsger W. Dijkstra (1968): *Letters to the Editor: Go To Statement Considered Harmful. Communications of the ACM* 11(3), pp. 147–148, doi:10.1145/362929.362947.

[28] Edsger W. Dijkstra (1976): *A Discipline of Programming*, 1st edition. Prentice Hall.

[29] Mike Dodds (2008): *Graph Transformation and Pointer Structures.* Ph.D. thesis, Department of Computer Science, University of York. Available at `https://www.cs.york.ac.uk/plasma/publications/pdf/DoddsThesis.08.pdf`.

[30] Heiko Dörr (1995): *Efficient Graph Rewriting and its Implementation. Lecture Notes in Computer Science* 922, Springer, doi:10.1007/BFb0031909.

[31] Frank Drewes (1993): *NP-completeness of k-connected hyperedge-replacement languages of order k. Information Processing Letters* 45(2), pp. 89–94, doi:10.1016/0020-0190(93)90221-T.

[32] Frank Drewes, Berthold Hoffmann & Mark Minas (2020): *Graph Parsing as Graph Transformation.* In Fabio Gadducci & Timo Kehrer, editors: *Proc. 13th International Conference on Graph Transformation (ICGT 2020), Lecture Notes in Computer Science* 12150, Springer, pp. 221–238, doi:10.1007/978-3-030-51372-6_13.

[33] Frank Drewes, Berthold Hoffmann & Mark Minas (2021): *Rule-Based Top-Down Parsing for Acyclic Contextual Hyperedge Replacement Grammars.* In Fabio Gadducci & Timo Kehrer, editors: *Proc. 14th International Conference on Graph Transformation (ICGT 2021), Lecture Notes in Computer Science* 12741, Springer, pp. 164–184, doi:10.1007/978-3-030-78946-6_9.

[34] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/3-540-31188-2.

[35] Hartmut Ehrig, Claudia Ermel, Ulrike Golas & Frank Hermann (2015): *Graph and Model Transformation.* Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-47980-3.

[36] Maribel Fernández, Hélène Kirchner & Bruno Pinaud (2019): *Strategic Port Graph Rewriting: an Interactive Modelling Framework. Mathematical Structures in Computer Science* 29(5), pp. 615–662, doi:10.1017/S0960129518000270.

[37] Wan Fokkink & Thuy Vu (2003): *Structural Operational Semantics and Bounded Nondeterminism. Acta Informatica* 39, pp. 501–516, doi:10.1007/s00236-003-0111-1.

[38] Charles L. Forgy (1982): *Rete: A Fast Algorithm For the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence* 19(1), pp. 17–37, doi:10.1016/0004-3702(82)90020-0.

[39] Michael Garey & David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[40] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack & Adam Szalkowski (2006): *GrGen: A Fast SPO-Based Graph Rewriting Tool.* In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro & Grzegorz Rozenberg, editors: *Proc. International Conference on Graph Transformation (ICGT 2006)*, Lecture Notes in Computer Science, Springer, pp. 383–397, doi:10.1007/11841883_27.

[41] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon & Maria Zimakova (2012): *Modelling and Analysis Using GROOVE. International Journal on Software Tools for Technology Transfer* 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[42] Annegret Habel & Detlef Plump (2001): *Computational Completeness of Programming Languages Based on Graph Transformation.* In: *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001), LNCS* 2030, Springer, pp. 230–245, doi:10.1007/3-540-45315-6_15.

[43] Annegret Habel & Detlef Plump (2002): *Relabelling in Graph Transformation.* In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski & Grzegorz Rozenberg, editors: *Graph Transformation*, *Lecture Notes in Computer Science* 2505, Springer, pp. 135–147, doi:10.1007/3-540-45832-8_12.

[44] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming. Communications of the ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[45] Ákos Hórvath, Gergely Varró & Dániel Varró (2007): *Generic Search Plans for Matching Advanced Graph Patterns.* In: *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, *Electronic Communications of the EASST* 6, doi:10.14279/tuj.eceasst.6.49.

[46] Ivaylo Hristakiev & Detlef Plump (2018): *Checking Graph Programs for Confluence.* In: *Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers*, *Lecture Notes in Computer Science* 10748, Springer, pp. 92–108, doi:10.1007/978-3-319-74730-9_8.

[47] Edgar Jakumeit, Sebastian Buchwald & Moritz Kroll (2010): *GrGen.NET - The Expressive, Convenient and Fast Graph Rewrite system. International Journal on Software Tools for Technology Transfer* 12(3–4), pp. 263–271, doi:10.1007/s10009-010-0148-8.

[48] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Arend Rensink, Louis Rose, Sebastian Wätzoldt, Markus Lepper & Steffen Mazanek (2014): *A Survey and Comparison of Transformation Tools Based on the Transformation Tool Contest. Science of Computer Programming* 85, pp. 41–99, doi:10.1016/j.scico.2013.10.009.

[49] Andrey N. Kolmogorov & Vladimir A. Uspenskii (217–245): *On the Definition of an Algorithm. American Mathematical Society Translations: Series 2* 29, pp. 217–245, doi:10.1090/trans2/029.

[50] Dénes Kőnig (1927): *Über eine Schlussweise aus dem Endlichen ins Unendliche. Acta Sci. Math.(Szeged)* 3(2-3), pp. 121–130. Available at `https://acta.bibl.u-szeged.hu/13338/`.

[51] Klaus-Jörn Lange & Emo Welzl (1987): *String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. Discrete Applied Mathematics* 16(1), pp. 17–30, doi:10.1016/0166-218X(87)90051-5.

[52] David R. Luginbuhl & Michael C. Loui (1993): *Hierarchies and Space Measures for Pointer Machines. Information and Computation* 104(2), pp. 253–270, doi:10.1006/inco.1993.1032.

[53] Greg Manning & Detlef Plump (2008): *The GP Programming System.* In: *Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)*, pp. 1–13, doi:10.14279/tuj.eceasst.10.150.

[54] Georg Moser & Manuel Schneckenreither (2020): *Automated Amortised Resource Analysis for Term Rewrite Systems. Science of Computer Programming* 185, doi:10.1016/j.scico.2019.102306.

[55] Hanne Riis Nielson & Flemming Nielson (2007): *Semantics with Applications: An Appetizer.* Springer-Verlag London, doi:10.1007/978-1-84628-692-6.

[56] Romain Pascual (2022): *Inference of Graph Transformation Rules for the Design of Geometric Modeling Operations.* Ph.D. thesis, Université Paris-Saclay. Available at `https://romainpascual.fr/uploads/phdthesis.pdf`.

[57] Gordon D. Plotkin (2004): *A Structural Approach to Operational Semantics. Journal of Logic and Algebraic Programming* 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001.

[58] Detlef Plump (2012): *The Design of GP 2.* In: *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *Electronic Proceedings in Theoretical Computer Science* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[59] Detlef Plump (2016): *Reasoning About Graph Programs.* In: *Proc. Computing with Terms and Graphs (TERMGRAPH 2016)*, *Electronic Proceedings in Theoretical Computer Science* 225, pp. 35–44, doi:10.4204/eptcs.225.6.

[60] Detlef Plump (2017): *From Imperative to Rule-based Graph Programs. Journal of Logical and Algebraic Methods in Programming* 88, pp. 154–173, doi:10.1016/j.jlamp.2016.12.001.

[61] Detlef Plump, Amritesh Singh & Robin Suri (2011): *Minimizing Finite Automata with Graph Programs.* In: *Proc. 3rd International Workshop*

on Graph Computation Models (GCM 2010), *Electronic Communications of the EASST* 39, doi:10.14279/tuj.eceasst.39.658. Revised Selected Papers.

[62] Detlef Plump & Sandra Steinert (2004): *Towards Graph Programs for Graph Algorithms.* In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce & Grzegorz Rozenberg, editors: *Graph Transformations*, *Lecture Notes in Computer Science* 3256, Springer, pp. 128–143, doi:10.1007/978-3-540-30203-2_11.

[63] Christopher M. Poskitt (2013): *Verification of Graph Programs.* Ph.D. thesis, Department of Computer Science, University of York. Available at `http://etheses.whiterose.ac.uk/4700/`.

[64] Christopher M. Poskitt & Detlef Plump (2023): *Monadic Second-Order Incorrectness Logic for GP 2. Journal of Logical and Algebraic Methods in Programming* 130, doi:10.1016/j.jlamp.2022.100825.

[65] John C. Reynolds (1998): *Theories of Programming Languages.* Cambridge University Press, doi:10.1017/cbo9780511626364.

[66] Marko A. Rodriguez (2015): *The Gremlin Graph Traversal Machine and Language.* In: *Proc. 15th Symposium on Database Programming Languages (DBPL 2015)*, Association for Computing Machinery (ACM), pp. 1–10, doi:10.1145/2815072.2815073.

[67] Olga Runge, Claudia Ermel & Gabriele Taentzer (2012): *AGG 2.0 — New Features for Specifying and Analyzing Algebraic Graph Transformations.* In: *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, *Lecture Notes in Computer Science* 7233, Springer, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[68] Florian Schöler & Volker Steinhage (2012): *Towards an Automated 3D Reconstruction of Plant Architecture.* In Andy Schürr, Dániel Varró & Gergely Varró, editors: *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, *Lecture Notes in Computer Science* 7233, Springer, pp. 51–64, doi:10.1007/978-3-642-34176-2_6.

[69] Arnold Schönhage (1980): *Storage Modification Machines. SIAM Journal on Computing* 9(3), pp. 490–508, doi:10.1137/0209036.

[70] Andreas Schürr (1991): *Operationales Spezifizieren mit programmierten Graphersetzungssystemen - formale Definitionen, Anwendungsbeispiele*

und Werkzeugunterstützung. DUV Informatik, Deutscher Universitätsverlag, doi:10.1007/978-3-663-14577-6.

[71] Andy Schürr (1997): *Programmed Graph Replacement Systems*, chapter 7, pp. 479–546. World Scientific, doi:10.1142/9789812384720_0007.

[72] Andy Schürr, Andreas J. Winter & Albert Zündorf (1999): *The PROGRES Approach: Language and Environment*, chapter 13, pp. 487–550. World Scientific, doi:10.1142/9789812815149_0013.

[73] Roger S. Scowen (1993): *Extended BNF - A Generic Base Standard.* In: *Software Engineering Standards Symposium (SESS 1993)*, pp. 25–34. Available at `https://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf`.

[74] Robert Sedgewick (1997): *Algorithms in C: Parts 1-4, Fundamentals, Data Structures, Sorting, and Searching*, 3rd edition. Addison-Wesley.

[75] Robert Sedgewick (2001): *Algorithms in C, Part 5: Graph Algorithms*, 3rd edition. Addison-Wesley.

[76] Steven S. Skiena (2008): *The Algorithm Design Manual*, 2nd edition. Springer, doi:10.1007/978-1-84800-070-4.

[77] Cees Slot & Peter van Emde Boas (1988): *The Problem of Space Invariance for Sequential Machines. Information and Computation* 77(2), pp. 93–122, doi:10.1016/0890-5401(88)90052-1.

[78] Robert Söldner & Detlef Plump (2022): *Towards Mechanised Proofs in Double-Pushout Graph Transformation.* In: *Proc. 13th International Workshop on Graph Computation Models (GCM 2022)*, *Electronic Proceedings in Theoretical Computer Science* 374, pp. 59–75, doi:10.4204/eptcs.374.6. Revised Selected Papers.

[79] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf & Matthias Tichy (2017): *Henshin: A Usability-Focused Framework for EMF Model Transformation Development.* In: *Proc. 10th International Conference on Graph Transformation (ICGT 2017)*, *Lecture Notes in Computer Science* 10373, Springer, pp. 196–208, doi:10.1007/978-3-319-61470-0_12.

[80] Janusz Szuba, Agnieszka Ozimek & Andy Schürr (2004): *On Graphs in Conceptual Engineering Design.* In John L. Pfaltz, Manfred Nagl & Boris Böhlen, editors: *Applications of Graph Transformations with Industrial*

Relevance (AGTIVE 2003), *Lecture Notes in Computer Science* 3062, Springer, pp. 75–89, doi:10.1007/978-3-540-25959-6_6.

[81] Harald Søndergaard & Peter Sestoft (1992): *Non-determinism in Functional Languages*. *The Computer Journal* 35(5), pp. 514–523, doi:10.1093/comjnl/35.5.514.

[82] Gabriele Taentzer (1996): *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-based Systems*. Ph.D. thesis, Technische Universität Berlin. Available at `https://www.user.tu-berlin.de/lieske/tfs/Dissertationen/GabiTaentzer.pdf`.

[83] 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 & Tamás Vajk (2008): *Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools*. In Andy Schürr, Manfred Nagl & Albert Zündorf, editors: *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, *Lecture Notes in Computer Science* 5088, Springer, pp. 514–539, doi:10.1007/978-3-540-89020-1_35.

[84] Julian R. Ullmann (1976): *An Algorithm for Subgraph Isomorphism*. *Journal of the ACM* 23(1), pp. 31–42, doi:10.1145/321921.321925.

[85] Peter van Emde Boas (1989): *Space Measures For Storage Modification Machines*. *Information Processing Letters* 30(2), pp. 103–110, doi:10.1016/0020-0190(89)90117-8.

[86] Peter van Emde Boas (1990): *Machine Models and Simulation*. In Jan van Leeuwen, editor: *Handbook of Theoretical Computer Science Algorithms and Complexity*, chapter 1, volume A, Elsevier, pp. 1–66, doi:10.1016/b978-0-444-88071-0.50006-0.

[87] Gergely Varró, Katalin Friedl & Dániel Varró (2006): *Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans*. In: *Proc. Workshop on Graph and Model Transformation (GraMoT 2005)*, *Electronic Notes in Theoretical Computer Science* 152, Elsevier, pp. 191–205, doi:10.1016/j.entcs.2005.10.025.

[88] Gergely Varró, Katalin Friedl & Dániel Varró (2006): *Implementing a Graph Transformation Engine in Relational Databases*. *Software & Systems Modeling* 5(3), pp. 313–341, doi:10.1007/s10270-006-0015-y.

[89] Gergely Varró, Andy Schürr & Dániel Varró (2005): *Benchmarking for Graph Transformation.* In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005)*, pp. 79–88, doi:10.1109/VLHCC.2005.23.

[90] William Waites, Matteo Cavaliere, David Manheim, Jasmina Panovska-Griffiths & Vincent Danos (2021): *Rule-based Epidemic Models. Journal of Theoretical Biology* 530, doi:10.1016/j.jtbi.2021.110851.

[91] David A. Watt (1988): *An Action Semantics of Standard ML.* In M. Main, A. Melton, M. Mislove & D. Schmidt, editors: *Mathematical Foundations of Programming Language Semantics (MFPS 1987)*, *Lecture Notes in Computer Science* 298, Springer, pp. 572–598, doi:10.1007/3-540-19020-1_30.

[92] Jens H. Weber-Jahnke (2008): *Modelling of Longitudinal Information Systems with Graph Grammars.* In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007)*, *Lecture Notes in Computer Science* 5088, Springer, pp. 59–65, doi:10.1007/978-3-540-89020-1_5.

[93] Robin J. Wilson (1972): *Introduction to Graph Theory*, 5th edition. Prentice Hall.

[94] Gia Wulandari (2021): *Verification of Graph Programs With Monadic Second-order Logic.* Ph.D. thesis, Department of Computer Science, University of York. Available at `https://etheses.whiterose.ac.uk/29370/`.

[95] Gia Wulandari & Detlef Plump (2021): *Verifying Graph Programs with Monadic Second-Order Logic.* In: *Proc. 14th International Conference on Graph Transformation (ICGT 2021)*, *Lecture Notes in Computer Science* 12741, Springer, pp. 240–261, doi:10.1007/978-3-030-78946-6_13.

[96] Paul Ziemann, Karsten Hölscher & Martin Gogolla (2005): *From UML Models to Graph Transformation Systems.* In: *Proc. Workshop on Visual Languages and Formal Methods (VLFM 2004)*, *Electronic Notes in Theoretical Computer Science* 127, pp. 17–33, doi:10.1016/j.entcs.2004.10.025.