Efficient Recognition of Graph Languages

Project Report

Graham Campbell\textsuperscript{1} and Detlef Plump

Department of Computer Science
University of York, United Kingdom

September 2019

\textsuperscript{1}Supported by a Vacation Internship from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.
Abstract

Graph transformation is the rule-based modification of graphs, and is a discipline dating back to the 1970s. In general, to match the left-hand graph of a fixed rule within a host graph requires polynomial time, but to improve matching performance, Dörr proposed to equip rules and host graphs with distinguished root nodes. This model was implemented by Plump and Bak, but unfortunately, such rules are not invertible. We address this problem by defining rootedness using a partial function into a two-point set rather than pointing graphs with root nodes, meaning derivations are natural double pushouts. Moreover, we give a sufficient condition on rules to give constant time rule application on graphs of bounded degree, and that, the graph class of trees can be recognised in linear time, given an input graph of bounded degree. Finally, we define a new notion of confluence up to garbage and non-garbage critical pairs, showing it is sufficient to require strong joinability of only the non-garbage critical pairs to establish confluence up to garbage. Finally, this new result, presented for conventional graph transformation systems, can be lifted to our rooted setting by encoding node labels and rootedness as looped edges.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Theorems</td>
<td>ix</td>
</tr>
<tr>
<td>Executive Summary</td>
<td>xi</td>
</tr>
<tr>
<td>Preface</td>
<td>xv</td>
</tr>
</tbody>
</table>

## 1 Theoretical Background

1.1 Graphs and Morphisms .......................... 1
1.2 Graph Transformation ............................. 2
1.3 Adding Relabelling ............................... 3
1.4 Rooted Graph Transformation ...................... 4
1.5 Graph Languages .................... 5
1.6 Abstract Reduction Systems ...................... 6
1.7 Graph Programming Languages ...................... 7

## 2 A New Theory of Rooted Graph Transformation

2.1 Graphs and Morphisms .......................... 10
2.2 Rules and Derivations .......................... 12
2.3 Foundational Theorems ........................... 13
2.4 Equivalence of Rules ............................ 16
2.5 Graph Transformation Systems ................... 17
2.6 Equivalence of GT Systems ....................... 19
## Fast Language Recognition

- **3.1 Complexity Theorems**
- **3.2 Recognising Trees in Linear Time**
- **3.3 Recognising Full Binary Trees**
- **3.4 Recognising Trees in GP2**

## Confluence Analysis I

- **4.1 Definitions and Closedness**
- **4.2 Non-Garbage Critical Pairs**
- **4.3 Extended Flow Diagrams**

## Confluence Analysis II

- **5.1 Encoding PLRGs**
- **5.2 Local Church-Rosser and Independence**
- **5.3 Confluence up to Garbage**
- **5.4 Tree Recognition Revisited**

## Conclusion

- **6.1 Evaluation**
- **6.2 Future Work**

## Basic Mathematical Notions

- **A.1 Sets I**
- **A.2 Functions**
- **A.3 Binary Relations**
- **A.4 Orders**
- **A.5 Sets II**

## Category Theory

- **B.1 Foundations**
- **B.2 Pushouts and Pullbacks**
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C Abstract Reduction Systems</strong></td>
<td>57</td>
</tr>
<tr>
<td>C.1 Foundations</td>
<td>57</td>
</tr>
<tr>
<td>C.2 Termination</td>
<td>58</td>
</tr>
<tr>
<td>C.3 Confluence</td>
<td>59</td>
</tr>
<tr>
<td><strong>D Graph Transformation</strong></td>
<td>61</td>
</tr>
<tr>
<td>D.1 Partially Labelled Graphs</td>
<td>61</td>
</tr>
<tr>
<td>D.2 Typed Graphs</td>
<td>62</td>
</tr>
<tr>
<td>D.3 Pushouts and Pullbacks</td>
<td>63</td>
</tr>
<tr>
<td>D.4 Rules and Derivations</td>
<td>63</td>
</tr>
<tr>
<td>D.5 Transformation Systems</td>
<td>64</td>
</tr>
<tr>
<td>D.6 Confluence and Termination</td>
<td>65</td>
</tr>
<tr>
<td>D.7 Critical Pair Analysis</td>
<td>65</td>
</tr>
<tr>
<td>D.8 Rooted Graph Transformation</td>
<td>67</td>
</tr>
<tr>
<td>D.9 Edge Replacement Grammars</td>
<td>68</td>
</tr>
<tr>
<td><strong>E Graph Theory</strong></td>
<td>69</td>
</tr>
<tr>
<td>E.1 Basic Definitions</td>
<td>69</td>
</tr>
<tr>
<td>E.2 Classes of Graphs</td>
<td>70</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>73</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Example Concrete Graph ........................................ 2
1.2 Graph Morphism Commuting Diagrams .......................... 2
1.3 Example Concrete Graphs ........................................ 2
1.4 Relabelling Non-Example ........................................ 3
1.5 Complexity Assumptions Table .................................. 5
1.6 Example Rooted Rule ............................................. 5

2.1 Example Rooted Derivation ....................................... 9
2.2 Example Graph .................................................... 10
2.3 Graph Morphism Commuting Diagrams .......................... 11
2.4 Commuting Squares ............................................... 13
2.5 Example (Non-)Isomorphic Rules ............................... 16
2.6 Derivations Diagram ............................................. 17
2.7 Example Rules Demonstrating Non-Equivalence ............... 20

3.1 Complexity Assumptions Table .................................. 22
3.2 Tree Grammar Rules ............................................... 24
3.3 Tree Recognition Rules ......................................... 25
3.4 Example Reductions .............................................. 25
3.5 Full Binary Tree Rules ........................................... 28
3.6 GP2 Implementation ............................................... 29
3.7 Graph Classes .................................................... 30
3.8 Measured Performance ............................................ 30
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Example Reduction Rules</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Example Non-Closed Rules</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Example Non-Joinable Pair</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Pair Factorisation Diagram</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>EFD Grammar Rules</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Non-Joinable Critical Pair</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Example Encoded PLRG</td>
<td>40</td>
</tr>
<tr>
<td>5.2</td>
<td>Non-Strongly Joinable Critical Pair</td>
<td>45</td>
</tr>
<tr>
<td>B.1</td>
<td>Pushout and Pullback</td>
<td>56</td>
</tr>
<tr>
<td>C.1</td>
<td>Confluence Properties</td>
<td>59</td>
</tr>
<tr>
<td>D.1</td>
<td>Partially Labelled Graph Diagram</td>
<td>61</td>
</tr>
<tr>
<td>D.2</td>
<td>Direct Derivation</td>
<td>63</td>
</tr>
<tr>
<td>D.3</td>
<td>Sequential Independence Diagram</td>
<td>65</td>
</tr>
<tr>
<td>D.4</td>
<td>Parallel Independence Diagram</td>
<td>66</td>
</tr>
<tr>
<td>D.5</td>
<td>Edge Replacement Rule</td>
<td>68</td>
</tr>
</tbody>
</table>
## List of Theorems

<table>
<thead>
<tr>
<th>Section</th>
<th>Theorem (Title)</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Theorem (Property Undecidability)</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Theorem (Derivation Theorem)</td>
<td>15</td>
</tr>
<tr>
<td>2.2</td>
<td>Theorem (Compatible Derivations)</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Theorem (Membership Test)</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Theorem (GT System Equivalence)</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Theorem (Fast Derivations)</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Theorem (Tree Recognition)</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>Theorem (FBT Recognition)</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Theorem (Undecidable Closedness)</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Theorem (Newman-Garbage Lemma)</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Theorem (Non-Garbage Critical Pair Lemma)</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Theorem (Efficient Recognition Correctness)</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Theorem (Efficient EFD Recognition)</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Theorem (Compatible Derivations)</td>
<td>41</td>
</tr>
<tr>
<td>5.2</td>
<td>Theorem (Sequential Independence)</td>
<td>42</td>
</tr>
<tr>
<td>5.3</td>
<td>Theorem (Parallel Independence)</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Theorem (Compatible Strong Joinability)</td>
<td>43</td>
</tr>
<tr>
<td>5.5</td>
<td>Theorem (Non-Garbage Critical Pair Lemma)</td>
<td>44</td>
</tr>
<tr>
<td>A.1</td>
<td>Theorem (Well-Ordered Sets)</td>
<td>54</td>
</tr>
<tr>
<td>A.2</td>
<td>Theorem (Countable Sets)</td>
<td>54</td>
</tr>
<tr>
<td>Theorem</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>B.1 Theorem (Limit Uniqueness)</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>C.1 Theorem (Noetherian Induction)</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>C.2 Theorem (Church-Rosser)</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>C.3 Theorem (Newman’s Lemma)</td>
<td>59</td>
<td></td>
</tr>
<tr>
<td>D.1 Theorem (Typed-Labelled Graph Correspondence)</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>D.2 Theorem (Limit Existence)</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>D.3 Theorem (Derivation Theorem)</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>D.4 Theorem (Property Undecidability)</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>D.5 Theorem (Sequential Independence)</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>D.6 Theorem (Parallel Independence)</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>D.7 Theorem (Critical Pair Lemma)</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>D.8 Theorem (Rooted Derivation Uniqueness)</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>D.9 Theorem (Decidable Membership Problem)</td>
<td>68</td>
<td></td>
</tr>
<tr>
<td>E.1 Theorem (Graph Decomposition)</td>
<td>70</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary

Motivation and Goals

Graph transformation is the rule-based modification of graphs, and is a discipline dating back to the 1970s, with the ‘algebraic approach’ invented at the Technical University of Berlin by Ehrig, Pfender, and Schneider [1], [2]. It is a comprehensive framework in which the local transformation of structures can be modelled and studied in a uniform manner [3]–[5]. Applications in Computer Science are wide-reaching including compiler construction, software engineering and concurrent systems [6]–[9].

The declarative nature of graph rewriting rules comes at a cost: In general, to match the left-hand graph $L$ of a rule within a host graph $G$ requires $|G|^{|L|}$ time. To improve matching performance, Dörr [10] proposed to equip rules and host graphs with distinguished (root) nodes, and to match roots in rules with roots in host graphs. This concept has been implemented in GP 2, allowing GP 2 specifications to rival the performance of traditional implementations in languages such as C [11]. Automated refinement of graph rewriting specifications to rooted versions remains an open research problem.

Graph transformation with root nodes and relabelling is not yet well understood. With only relabelling, Habel and Plump have been able to recover many, but not all, of the standard results [12], [13]. Moreover, Bak and Plump’s model suffers from the problem that derivations are not necessarily invertible. This motivates us to develop a new model of rooted graph transformation with relabelling which does not suffer this problem. If we have termination and invertibility, then we have an algorithm for testing graph language membership, and if we have confluence (and constant time matching), then we have an efficient algorithm too [14], [15].

Testing for ‘confluence’ is not possible in general [16], however we can sometimes use ‘critical pair’ analysis to show confluence. Confluence remains poorly understood, and while there are techniques for classifying ‘conflicts’ [17]–[19], it is rarely possible to actually show confluence. Moreover, in general, confluence is stronger than required for language efficient membership testing, motivating a weaker definition of confluence.
Our method will be to use mathematical definitions and proofs, as is usual in theoretical computer science. We aim to:

1. Outline rooted DPO graph transformation with relabelling;
2. Repair the problem of lack of invertibility in rooted GT systems;
3. Develop a new example of linear time graph algorithm;
4. Develop new results for confluence analysis of GT systems.

Results and Evaluation

We regard this project as a success, both the original and extended versions, having achieved our four goals. Our first, second and third goals have been addressed by the first, second and third chapters, respectively, and the final goal by the next two chapters. We started by briefly reviewing the current state of graph transformation, with a particular focus on the ‘injective DPO’ approach with relabelling and graph programming languages, establishing issues with the current approach to rooted graph transformation due to its ‘pointed’ implementation. We also briefly reviewed DPO-based graph programming languages.

We address the lack of invertibility of rooted derivations by defining rootedness using a partial function onto a two-point set rather than pointing graphs with root nodes. We have shown rule application corresponds to ‘NDPOs’, how Dodds’ complexity theory [20] applies in our system, and briefly discussed the equivalence of and refinement of GT systems. Developing a fully-fledged theory of correctness and refinement for (rooted) GT systems remains future work, as does establishing if the Parallelism and Concurrency theorems hold [13, 21]. Applications of our model to efficient graph class recognition are exciting due to the invertibility of derivations.

We have shown a new result that the graph class of trees can be recognised by a rooted GT system in linear time, given an input graph of bounded degree. Moreover, we have given empirical evidence by implementing the algorithm in GP2 and collecting timing results. The program and results were presented at CALCO 2019 [22]. Overcoming the restriction of host graphs to be of bounded degree remains open research, as well as showing further case studies and applications.

We have defined a new notion of ‘confluence up to garbage’ and ‘non-garbage critical pairs’, and shown that it is sufficient to require strong joinability of only the non-garbage critical pairs to establish confluence up to garbage. Moreover, we have lifted our results to our new framework for rooted graph transformation with relabelling via a faithful functor. We have applied this theory to Extended Flow Diagrams [23] and the encoding of partially labelled (rooted) GT systems as standard GT systems, performing non-garbage crit-
ical pair analysis on the encoded system. Further exploring the relationship between confluence up to garbage and closedness remains open work, as does improving the analysis of (non-garbage) critical pairs to allow us to decide confluence in more cases than currently possible via pair analysis.

Ethical Considerations

This project is of a theoretical nature. As such, no human participants were required, and no confidential data has been collected. Moreover, there are no anticipated ethical implications of this work or its applications.
Preface

This report is an extended version of the BSc Thesis of Graham Campbell [24] which incorporates additional results developed over 10 weeks, funded by a Vacation Internship of the Engineering and Physical Sciences Research Council (EPSRC) and supervised by Dr. Detlef Plump. Part of Chapter 3 was presented at CALCO 2019 as part of a co-authored paper looking at linear time algorithms in GP 2 [22].

Compared to the original thesis, the focus has been even more on fast language recognition. We have provided a more detailed introduction and theoretical background, and more care has been taken when defining the new theory, due to the page limit being removed. We have separated the semantic definitions of bounded degree and bounded roots preservation from the syntactic condition, which gives clarity that the syntactic condition is both sufficient and necessary. We have also provided a linear time full binary tree recognition system on graphs of bounded degree.

Additionally, in the second half of the report, we have reworked the previous definitions of confluence up to garbage and closure, changing, give an result that closedness is undecidable in general, and an example showing that a system can be locally confluent up to garbage and terminating, but not confluent up to garbage. Finally, we give a formal encoding of our new rooted GT systems as standard GT systems, which allows us to recover the Local Church-Rosser Theorem and the Non-Garbage Critical Pair Lemma in our new setting.

The structure of the report is as follows. We:

1. Provide a quick introduction and outline the theoretical background;
2. Present our new framework for rooted DPO graph transformation with relabelling and invertible rules;
3. Look at complexity of derivations in our new framework, and show that the framework is sufficient to recognise trees in linear time;
4. Develop new results for confluence analysis of conventional GT systems;
5. Lift our new results to the rooted framework via a faithful functor;
6. Summarise and evaluate our results, and list future work.
In the appendices, we have provided an overview of the notation used for sets, functions, relations, categories, and abstract reduction systems, and also some key results that we will be needing, from the literature. We also provide supplementary technical background on ‘conventional’ graph transformation, and graph theory, including the definitions of various graph classes.

During the period covered by the grant, Campbell also worked on improving the GP2 Compiler. That work has not been summarised in this report, however is available in a separate report [25].
Chapter 1

Theoretical Background

“Mathematics is a game played according to certain simple rules with meaningless marks on paper.”

– David Hilbert, N Rose Mathematical Maxims and Minims (1988)

Graph transformation is the rule-based modification of graphs, and is a discipline dating back to the 1970s, with the ‘algebraic approach’ invented at the Technical University of Berlin by Ehrig, Pfender, and Schneider [1], [2]. It is a comprehensive framework in which the local transformation of structures can be modelled and studied in a uniform manner [3]–[5].

In this chapter, we will review the rewriting of totally labelled graphs with relabelling, and Bak and Plump’s modifications adding ‘root’ nodes [26]. We will see how (rooted) graph transformation systems are instances of abstract reduction systems, and will look at graph programming languages.

1.1 Graphs and Morphisms

There are various definitions of a ‘graph’. In particular, we are interested in graphs where edges are directed and parallel edges are permitted. We will start by introducing ‘unlabelled’ graphs. We will then build on these with (partially) labelled (rooted) graphs and morphisms.

Definition 1.1. We can formally define a concrete graph as:

\[ G = (V, E, s : E \to V, t : E \to V) \]

where \( V \) is a finite set of vertices, \( E \) is a finite set of edges. We call \( s : E \to V \) the source function, and \( t : E \to V \) the target function.

Definition 1.2. If \( G \) is a concrete graph, then \( |G| = |V_G| + |E_G| \).
Example 1.1. Consider the concrete graph $G = (\{1, 2, 3\}, \{a, b, c, d\}, s, t)$ where $s = \{(a, 1), (b, 2), (c, 3), (d, 3)\}$, $t = \{(a, 2), (b, 1), (c, 1), (d, 3)\}$ (treating functions as sets). Its graphical representation is given in Figure 1.1. Note that the numbers are not ‘labels’, but ‘node ids’.

\[ \text{Figure 1.1: Example Concrete Graph} \]

Definition 1.3. Given two concrete graphs $G$ and $H$, a graph morphism $g : G \rightarrow H$ is a pair of maps $g = (g_V : V_G \rightarrow V_H, g_E : E_G \rightarrow E_H)$ such that sources and targets are preserved. That is, $\forall e \in E_G, g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$. Equivalently, both of the squares in Figure 1.2 commute.

\[ \text{Figure 1.2: Graph Morphism Commuting Diagrams} \]

Definition 1.4. A graph morphism $g : G \rightarrow H$ is injective/surjective iff both $g_V$ and $g_E$ are injective/surjective as functions. We say $g$ is an isomorphism iff it is both injective and surjective.

Example 1.2. The identity morphism $(id_V, id_E)$ is an isomorphism between any graph and itself.

Example 1.3. Consider the graphs in Figure 1.3. There are four morphisms $G \rightarrow H$, three of which are injective, none of which are surjective. There are actually also four morphisms $H \rightarrow G$, three of which are surjective.

\[ \text{Figure 1.3: Example Concrete Graphs} \]

Definition 1.5. We say that graphs $G, H$ are isomorphic iff there exists a graph isomorphism $g : G \rightarrow H$, and we write $G \cong H$. This naturally gives rise to equivalence classes $[G]$, called abstract graphs.

Proposition 1.1. The quotient (Definition A.21) of the collection of all concrete graphs with $\cong$ is the countable set of all abstract graphs.

1.2 Graph Transformation

There are various approaches to graph transformation, most notably the ‘edge replacement’ [27], ‘node replacement’ [28], and ‘algebraic’ approaches [3], [4]. The two major approaches to algebraic graph transformation are the so called
1.3. ADDING RELABELLING

‘double pushout’ (DPO) approach, and the ‘single pushout’ (SPO) approach. Because the DPO approach operates in a structure-preserving manner (rule application in SPO is without an interface graph, so there are no dangling condition checks), this approach is more widely used than the SPO [4], [5, p.9-14]. For this reason, we will focus only on the DPO approach with injective matching.

Given an unlabelled graph (Definition 1.1), there are two common approaches to augmenting it with data: typed graphs and totally labelled graphs. We choose to work with the labelled approach (Section D.1) because it is easy to understand and reason about, has a relabelling theory (Section 1.3), and a ‘rooted’ modification (Section 1.4). Details of the typed approach can be found in Section D.2. Note that typed (attributed) (hyper)graphs have a rich theory [15], [29]–[35].

A review of graph transformation of labelled graphs using the DPO approach with injective matching can be found in Appendix D. We will also cover the definitions and results for our new type of system in Chapter 2, so we will not repeat ourselves in this chapter by giving all of the detail again. Additionally, an example system and grammar can be found in Chapter 3.

1.3 Adding Relabelling

The origin of partially labelled graphs is from the desire to have ‘relabelling’. If the interface $K$ is totally labelled, then any node which has context (incident edges) cannot be deleted, and so we must preserve its label to avoid breaking uniqueness of rule application. We can get around this problem with partial labelling of interface graphs, and thus with modest modifications to the theory for totally labelled graphs we allow rules to ‘relabel’ nodes. We shall be using this foundation going forward. All the relevant definitions and theorems are in Appendix D.

We are in fact using a restricted version of the theory presented by Habel and Plump [12], the restriction being that we allow the interface $K$ to be partially labelled, but require $L$, $R$ and $G$ to be totally labelled, ensuring that given a totally labelled input graph $G$, the result graph $H$ is also totally labelled. Thus, derivations are defined only on totally labelled graphs, but allow us to relabel nodes.

**Example 1.4.** Consider the following totally labelled ‘rule’, over the label alphabet ({$1, 2$}, {□}) where $x, y$ are to be determined:

$$
\begin{array}{ccc}
1 & \xrightarrow{1} & 2 \\
\downarrow & & \downarrow \\
1 & & 1 \\
\end{array}
\begin{array}{ccc}
\xrightarrow{2} & \xrightarrow{2} & \rightarrow \\
\downarrow & & \downarrow \\
1 & & 1 \\
\end{array}
$$

Figure 1.4: Relabelling Non-Example
We want to swap the labels without deleting the nodes, because they may have context. There is no value we can choose for $x$ or $y$ such that the conditions to be a totally labelled graph morphism are satisfied. Now consider the setting where we allow the interface graph to have a partial node label map. We could simply not label the interface nodes, and then we have exactly what we want. \[\triangle\]

### 1.4 Rooted Graph Transformation

Rooted graph transformation first appeared when Dörr [10] proposed to equip rules and host graphs with distinguished (root) nodes, and to match roots in rules with roots in host graphs. Root nodes were first considered in the DPO setting by Dodds and Plump [14], [20], and most recently, Bak and Plump [11], [26] have used rooted graph transformation in conjunction with the theory of partially labelled graph transformation in GP2.

The motivation for root nodes is to improve the complexity of finding a match of the left-hand graph $L$ of a rule within a host graph $G$. In general, linear time graph algorithms may, instead, take polynomial time when expressed as graph transformation systems [14], [26], [36], [37]. An excellent account of this is available in Part II of Dodds’ Thesis [20].

We can define rooted graphs in a pointed style, just as for typed graphs. An account of the theoretical modifications is provided in Section D.8, using Bak’s approach [26]. This is different to Dodds’ approach [14], [20] which implemented roots as a special node label. Bak’s approach allows nodes to have genuine labels and also be rooted, and has been implemented in GP2.

We can formalise the problem of applying a rule:

**Definition 1.6** (Graph Matching Problem (GMP)). Given a graph $G$ and a rule $r = \langle L \leftarrow K \rightarrow R \rangle$, find the set of injective graph morphisms $L \rightarrow G$.

**Definition 1.7** (Rule Application Problem (RAP)). Given a graph $G$, a rule $r = \langle L \leftarrow K \rightarrow R \rangle$, and an injective match $g : L \rightarrow G$, find the result graph $H$. That is, does it satisfy the ‘dangling condition’, and if so, construct $H$.

**Proposition 1.2.** The GMP requires time $O(|G|^{|L|})$ time given the assumptions in Figure 1.5, derived from Dodds’ PhD Thesis [20]. Moreover, given a match, one can decide if it is applicable in $O(|r|)$ time. That is, the RAP requires $O(|r|)$ time. [20]

To improve matching performance, one can add root nodes to rules and match roots in rules with roots in host graphs, meaning we need only consider subgraphs of bounded size for matching, vastly improving the time complexity. That is, given a graph $G$ of bounded degree (Definition E.7) containing a
bounded number of root nodes, and a rule of bounded size with $L$ containing a single root node, then the time complexity of GMP is only constant [20].

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>label $l$</td>
<td>The set $Z$ of nodes with label $l$.</td>
<td>$O(</td>
</tr>
<tr>
<td>node $v$</td>
<td>Values $\deg(v)$, $\text{indeg}(v)$, $\text{outdeg}(v)$.</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>node $v$, label $l$</td>
<td>No. edges with source $v$, label $l$.</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>node $v$, label $l$</td>
<td>No. edges with target $v$, label $l$.</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>node $v$, label $l$</td>
<td>Set $X$ of edges with source $v$, label $l$.</td>
<td>$O(</td>
</tr>
<tr>
<td>node $v$, label $l$</td>
<td>Set $X$ of edges with target $v$, label $l$.</td>
<td>$O(</td>
</tr>
<tr>
<td>graph $G$</td>
<td>$</td>
<td>V_G</td>
</tr>
</tbody>
</table>

Figure 1.5: Complexity Assumptions Table

Example 1.5. Figure 1.6 "moves the root node" and also ‘relabels’ the nodes in the host graph. A ‘fast’ rooted implementation of the 2-colouring problem is available at [26], showcasing root nodes in GP 2. △

![Figure 1.6: Example Rooted Rule](image)

We will revisit time complexity in Section 3.1, showing that if rules are of a certain type, then derivations take only constant time, allowing us to use only derivation length as a measure of time complexity, as in standard complexity analysis theory for (non-deterministic) Turing Machines, first considered by Hartmanis and Stearns in 1965 [38].

1.5 Graph Languages

Intuitively, a graph language is simply a set of graphs, just like a string language is a set of strings. Just like we can define string languages using string grammars, we can define graph languages using graph grammars, where we rewrite some start graph using a set of graph transformation rules. Derived graphs are then defined to be in the language if they are terminally labelled. We formally define graph languages in Section D.5.

**Definition 1.8.** Let $\mathcal{L}$ be some fixed label alphabet (Definition D.1). We let $\mathcal{G}(\mathcal{L})$ be the **collection** of all totally labelled **abstract graphs**, and $\mathcal{G}_r(\mathcal{L})$ be the **collection** of all totally labelled, totally rooted **abstract graphs**.

**Proposition 1.3.** Given some $\mathcal{L}$, $\mathcal{G}(\mathcal{L})$ and $\mathcal{G}_r(\mathcal{L})$ are **countable sets**.

There are only countably many abstract graphs; an essential property when setting up graph languages. We would be in trouble if the universal language
was not countable, or not even a set! Naming of elements from uncountable sets requires non-standard theory (such as Type-2 Computability Theory \[39\]), and we get into even more trouble when we want to name elements from sets larger than the continuum cardinality. When we make computability statements about languages, we are tacitly assuming a Gödel numbering \[40\].

We choose to define languages as abstract graphs, so that they are countable sets, rather than proper classes. Moreover, when one talks of a finite graph language, we can genuinely be referring to its cardinality, and not the cardinality of its quotient under graph isomorphism. This approach is not standardised, and many authors prefer to define graph languages in terms of concrete graphs. The other obvious approach (used by Courcelle and Engelfriet \[41\]) is to restrict the ambient from which nodes and edges are drawn to some countable set which is closed under pairing (to allow for the disjoint union construction when computing pushouts).

Uesu showed in 1978 that the DPO graph grammars can generate every recursively enumerable (Definition A.30) set of graphs \[42\]. A large body of literature exists when it comes to context-free graph grammars, however the focus is on (hyper)edge replacement grammars \[27\] and node replacement grammars \[28\], which are less powerful than DPO-grammars. We have provided a formal definition of edge replacement grammars in Section D.9.

1.6 Abstract Reduction Systems

Abstract reduction systems (or simply reduction systems or ARS) are a much more general setting than graph transformation systems (GT systems or GTS), and model the step-wise transformation of objects (see Appendix C). These systems were studied for the first time by Newman in the early 40s \[43\]. Turing Machines and GT systems clearly fit into this model of reduction. Moreover, the formal semantics of programming languages is often defined in terms of a step-wise computation relation.

Example 1.6. \((\mathbb{N}, >)\) is a terminating (Definition C.4), finitely branching (Definition C.6), confluent (Definition C.8) ARS (Definition C.1). \(\triangle\)

Example 1.7. \((\mathbb{Z}, >)\) by comparison is not terminating or finitely branching, but it is confluent! \(\triangle\)

Definition 1.9. Let \(T = (\mathcal{L}, \mathcal{R})\) be a (rooted) GTS. Then \((\mathcal{G}(\mathcal{L}), \rightarrow)\) is the induced ARS defined by \(\forall [G], [H] \in \mathcal{G}(\mathcal{L}), [G] \rightarrow [H] \iff G \Rightarrow R H.\)

Lemma 1.1. Consider the ARS \((\mathcal{G}(\mathcal{L}), \rightarrow)\) induced by a (rooted) GTS. Then \(\rightarrow\) is a binary relation (Definition A.14) on \(\mathcal{G}(\mathcal{L})\). Moreover, it is finitely branching (Definition C.6) and decidable (Definition A.29).

Proof. By Proposition 1.3, \(\mathcal{G}(\mathcal{L})\) is a countable set, and so \(\rightarrow\) is a countable set (by Theorem A.2), and is well-defined since derivations are unique up to
isomorphism (Theorem D.8). Finally, we have only finitely many rules, and for each rule, there can only exist finitely many matches \( L \rightarrow G \), so there can only ever be finitely many result graphs \( H \) (up to isomorphism) \( G \Rightarrow_R H \) for any given \( G \).

\[ \square \]

**Theorem 1.1** (Property Undecidability). Consider the ARS \( (G(L), \rightarrow) \) induced by a (rooted) GTS. Then testing if \( \rightarrow \) is terminating, acyclic, or (locally) confluent is undecidable in general.

**Proof.** Testing for acyclicity or termination was shown to be undecidable in general by Plump in 1998 [44]. Undecidability of (local) confluence checking was shown by Plump in 1993 [16], even for terminating GT systems [45]. \( \square \)

While confluence testing is undecidable, there are some instances in which one can decide confluence or non-confluence of GT systems. In 1970, Knuth and Bendix showed that confluence checking of terminating term rewriting systems is decidable [46]. Moreover, it suffices to compute all ‘critical pairs’ and check their joinability [47]–[49]. Unfortunately, joinability of critical pairs does not imply local confluence of GT system, otherwise we could have contradicted our theorem! In 1993, Plump showed that ‘strong joinability’ of all critical pairs is sufficient but not necessary to show local confluence [16], [45].

We have summarised these results in Section D.7.

### 1.7 Graph Programming Languages

GT systems naturally lend themselves to expressing computation by considering the normal forms of the input graph.

**Example 1.8.** Given a GT system \( T = (L, R) \), consider the state space \( \Sigma = G(L) \cup \{ \bot \} \) and the induced ARS \( (G(L), \rightarrow_R) \). We may define the semantic function \( f_T : G(L) \rightarrow P(\Sigma) \) by \( f_T([G]) = \{ [H] \mid [H] \) is a normal form of \([G]\) with respect to \( \rightarrow_R \} \cup \{ \bot \mid \) there is an infinite reduction sequence starting from \([G]\) and \( f_T(\bot) = \{ \bot \}. \)

\[ \triangle \]

There are a number of GT languages and tools, such as AGG [50], GMTE [51], Dactl [52], GP 2 [53], GReAT [54], GROOVE [55], GrGen.Net [56], Henshin [57], PROGRES [58], and PORGY [59]. Habel and Plump [60] show that such languages can be ‘computationally complete’:

**Proposition 1.4.** To be computationally complete, the three constructs:

1. Nondeterministic application of a rule from a set of rules (\( R \));
2. Sequential composition (\( P_1; P_2 \));
3. Iteration in the form that rules are applied as long as possible \( P_\downarrow \).

are not only sufficient, but necessary (using DPO-based rule application).
Example 1.9. The semantics of some program $P$ is a binary relation $\rightarrow_P$ on some set of abstract (rooted) graphs $\mathcal{G}$, inductively defined as follows:

1. $\rightarrow_{\mathcal{R}} := \rightarrow$ (where $\rightarrow$ is the induced ARS relation on $\mathcal{R}$).
2. $\rightarrow_{P_1; P_2} := \rightarrow_{P_2} \circ \rightarrow_{P_1}$.
3. $\rightarrow_{\downarrow} P := \{([G], [H]) \mid [G] \rightarrow^* P [H] \text{ and } [H] \text{ is in normal form}^1\}$.

Remark 1.1. While GT systems can ‘simulate’ any Turing Machine, this does not make them ‘computationally complete’ in the strong sense that any computable function on arbitrary graphs can be programmed.

GP2 is an experimental rule-based language for problem solving in the domain of graphs, developed at York, the successor of GP [53], [61]. GP2 is of interest because it has been designed to support formal reasoning on programs [62], with a semantics defined in terms of partially labelled graphs, using the injective DPO approach with relabelling [12], [63]. Poskitt and Plump have set up the foundations for verification of GP2 programs [64]–[67] using a Hoare-Style [68] system (actually for GP [61], [69]), Hristakiev and Plump have developed static analysis for confluence checking [18], [70], and Bak and Plump have extended the language, adding root nodes [11], [26]. Plump has shown computational completeness [71].

GP2 uses a model of ‘rule schemata’ with ‘application conditions’, rather than ‘rules’ as we have seen up until now. The label alphabet used for both nodes and edges is $(\mathbb{Z} \cup \text{Char}^*)^* \times \mathcal{B}$. Roughly speaking, rule application works by finding an injective ‘premorphism’ by ignoring labels, and then checking if there is an assignment of values such that after evaluating the label expressions, the morphism is label-preserving. The application condition is then checked, then rule application continues. [53]

The formal semantics of GP2 is given in the style of Plotkin’s structural operational semantics [72]. Inference rules inductively define a small-step transition relation $\rightarrow$ on configurations. The inference rules and definition of the semantic function $\llbracket \cdot \rrbracket : \text{ComSeq} \rightarrow \mathcal{G} \rightarrow \mathcal{P}(\mathcal{G} \cup \{\text{fail}, \bot\})$ were first defined in [53]. Up-to-date versions can be found in [11].

---

$^1[H]$ is in normal form iff it is not reducible using $\rightarrow_P$. 

Chapter 2

A New Theory of Rooted Graph Transformation

“In mathematics the art of asking questions is more valuable than solving problems.”

– Georg Cantor, Doctoral thesis (1867)

Graph transformation with relabelling as described in Sections 1.2, 1.3, D.4 and D.5 has desirable properties. It was shown by Habel and Plump in 2002 [12] that derivations are natural double pushouts (Theorem D.3) and thus are invertible. Unfortunately, Bak and Plump’s modifications to add root nodes (Sections 1.4 and D.8) mean that derivations no longer exhibit these properties. That is, only the right square of a derivation in a rooted GT system need be a natural pushout. This asymmetry is unfortunate, because derivations are no longer invertible. Figure 2.1 shows an example rooted derivation in which the right square is not a pullback.

![Diagram of a rooted derivation](image)

Figure 2.1: Example Rooted Derivation

We propose an alternative theory for rooted graph transformation with relabelling, with some more desirable properties. Critically, we restore invertibility of derivations (Corollary 2.1), and remove some undesirable matching cases (Lemma 2.5), allowing us to prove a handy root node invariance result (Corollary 2.2). Our restrictions are slightly more than Habel and Plump’s original version of graph transformation with relabelling, however our interest
2.1 Graphs and Morphisms

Fix some common label alphabet (Definition 2.1) \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \). In this section we define our new notions of graphs and morphisms.

Definition 2.1. A graph over \( \mathcal{L} \) is a tuple \( G = (V, E, s, t, l, m, p) \) where:

1. \( V \) is a finite set of vertices;
2. \( E \) is a finite set of edges;
3. \( s : E \to V \) is a total source function;
4. \( t : E \to V \) is a total target function;
5. \( l : V \to \mathcal{L}_V \) is a partial function, labelling the vertices;
6. \( m : E \to \mathcal{L}_E \) is a total function, labelling the edges;
7. \( p : V \to \mathbb{Z}_2 \) is a partial function, determining vertex rootedness.

Definition 2.2. If \( G \) is a graph, then we define its size \( |G| = |V_G| + |E_G| \).

Definition 2.3. A graph \( G \) is totally labelled iff \( l_G \) is total, and totally rooted if \( p_G \) is total. If \( G \) is both, then we call it a TLRG.

Remark 2.1. A totally rooted graph need not have every node a root node, only \( p_G \) must be total. 0 denotes unrooted, and 1 rooted. When we draw graphs, we shall denote the absence of rootedness with diagonal stripes. If a node has a double border, it is rooted, otherwise, it is unrooted.

Example 2.1. Let \( \mathcal{L} = (\{\square, \triangle\}, \{x, y\}) \). Then \( G = (V, E, s, t, l, m, p) \) is a graph over \( \mathcal{L} \) where \( V = \{1, 2, 3, 4\} \), \( E = \{1, 2\} \), \( s = \{(1, 1), (2, 2)\} \), \( t = \{(1, 2), (2, 3)\} \), \( l = \{(1, \square), (2, \triangle)\} \), \( m = \{(1, x), (2, y)\} \), and \( p = \{(1, 0), (2, 1), (4, 0)\} \).

Its graphical representation is shown in Figure 2.2. \( G \) is neither totally rooted nor totally labelled, since node 3 has both undefined rootedness and no label, and node 4 also has no label.

\[\text{Figure 2.2: Example Graph}\]

Definition 2.4. A graph morphism between graphs \( G \) and \( H \) is a pair of functions \( g = (g_V : V_G \to V_H, g_E : E_G \to E_H) \) such that sources, targets, labels, and rootedness are preserved. That is:

1. \( \forall e \in E_G, g_V(s_G(e)) = s_H(g_E(e)) \); [Sources]
2. \( \forall e \in E_G, g_V(t_G(e)) = t_H(g_E(e)) \); [Targets]

\[^2\mathbb{Z}_2 \text{ is the quotient } \mathbb{Z}/2\mathbb{Z} = \{0, 1\}.\]
3. \( \forall e \in E_G, m_G(e) = m_H(g_E(e)) \); [Edge Labels]
4. \( \forall v \in \text{int}^{-1}_G(\mathcal{L}_V), l_G(v) = l_H(g_V(v)) \); [Node Labels]
5. \( \forall v \in \text{int}^{-1}_G(\mathbb{Z}_2), p_G(v) = p_H(g_V(v)) \). [Rootedness]

**Remark 2.2.** If \( G \) and \( H \) are TLRGs, then this is equivalent to the following diagram commuting (for \( s_G, s_H \) and \( t_G, t_H \) separately):

![Figure 2.3: Graph Morphism Commuting Diagrams](image)

An important property of morphisms is that if a node is labelled, then its image under the morphism must have the same label, and the same for the rootedness. However, if the label or rootedness of a node is not defined, then we do not specify anything about its image. Nodes with undefined labels may be mapped to nodes with undefined labels, but could also be mapped to labelled nodes, and similarly for rootedness. There are some circumstances where it is useful to additionally insist on undefinedness being preserved, which motivates our next definition.

**Definition 2.5.** A graph morphism \( g : G \to H \) is **undefinedness preserving** iff it is both:

1. **Label-undefinedness preserving**: \( l_G = l_H \circ g_V \);
2. **Root-undefinedness preserving**: \( p_G = p_H \circ g_V \).

**Remark 2.3.** The undefinedness preserving property is exactly requiring that the two right-most triangles in Figure 2.3 commute. It is thus a triviality that all morphisms between TLRG are undefinedness preserving.

Just like unlabelled (totally labelled) graphs and morphisms, our new notion of graphs and morphisms forms a category where the graphs are the objects and the morphisms are the arrows.

**Proposition 2.1.** Our new notion of graphs and morphisms is a **locally small** (Definition B.3), **finitary** (Definition B.4) **category** (Definition B.1).

**Definition 2.6.** A graph morphism \( g : G \to H \) is **injective** (surjective) iff the underlying functions \( g_V, g_E \) are injective (surjective). We say that \( g \) is an **isomorphism** iff it is both injective and surjective, and \( g^{-1} : H \to G \) is also a graph morphism.

**Proposition 2.2.** Morphisms are injective (surjective, isomorphisms) iff they are **monic** (epic, isomorphisms) in the category theoretic sense.
Proposition 2.3. A morphism is an isomorphism iff it is injective, surjective, and undefinedness preserving.

Example 2.2. There exist morphisms that are injective, surjective graph morphisms that are not isomorphisms. Just consider two discrete (no edges) graph with one node, the first with no node label, and the second with some label. There is an injective, surjective morphism from the first to second, but not in the other direction. △

Definition 2.7. We say \( H \) is a subgraph of \( G \) iff there exists an inclusion morphism \( H \leftarrow G \). This happens iff \( V_H \subseteq V_G, E_H \subseteq E_G, s_H = s_G|_{E_H}, t_H = t_G|_{E_H}, m_H = m_G|_{E_H}, l_H \subseteq l_G, p_H \subseteq p_G \) (treating functions as sets).

Remark 2.4. There are two obvious (non-equivalent) definitions of subgraph. We have chosen here to define subgraphs to be exactly those that can can be included in the parent structure. This is particularly useful when it comes to defining rules later. The other definition (closer to that given by Habel and Plump for partially labelled graphs) would be to require the morphism to also be undefinedness preserving.

Definition 2.8. We say that graphs \( G, H \) are isomorphic iff there exists a graph isomorphism \( g : G \rightarrow H \). This gives equivalence classes \([G]\) over \( \mathcal{L} \). We denote by \( \mathcal{G}^\oplus(\mathcal{L}) \) the collection of totally labelled, totally rooted abstract graphs over some fixed \( \mathcal{L} \).

There are only countably many abstract graphs; an essential property when setting up graph languages. As we remarked in Chapter 1, we would be in trouble if the universal language was not a countable set.

Proposition 2.4. \( \mathcal{G}^\oplus(\mathcal{L}) \) is a countable set.

2.2 Rules and Derivations

Fixing some common \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \), we define rules and derivations. We only concern ourselves, for now, with ‘linear rules’ and injective matches [63]. Moreover, some of the results we will be showing later don’t hold in more general versions of this set up.

Definition 2.9. A rule \( r = \langle L \leftarrow K \rightarrow R \rangle \) consists of left/right TLRGs \( L, R \), the interface graph \( K \), and inclusions \( K \leftarrow L \) and \( K \rightarrow R \).

Remark 2.5. Insisting on inclusions is without loss of generality. We could have only required two injective graph morphisms, but assuming inclusions makes for neater proofs involving rules. Moreover, we can think of \( K \) as a subgraph of both \( L \) and \( R \).

Example 2.3. See Figure 3.3. △
Definition 2.10. We define the inverse rule to be \( r^{-1} = \langle R \leftarrow K \rightarrow L \rangle \).

Definition 2.11. If \( r = \langle L \leftarrow K \rightarrow R \rangle \) is a rule, then \(|r| = \max\{|L|, |R|\}\).

Definition 2.12. Given a rule \( r = \langle L \leftarrow K \rightarrow R \rangle \) and a TLRG \( G \), we say that an injective morphism \( g : L \leftarrow G \) satisfies the dangling condition iff no edge in \( G \setminus g(L) \) is incident to a node in \( g(L \setminus K) \).

Definition 2.13. To apply a rule \( r = \langle L \leftarrow K \rightarrow R \rangle \) to some TLRG \( G \), find an injective graph morphism \( g : L \leftarrow G \) satisfying the dangling condition, then:

1. Delete \( g(L \setminus K) \) from \( G \). For each unlabelled node \( v \) in \( K \), make \( g_{V}(v) \) unlabelled, and for each node \( v \) in \( K \) with undefined rootedness, make \( g_{V}(v) \) have undefined rootedness, giving intermediate graph \( D \).
2. Add disjointly \( R \setminus K \) to \( D \), keeping their labels and rootedness. For each unlabelled node \( v \) in \( K \), label \( g_{V}(v) \) with \( l_{R}(v) \), and for each node with undefined rootedness \( v \) in \( K \), make \( g_{V}(v) \) have rootedness \( p_{R}(v) \), giving the result graph \( H \).

If the dangling condition fails, then the rule is not applicable using the match \( g \). We can exhaustively check all matches to determine applicability.

Definition 2.14. We write \( G \Rightarrow_{r,g} M \) for a successful application of \( r \) to \( G \) using match \( g \), obtaining result \( M \cong H \). We call this a direct derivation. We may omit \( g \) when it is not relevant, writing simply \( G \Rightarrow_{r} M \).

Remark 2.6. Formally, a direct derivation is actually the diagram Figure 2.4.

Definition 2.15. For a given set of rules \( \mathcal{R} \), we write \( G \Rightarrow_{\mathcal{R}} H \) iff \( H \) is directly derived from \( G \) using any of the rules from \( \mathcal{R} \).

Definition 2.16. We write \( G \Rightarrow^{+}_{\mathcal{R}} H \) iff \( H \) is derived from \( G \) in one or more direct derivations, and \( G \Rightarrow^{*}_{\mathcal{R}} H \) iff \( G \cong H \) or \( G \Rightarrow^{+}_{\mathcal{R}} H \).

2.3 Foundational Theorems

We will show that gluing and deletions correspond to natural pushouts and natural pushout complements, respectively. Thus, derivations are invertible.

\[
\begin{array}{c}
L \leftarrow K \longrightarrow R \\
\downarrow \quad \downarrow (1) \quad \downarrow (2) \\
G \leftarrow D \longrightarrow H
\end{array}
\]

Figure 2.4: Commuting Squares

Our first technical lemma ensures that pullbacks exist in our new setting.
Lemma 2.1. Given graph morphisms \( g : L \to G \) and \( c : D \to G \), there exist a graph \( K \) and graph morphisms \( b : K \to L, d : K \to D \) such that the resulting square is a pullback (Definition B.8).

Proof. The constructions are exactly as in Lemma 1 of [12], with the rootedness function defined analogously to the node labelling function. Trivial modifications to the proof give the result.

Our second technical lemma, roughly speaking, ensures that pullbacks exist in our new setting, in the case \( K \to D \) is injective and that we don’t have any confusion created by conflicting labels or rootedness. In the totally labelled case (without root nodes), we don’t need to worry about such confusion.

Lemma 2.2. Let \( b : K \to R, d : K \to D \) be graph morphisms such that:

1. \( d \) is injective,
2. \( \forall v \in V_R, |l_R(\{v\}) \cup l_D(dV(b^{-1}(\{v\}))| \leq 1 \),
3. \( \forall v \in V_R, |p_R(\{v\}) \cup p_D(dV(b^{-1}(\{v\}))| \leq 1 \).

Then, there exist a graph \( H \) and graph morphisms \( h : R \to H, c : D \to H \) such that the resulting square is a pushout (Definition B.7).

Proof. The constructions are exactly as in Lemma 2 of [12], with the rootedness function defined analogously to the node labelling function.

Our third technical lemma says that as long as \( K \to D \) is injective, and no additional labels or roots are introduced in \( D \), then the left square is a pullback if it is a pushout.

Lemma 2.3. Given two graph morphisms \( b : K \to L \) and \( d : K \to D \) such that \( b \) is injective and \( L \) is a TLRG, then the pushout \((1)\) is natural (Definition B.9) if \( l_D(dV(V_K \setminus l_K^{-1}(L_V))) = \emptyset = p_D(dV(V_K \setminus p_K^{-1}(Z_2))) \).

Proof. Let square \((1)\) in Figure 2.4 be a natural pushout with graph morphisms \( g : L \to G \) and \( c : D \to G \). Once again, we can proceed as in Lemma 3 of [12] with the obvious modifications. Similar for the other direction.

Our final technical lemma shows a correspondence between the existence of natural pushout complements and matches satisfying the dangling condition. Moreover, that the intermediate graph \( D \) is essentially unique (unique up to isomorphism).

Lemma 2.4. Let \( g : L \to G \) be an injective graph morphism and \( K \to L \) an inclusion morphism. Then, there exist a graph \( D \) and morphisms \( K \to D \) and \( D \to G \) such that the square \((1)\) is a natural pushout iff \( g \) satisfies the dangling condition. Moreover, in this case, \( D \) is unique up to isomorphism.

Proof. Proceed as in Lemma 4 of [12] with the obvious modifications.
2.3. FOUNDATIONAL THEOREMS

We now present our first major theorem. The explicit construction of derivations given earlier corresponds exactly to a NDPO diagram. Moreover, the result graph is essentially unique and the input is a \textbf{TLRG} iff the result is.

**Theorem 2.1** (Derivation Theorem). Given a rule \(<L \leftarrow K \rightarrow R>\) and an \textbf{injective graph morphism} \(g : L \rightarrow G\), then there exists a \textbf{natural DPO} diagram as above iff \(g\) satisfies the \textbf{dangling condition}. In this case, \(D\) and \(H\) are unique up to isomorphism, and rule application exactly corresponds to that given in Definition 2.13. Moreover, if \(G \Rightarrow_r H\), then \(G\) is a \textbf{TLRG} iff \(H\) is a \textbf{TLRG}.

**Proof.** Proceed as in Theorem 1 of [12] with the obvious modifications. Totality of labelling is given by Theorem 2 of [12], and totality of rootedness is given by replacing all occurrences of the labelling function with the rootedness function in the proof. \(\Box\)

An important consequence of having both squares as natural pushouts is that we have a symmetry giving invertibility of derivations. This symmetry is unique to this new approach to rooted graph transformation. In Bak’s approach (Appendix D.8), derivations are not, in general, invertible (Figure 2.1). In Bak’s system, the intermediate graph \(D\) must not have a root if we want to invert the derivation.

**Corollary 2.1.** Derivations are \textbf{invertible}. That is \(G \Rightarrow_r H\) iff \(H \Rightarrow_r^{-1} G\).

**Proof.** By the last theorem, \(G \Rightarrow_r H\) means we have a match \(g : L \rightarrow G\), and a comatch \(h : R \rightarrow H\), and so by symmetry, we have the result. \(\Box\)

It absolutely cannot be overstated how important invertibility is! It will be the foundation of so many arguments within the rest of the theory and examples, in particular when it comes to language recognition. Moreover, if it weren’t for invertibility, there is no way that we could perform an encoding in Chapter 5 to recover the Critical Pair Lemma!

Finally, we can now show our root node invariance result. By comparison, in Bak’s system, non-root nodes could be matched against root nodes.

**Lemma 2.5.** Let \(G\) be a \textbf{TLRG}, and \(r = <L \leftarrow K \rightarrow R>\) a rule. Then root nodes in \(L\) can only be matched against root nodes in \(G\), and similarly for non-root nodes.

**Proof.** Immediate from the definitions. \(\Box\)

**Corollary 2.2.** Let \(G\) be a \textbf{TLRG}, and \(r = <L \leftarrow K \rightarrow R>\) a rule such that \(L\) and \(R\) both contain \(k\) root nodes, for some fixed \(k \in \mathbb{N}\). Then any \textbf{TLRG} \(H\) derived from \(G\) using \(r\) contains \(n\) root nodes iff \(G\) contains \(n\) root nodes.
Proof. By Lemma 2.5 (non-)roots in $L$ can only be identified with (non-)roots in $G$, and by symmetry the same for $R$ in $H$. By Theorem 2.1, NDPO existence corresponds to Definition 2.13, so, $|p^{-1}_G(\{1\})| = |p^{-1}_H(\{1\})|$. \hfill \qed

### 2.4 Equivalence of Rules

We now consider equivalence of rules, starting by formalising what it means to say that two rules are isomorphic, and then we will show that we can find a normal form for rules, unique up to isomorphism. In the next sections, we will see these notions of equivalence can be extended to GT systems in a natural way. Derivation compatibility will prove to be an important notion.

**Definition 2.17.** Given rules $r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle$, $r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle$. We call $r_1$ and $r_2$ isomorphic iff there exists isomorphisms $f : L_1 \to L_2$, $g : R_1 \to R_2$ such that $f|_{K_1} = g|_{K_1}$ and $f(K_1) = K_2$. Write $r_1 \cong r_2$.

The above notion of rule isomorphism is an equivalence, and gives rise to abstract rules $[r]$. Moreover, the collection of all abstract rules over some fixed alphabet forms a countable set. This is a nice little result, since the collection of all rules is actually not a set.

**Definition 2.18.** Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$, define its normal form $r \downarrow = \langle L \leftarrow K' \rightarrow R \rangle$ where $K' = (V_K, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. We say two rules $r_1$, $r_2$ are normalisation equivalent iff $r_1 \downarrow \cong r_2 \downarrow$. We write $r_1 \asymp r_2$.

Clearly, this gives us a coarser notion of equivalence for rules than the notion of isomorphism. This notation of equivalence is important because it has the property that all concrete representatives behave the same, in the sense of Theorem 2.2.

**Example 2.4.** Consider the rules over $\{\square, \bigtriangleup\}$ as given in Figure 2.5. Clearly $r_1$ and $r_2$ are isomorphic, but $r_3$ is not isomorphic to either. Rule $r_1$ has normal form $r'_1$.

![Example (Non-)Isomorphic Rules](figure2.5)

**Theorem 2.2** (Compatible Derivations). Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and its normal form $r \downarrow = \langle L \leftarrow K' \rightarrow R \rangle$, then for all TLRGs $G$, $H$, $G \Rightarrow_r H$ iff $G \Rightarrow_{r \downarrow} H$. 

2.5. GRAPH TRANSFORMATION SYSTEMS

Figure 2.6: Derivations Diagram

Proof. Consider some fixed graph $G$. The set of injective morphisms $g : L \to G$ satisfying the dangling condition must be identical for both rules since $L$ is the same and so is $V_K$. Then, by the explicit construction of $H$ given by Definition 2.13, $G \Rightarrow_{r,g} H$ iff $G \Rightarrow_{r,g} H$. \qed

Remark 2.7. Normal forms for rules is not actually a new observation, and is the foundation of rule schemata in GP2 [53]. Moreover, maximising the number of edges in the interface of rules leads to a reduction of the number of critical pairs (Section D.7) of a GT system [70].

2.5 Graph Transformation Systems

We can now define graph transformation systems using our new definitions of graphs and rules. Next, we will look at different notions of equivalence for such system. This will become important in Chapter 5, when we rely on derivation compatibility.

Definition 2.19. A graph transformation system $T = (\mathcal{L}, \mathcal{R})$, consists of a label alphabet $\mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E)$, and a finite set $\mathcal{R}$ of rules over $\mathcal{L}$.

Definition 2.20. Given a graph transformation system $T = (\mathcal{L}, \mathcal{R})$, we define the inverse system $T^{-1} = (\mathcal{L}, \mathcal{R}^{-1})$ where $\mathcal{R}^{-1} = \{r^{-1} \mid r \in \mathcal{R}\}$.

Definition 2.21. Given a graph transformation system $T = (\mathcal{L}, \mathcal{R})$, a subalphabet of non-terminals $\mathcal{N}$, and a start graph $S$ over $\mathcal{L}$, then a graph grammar is the system $GGG = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$.

A graph is terminally labelled whenever none of the nodes or edges are labelled by non-terminals. Clearly the property of being terminally labelled is preserved under isomorphism, so we can lift it to a property of abstract graphs. We can then define the language generated by a grammar as follows:

Definition 2.22. Given a graph grammar $GGG$ as defined above, we say that a graph $G$ is terminally labelled iff $l(V) \cap \mathcal{N}_V = \emptyset$ and $m(E) \cap \mathcal{N}_E = \emptyset$. Thus, we can define the graph language generated by $GGG$:

$$L(GG) = \{[G] \mid S \Rightarrow^*_\mathcal{R} G, G \text{ terminally labelled}\}$$

One can think of graph transformation systems in terms of grammars that define languages. If they are terminating, then membership testing is decidable, but in general, non-deterministic in the sense that a deterministic
algorithm must backtrack if it produces a normal form not equal to the start graph, to determine if another derivation sequence could have reached it. If the system is confluent too, then the algorithm becomes deterministic.

**Definition 2.23** (Language Recognition). \( T = (\mathcal{L}, \mathcal{R}) \) recognises a language \( \mathcal{L} \) over \( \mathcal{P} \subseteq \mathcal{L} \) iff \( \exists [S] \in \mathcal{L} \) such that \( \forall [G] \in \mathcal{G}^\mathcal{P}, [G] \in \mathcal{L} \) iff \( G \Rightarrow^*_\mathcal{R} S \).

**Theorem 2.3** (Membership Test). Given a grammar \( G = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S) \), \([G] \in \mathcal{L}(G)\) iff \( G \Rightarrow^*_\mathcal{R} -1 S \) and \( G \) is terminally labelled. That is, \((\mathcal{L}, \mathcal{R}^{-1})\) recognises \( \mathcal{L}(G) \) over \( \mathcal{L} \setminus \mathcal{N} \).

**Proof.** The key point is that rules and derivations are invertible, which means that if \( S \) can be derived from \( G \) using the reverse rules, then \( G \) can be derived from \( S \) using the original rules so is in the language. If \( S \) cannot be derived from \( G \), then \( G \) cannot be in the language since that would imply there was a derivation sequence from \( S \) to \( G \) which we could invert to give a contradiction. By construction, we only consider terminally labelled \( G \). \( \square \)

**Remark 2.8.** If \( T = (\mathcal{L}, \mathcal{R}) \) recognizing a language \( \mathcal{L} \) is terminating, then membership testing for \( \mathcal{L} \) is decidable. Moreover, if \( T \) is also confluent, then recognition is efficient in the sense that we need not backtrack. The algorithm is simply to compute a normal form, and test if it is isomorphic to \( S \).

Just like in Section 1.6, we can define the induced ARS of a GT system.

**Definition 2.24.** Let \( T = (\mathcal{L}, \mathcal{R}) \) be a GT system. Then \( (\mathcal{G}^\mathcal{P}(\mathcal{L}), \rightarrow_\mathcal{R}) \) is the induced ARS defined by \( \forall [G], [H] \in \mathcal{G}^\mathcal{P}(\mathcal{L}), [G] \rightarrow_\mathcal{R} [H] \) iff \( G \Rightarrow_\mathcal{R} H \). We may also use the notation \( \rightarrow_T \) in place of \( \rightarrow_\mathcal{R} \) whenever the ambient label alphabet is not clear.

**Lemma 2.6.** Consider the ARS \( (\mathcal{G}^\mathcal{P}(\mathcal{L}), \rightarrow_\mathcal{R}) \) induced by a GT system. Then \( \rightarrow_\mathcal{R} \) is a binary relation (Definition A.14) on \( \mathcal{G}^\mathcal{P}(\mathcal{L}) \). Moreover, it is finitely branching (Definition C.6) and decidable (Definition A.29).

**Remark 2.9.** This does not, in general, imply that \( \rightarrow_\mathcal{R}^* \) is decidable. We say that \( T \) is (locally) confluent (terminating) iff its induced ARS is.

**Lemma 2.7.** A GT system is terminating iff there is a monotone (Definition A.27) embedding from the induced ARS into \((\mathbb{N}, >)\).

**Proof.** By Lemma 2.6, the induced ARS of a GT system is finitely branching, so the result follows from Lemma C.3 \( \square \)

When we were thinking about reversing the rules to produce a membership test, we were thinking about them as a computation device. There was an input graph, and output graph(s) (and possibly non-termination). We defined a semantic function in Example 1.8. We can do this again in our new setting.
Definition 2.25. Given $T = (L, R)$, define state space $\Sigma = G^\oplus(L) \cup \{ \bot \}$ and induced ARS $(G^\oplus(L), \rightarrow_R)$. Define the semantic function $f_T : G^\oplus(L) \rightarrow \mathcal{P}(\Sigma)$ by $f_T([G]) = \{ [H] \mid [H] \text{ is a normal form of } [G] \text{ w.r.t. } \rightarrow_R \} \cup \{ \bot \mid \text{there is an infinite reduction sequence starting from } [G] \text{ w.r.t. } \rightarrow_R \}$ and $f_T(\bot) = \{ \bot \}$.

Proposition 2.5. Given $T = (L, R)$, then $\forall [G] \in G^\oplus(L), |f_T([G])| \geq 1$. Moreover, if $T$ is confluent and terminating, then $\forall [G] \in G^\oplus(L), |f_T([G])| = 1$.

Proposition 2.6. Given $T = (L, R)$, in general it is undecidable if $[H] \in f_T([G])$ for any TLRG $G, H$, and also if $\bot \in f_T([G])$.

2.6 Equivalence of GT Systems

Building on the work from Section 2.4, we can ask when two graph transformations are equivalent, or rather, when they are distinct. We will give various notions of equivalence, and show there is a hierarchy of inclusion, as each notion is more and more general than the last.

Definition 2.26. Two GT systems $T_1 = (L, R_1)$, $T_2 = (L, R_2)$ are:

1. Isomorphic ($T_1 \cong T_2$) iff their quotients (Definition A.21) under rule isomorphism (Definition 2.17) are equal ($R_1/\cong = R_2/\cong$);
2. Normalisation equivalent ($T_1 \simeq_N T_2$) iff their quotients under rule equivalence (Definition 2.18) are equal ($R_1/\simeq = R_2/\simeq$);
3. Step-wise equivalent ($T_1 \simeq_S T_2$) iff their induced ARSs (Definition 2.24) are equal ($\rightarrow_{T_1} = \rightarrow_{T_2}$);
4. Semantically equivalent ($T_1 \simeq_F T_2$) iff their semantic functions are equal ($f_{T_1} = f_{T_2}$).

Proposition 2.7. Each of the above notions are equivalences.

Remark 2.10. A way of looking at step-wise equivalence is to say that it is a formalisation of what it means to have derivation compatibility in the sense of Theorem 2.2.

Proposition 2.8. This notion of isomorphism gives rise to abstract graph transformation systems $[T]$ over some fixed label alphabet $L$. Let $\mathcal{T}(L)$ denote the collection of all such classes. Then, $\mathcal{T}(L)$ is a countable set.

Remark 2.11. Clearly isomorphism and normalisation equivalence are well behaved. That is, it is decidable to check if two GT systems share the same class. The same is not true of semantic equivalence.

Theorem 2.4 (GT System Equivalence). GT system isomorphism is finer than normalisation equivalence is finer than step-wise equivalence is finer than semantic equivalence. That is, $T_1 \cong T_2$ implies $T_1 \simeq_N T_2$ implies $T_1 \simeq_S T_2$ implies $T_1 \simeq_F T_2$. Moreover, there exist GT systems in which the implication direction is strictly one-way.
Proof. Let $T_1, T_2$ be GT systems over some $\mathcal{L}$, with rule sets $R_1, R_2$. Within this proof, rules $r_1, r_2, r_3, r_4, r_5, r_6$ can be found in Figure 2.7.

Suppose $T_1 \cong T_2$. Then the $\cong$-classes of $R_1$ correspond to those of $R_2$. Clearly, if we find the normal form of each class, then the correspondence between these classes of normal forms is preserved. So $T_1 \simeq_N T_2$. To see the inclusion is strict, consider the two systems $(\mathcal{L}, \{r_1\}), (\mathcal{L}, \{r_2\})$. They are non-isomorphic, but are normalisation equivalent.

Next suppose $T_1 \simeq_N T_2$. Then by Theorem 2.2, the choice of representative element from each class is irrelevant, that is, the derivations possible are identical. Now, since the $\simeq_N$-classes of $R_1$ and $R_2$ are identical, combining all possible derivations from the classes leaves us with identical possible derivations for each. Thus, it is immediate that the induced ARS is identical. To see the inclusion is strict, consider the two systems $(\mathcal{L}, \{r_3\}), (\mathcal{L}, \{r_3, r_4\})$. They are not normalisation equivalent, but are step-wise.

Finally, suppose $T_1 \simeq_S T_2$. Then the induced ARS relations $\rightarrow_{R_1}, \rightarrow_{R_2}$ are equal, so clearly $f_{T_1} = f_{T_2}$. To see the inclusion is strict, consider the two systems $(\mathcal{L}, \{r_5\}), (\mathcal{L}, \{r_6\})$ are not step-wise equivalent since $r_5$ is always applicable with no effect, but $r_6$ is also always applicable, adding a new node. They are, however, semantically equivalent since their semantic functions both evaluate to $\{\bot\}$ on all inputs.

Thus, we have shown each forward implication, and that there exist cases where the reverse implication is false.

\begin{figure}[h]
\centering
\begin{tabular}{ccc}
$r_1$: & $\bullet$ & $\rightarrow$ & $\bullet$ \\
r_2: & $\emptyset$ & $\leftarrow$ & $\bullet$ & $\rightarrow$ & $\emptyset$ \\
r_3: & $\bullet$ & $\leftarrow$ & $\bullet$ & $\rightarrow$ & $\bullet$ \\
r_4: & $\bullet$ & $\leftarrow$ & $\emptyset$ & $\rightarrow$ & $\bullet$ \\
r_5: & $\emptyset$ & $\leftarrow$ & $\emptyset$ & $\rightarrow$ & $\emptyset$ \\
r_6: & $\emptyset$ & $\leftarrow$ & $\emptyset$ & $\rightarrow$ & $\bullet$
\end{tabular}
\caption{Example Rules Demonstrating Non-Equivalence}
\end{figure}

In general, we might be interested in more than proving just equivalence. That is, when does one GT system ‘refine’ the other. The (stepwise) refinement of programs was originally proposed by Dijkstra [73], [74] and Wirth [75]. Thinking in terms of GT systems, one may want to consider compatibility of the semantic function. Development of a refinement calculus that behaves properly with rooted GT systems remains open research.
Chapter 3

Fast Language Recognition

“Computer Science is no more about computers than astronomy is about telescopes.”

– Edsger W. Dijkstra, Unknown source

The Graph Matching Problem (Definition 1.6) and Rule Application Problem (Definition 1.7) can be considered in our setting. We will see that if we have an input graph with bounded degree (Definition E.7) and a bounded number of root nodes, and a finite set of ‘fast’ rules, then we can perform matching in constant time. Moreover, under mild constraints, for systems of fast rules, we need only analyse the derivation length to determine the time complexity of finding a normal form.

The language of all unlabelled trees is well-known to be expressible using classical graph transformation systems, using a single rule. The question of recognising trees ‘efficiently’ is less understood. We present a GT system that can test if a graph is a tree in linear time, given the input is of ‘bounded degree’: a new result for graph transformation systems. We also show that this can easily be adapted to recognise only full binary trees, and can be translated easily into GP2.

Part of this chapter was presented at CALCO 2019 as part of a co-authored paper looking at linear time algorithms in GP2 [22].

3.1 Complexity Theorems

We begin by defining the notion of a ‘fast’ rule. Intuitively, requiring every connected component of a rule to contain a root node is going to allow an implementation to both match each root node in the host graph in constant time, and then match the rest of the subgraphs around the root nodes quickly.
Definition 3.1. We call a rule $r = (L \leftarrow K \rightarrow R)$ **fast** iff every connected component (Definition E.5) of $L$ contains a root node. Additionally, we call a GT system **fast** iff all its rules are **fast**.

Just like in Lemma 1.2, we need to set up some assumptions about the complexity of various problems. We will assume that graphs are stored in a format such that the time complexities of various problems are as given in the table (Figure 3.1), based on Dodds’ assumptions (Figure 1.5) [20]. In our model, we have root nodes specified explicitly with a rootedness function, so we have updated the table to reflect this. Moreover, we have made our assumptions as weak as possible to see the results we need. An additional advantage of our assumptions is that they are realistic even when the alphabet is not finite, such as in GP2.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>label $l$</td>
<td>The set $Z$ of rooted nodes with label $l$.</td>
<td>$O(</td>
</tr>
<tr>
<td>label $l$</td>
<td>The set $Z$ of unrooted nodes with label $l$.</td>
<td>$O(</td>
</tr>
<tr>
<td>node $v$</td>
<td>Values $\text{deg}(v)$, $\text{indeg}(v)$, $\text{outdeg}(v)$.</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>node $v$</td>
<td>Set $X$ of edges with source $v$.</td>
<td>$O(</td>
</tr>
<tr>
<td>node $v$</td>
<td>Set $X$ of edges with target $v$.</td>
<td>$O(</td>
</tr>
<tr>
<td>graph $G$</td>
<td>$</td>
<td>V_G</td>
</tr>
</tbody>
</table>

Figure 3.1: Complexity Assumptions Table

Lemma 3.1. Given a TLRG $G$ of **bounded degree** containing a **bounded** number of root nodes, and a **fast** rule $r$, then the GMP (Definition 1.6) requires $O(|r|)$ time and produces $O(|r|)$ matches.

Proof. Under the same assumption as in Dodds’ Thesis [20, p. 39], this is easy to see, since there are only a constant number of subgraphs to consider. The full proof is a minor modification of Dodds’ proof. The most notable difference is that in our model, root nodes are not encoded as a special label, rather, via the rootedness function. This is handled in our complexity assumptions table. Then, because we have a bounded number of root nodes in $G$, we can then, ultimately, conclude $O(|r|)$ time rather than $O(|V_G|)$ time.

Lemma 3.2. Given a TLRG $G$ of **bounded degree**, a rule $r$, and an injective match $g$, then RAP (Definition 1.7) requires $O(|r|)$ time.

Proof. Obvious modifications of the proofs in Dodds’ Thesis.

Definition 3.2. Define $\text{mx} : \mathcal{P}(\mathbb{N}) \rightarrow \mathbb{N}$ by $\text{mx}(\emptyset) = 0$ and $\text{mx}(S) = \max(S)$ where $S \neq \emptyset$. We can then define $\text{mx}_\text{deg}(G) = \text{mx}(\{\text{deg}_G(v) \mid v \in V_G\})$.

Definition 3.3. Given an upper bound on the degree of nodes $N$ and an upper bound on the number of root nodes $M$, a GT system $T = (\mathcal{L}, \mathcal{R})$ is:

1. **Bounded degree preserving** iff for all TLRG $G, H$ such that $\text{mx}_\text{deg}(G) \leq N$, then $G \Rightarrow_R H$ implies that $\text{mx}_\text{deg}(H) \leq N$;
2. **Bounded roots preserving** iff for all TLRG $G, H$ such that $|p_G^{-1}(\{1\})| \leq M$, then $G \Rightarrow_R H$ implies that $|p_H^{-1}(\{1\})| \leq M$.

Luckily, these properties can be easily statically checked by looking at the shape of the rules (Lemma 3.3).

**Definition 3.4.** Given an upper bound on the degree of nodes $N \in \mathbb{N}$, a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is:

1. **Bounded degree preserving** iff $\forall v \in (V_R \setminus V_K), \deg_R(v) \leq N$ and $\forall v \in V_K, \deg_R(v) \leq \deg_L(v)$;
2. **Bounded roots preserving** iff $|p_R^{-1}(\{1\})| \leq |p_L^{-1}(\{1\})|$.

**Lemma 3.3.** A GT system is bounded degree preserving (bounded roots preserving) iff every rule is bounded degree preserving (bounded roots preserving).

**Proof.** The forwards direction can be seen by contradiction. Suppose that a rule were not preserving, then one could construct a graph were one could derive a graph of too large degree or too many root nodes. The reverse direction is obvious by Theorem 2.1 and Corollary 2.2.

**Remark 3.1.** Being bounded degree preserving does not mean that the maximum degree of a direct successor of a graph need be reduced, unlike for bounded root nodes. The rule that introduces a new node with a looped edge is bounded degree preserving, but one could imagine starting a derivation with the empty graph, then applying such a rule, taking the maximum degree of the graph from 0 to 1.

**Theorem 3.1** (Fast Derivations). Given a TLRG $G$ of bounded degree containing a bounded number of root nodes, and a fast GT system $T = (L, R)$, then one can decide in constant time the direct successors (Definition C.3) of $G$, up to isomorphism.

**Proof.** Combine the above lemmas. There is a constant number of rules to apply. For each rule, a bounded number of matches are produced in constant time, and then the RAP takes constant time for each match.

**Corollary 3.1** (Derivation-Complexity Lemma). Given $G$ as above, and $T = (L, R)$ a fast, bounded degree preserving, bounded roots preserving, terminating GT system with maximum derivation length $N \in \mathbb{N}$, then one can find a normal form (Definition C.3) of $G$ in $O(N)$ time.

**Proof.** By induction, the application of a rule satisfying the stated conditions will preserve the bound on the number of root nodes and the bound on the degree of the nodes. Thus, we have the result.

Thus, we have shown that if we have a set of rules as per Corollary 3.1, we need only consider the maximum length of derivations when reasoning about time complexity, as mentioned at the end of Section 1.4.
3.2 Recognising Trees in Linear Time

A **tree** is a graph containing a node from which there is a unique **directed path** to each node in the graph (Proposition 3.2). It is easy to see that it is possible to generate the collection of all trees by inductively adding new leaf nodes to the discrete graph of size one. That is, the language of all unlabelled trees can be specified by DPO-based graph grammar. The question of ‘efficient’ recognition of trees is less understood. In this section, we will show that it is possible to recognise the collection of all unlabelled trees in linear time. Moreover, at the end of the chapter, we will give an implementation of the algorithm in GP2, providing empirical evidence of the linear run time.

Writing a graph grammar that generates all unlabelled trees is straightforward. Simply start with the trivial tree (a single node), and arbitrarily add edges pointing to a new node, away from this start node.

![Figure 3.2: Tree Grammar Rules](image)

**Example 3.1** (Tree Grammar). Let $\mathcal{TREE} = (\mathcal{L}, \mathcal{N}, S, R)$ where:

1. $\mathcal{L} = (\{\Box\}, \{\Box\})$ where $\Box$ denotes the empty label;
2. $\mathcal{N} = (\emptyset, \emptyset)$;
3. $S$ be the graph with a single node labelled with $\Box$;
4. $R = \{r\}$.

To see that this grammar generates the set of all trees, we must show that every graph in the language is a tree, and then that every tree is in the language. This is easy to see by induction. △

Notice how the above construction has given us a decision procedure for testing if $[G] \in L(\mathcal{TREE})$:

**Lemma 3.4**. The GT system $((\{\Box\}, \{\Box\}), \{r^{-1}\})$ recognises $L(\mathcal{TREE})$, in the sense of Definition 2.23. That is, $[G] \in L(\mathcal{TREE})$ iff $G \Rightarrow_{r^{-1}} S$. Moreover, the system is confluent and terminating.

**Proof.** We first must comment that we are considering the GT system as a classical system as in Appendix D, without root nodes. As such, we must adjust Definition 2.23 accordingly. Now, by Theorem 2.3 (also adjusted accordingly), it is immediate that the GT system recognises the language of trees. To see that it is terminating is easy, since each rule is size reducing. Finally, local confluence can be seen via critical pair analysis (Section D.7) since there are no ‘critical pairs’. Thus, by Newman’s Lemma (Theorem C.3), the system is confluent. □
While the algorithm may be fast in the sense that we need not backtrack, the process of finding a match at each derivation step is slow, since we must consider the entire host graph when finding a match. Next, we will see that rooted graph transformation rules can actually recognise trees in linear time.

![Tree Recognition Rules](image)

Let \( \mathcal{L} = (\{\square, \triangle\}, \{\square\}) \) and \( \mathcal{R} = \{r_0, r_1, r_2\} \). We are going to show that \( \mathcal{R} \) induces a linear time algorithm for testing if a graph is a tree. Intuitively, this works by pushing a special node (a ‘root’ node) to the bottom of a branch, and then pruning. If we start with a tree and run this until we cannot do it anymore, we must be left with a single node. The triangle labels are necessary so that, in the case that the input graph is not a tree, we could ‘get stuck’ in a directed cycle.

**Example 3.2.** Figure 3.4 shows a reduction of a tree and non-trees. Note, in particular, how the triangles prevent an infinite sequence of derivations moving the root node around the 3-cycle.

![Example Reductions](image)

**Definition 3.5.** Given a graph \( G \), we define \( G^{\ominus} \) to be exactly \( G \), but with every node unrooted, and everything labelled by \( \square \). That is:

\[
G^{\ominus} = (V_G, E_G, s_G, t_G, V_G \times \{\square\}, E_G \times \{\square\}, V_G \times \{0\}).
\]

For the rest of this section, by ‘input graph’, we mean any TLRG containing exactly one ‘root’ node, with edges and vertices all labelled \( \square \). By ‘input tree’, we mean an ‘input graph’ that is also a tree (Definition [E.9]). Next, we show some technical lemmata, giving names to the results that are directly used in our proof of main result (Theorem [3.2]).
Lemma 3.5 (Tree Termination). The system \((\mathcal{L}, \mathcal{R})\) is terminating. Moreover, derivations have length at most \(2|V_G|\).

Proof. Let \(\#G = |V_G|, \square G = \{|v \in V_G \mid \ell_G(v) = \square\}\), for any TLRG \(G\). If \(G \Rightarrow^*_0 H\) or \(G \Rightarrow^*_1 H\), then \(\#G > \#H\) and \(\square G > \square H\). If \(G \Rightarrow^*_2 H\) then \(\#G = \#H\) and \(\square G > \square H\). Thus, termination follows from Lemma 2.7 since \(\varphi([G]) = \#G\) is a monotone embedding into \((\mathbb{N}, >)\). To see the last part, notice that \(\square G \leq \#G\) for all TLRGs \(G\), so the result is immediate since there are only \(2\#G\) natural numbers less than \(2\#G\).

Lemma 3.6. If \(G\) is a tree and \(G \Rightarrow^*_\mathcal{R} H\), then \(H\) is a tree. If \(G\) is not a tree and \(G \Rightarrow^*_\mathcal{R} H\), then \(H\) is not a tree.

Proof. Clearly, the application of \(r_2\) preserves structure. Suppose \(G\) is a tree. \(r_0\) or \(r_1\) are applicable iff node 2 is matched against a leaf node due to the dangling condition. Upon application, the leaf node and its incoming edge is removed. Clearly the result graph is still a tree.

If \(G\) is not a tree and one of \(r_0\) or \(r_1\) is applicable, then we can see the properties of not being a tree are preserved. That is, if \(G\) is not connected, \(H\) is certainly not connected. If \(G\) had parallel edges, due to the dangling condition, they must exist in \(G \setminus g(L)\), so \(H\) has parallel edges. Similarly, cycles are preserved. Finally, if \(G\) had a node with incoming degree greater than one, then \(H\) must too, since the node in \(G\) that is deleted in \(H\) had incoming degree one, and the degree of all other nodes is preserved.

Corollary 3.2 (Tree Preservation). If \(G\) is an input graph and \(G \Rightarrow^*_\mathcal{R} H\), then \(G\) is a tree iff \(H\) is a tree.

Proof. Induction.

Lemma 3.7. The system \((\mathcal{L}, \mathcal{R})\) preserves the number of root nodes in graphs. That is, if \(G \Rightarrow^*_\mathcal{R} H\), then \(|p_G^{-1}(\{1\})| = |p_H^{-1}(\{1\})|\).

Proof. Each of the rules \(r_0, r_1, r_2\) preserves the number of roots, so by Corollary 2.2 the number of root nodes is invariant across rule applications. Thus, the result holds by induction on derivation length.

Corollary 3.3. If \(G\) is an input graph and \(G \Rightarrow^*_\mathcal{R} H\), then \(H\) has exactly one root node. Moreover, there is no derivation sequence that derives the empty graph.

Proof. The first part follows from Lemma 3.7 since input graphs have exactly one root node. To see that the empty graph cannot be derived, notice that each derivation reduces \(\#G\) by at most one, and no rules are applicable when \(\#G = 1\).

Remark 3.2. In Bak and Plump’s model (and hence GP2), Corollary 3.3 is still true, however a more direct proof is needed. Since the root node in the LHS of each rule must be matched against a root node in the host graph, so the other non-roots can only be matched against non-roots.
Lemma 3.8. If $G$ is an input graph and $G \Rightarrow^*_{R} H$. Then, every $\triangle$-node in $H$ either has a child $\triangle$-node or a root-node child.

Proof. Clearly $G$ satisfies this, as there are no $\triangle$-nodes. We now proceed by induction. Suppose $G \Rightarrow^*_{R} H \Rightarrow^*_{R} H'$ where $H$ satisfies the condition. If $r_0$ or $r_1$ is applicable, we introduce no new $\triangle$-nodes. Additionally, in the case of $r_1$, any $\triangle$-node parents of the image 1 are preserved. So $H'$ satisfies the condition. Finally, if $r_2$ is applied, then the new $\triangle$-node has a root-node child, and the $\triangle$-nodes in $H' \setminus h(R)$ have the same children, so $H'$ satisfies the condition.

Corollary 3.4. Let $G$ be an input tree and $G \Rightarrow^*_{R} H$. Then the root-node in $H$ has no $\triangle$-node children.

Proof. By Corollary 3.3, $H$ has exactly one root node, and by Lemma 3.8, all chains of $\triangle$-nodes terminate with a root-node. If said root-node were to have a $\triangle$-node child, then we would have a cycle, which contradicts that $H$ is a tree (Corollary 3.2).

Lemma 3.9 (Tree Progress). Let $G$ be an input tree and $G \Rightarrow^*_{R} H$. Then, either $|V_H| = 1$ or $H$ is not in normal form.

Proof. By Corollary 3.3, $|V_H| \geq 1$. If $|V_G| = 1$, then $G$ is in normal form. Otherwise, either the root node has no children, or it has at least one $\square$-child. In the first case, $r_0$ must be applicable, and in the second, $r_2$.

Suppose $G \Rightarrow^*_{R} H$. If $|V_H| = 1$, then $H$ is in normal form by the proof to Corollary 3.3. Otherwise, by Corollary 3.2, $H$ is a tree and $|V_H| > 1$. Now, the root-node in $H$ (Corollary 3.3) must have a non-empty neighbourhood. If it has no children, then $r_0$ or $r_1$ must be applicable. Otherwise, $r_2$ must be applicable, since by Corollary 3.4, there must be a $\square$-node child. So $H$ is not in normal form.

We now present the main result of this chapter:

Theorem 3.2 (Tree Recognition). Given an input graph $G$, one may use the system $(L, R)$ from $G$ to find a normal form for $G$, say $H$. $H$ is the single root-node graph labelled by $\square$ iff $[G^\square] \in L(TREE)$. Moreover, for input graphs of bounded degree, we terminate in linear time.

Proof. By Lemma 3.5, our system is terminating and derivations have maximum length $2\#G$. By Corollary 3.2 and Lemma 3.9, $G$ is a tree iff we can derive the singleton tree without backtracking. Finally, by Corollary 3.1, the algorithm terminates in linear time, since our ruleset satisfies the necessary conditions.

We will return to this example in Section 5.4 where we see that the system is actually not confluent! We will show, however, that it is confluent ‘up to garbage’ on a class containing the ‘input trees’.
3.3 Recognising Full Binary Trees

In the previous section, we gave a grammar defining the language of all unlabelled trees, and showed that it was possible to adapt it to efficiently recognise the language by using a ‘root’ node. In this section, we will show that this technique can be adapted to recognise the language of all unlabelled full binary trees (Definition E.9) in linear time.

Let \( L = (\{\Box, \triangle\}, \{\Box\}) \), and \( R = \{r_0, r_1, r_2, r_3, r_4\} \). We are going to show that \( R \) induces a linear time algorithm for testing if a graph is a full binary tree (FBT). Intuitively, this works in a similar way to the reduction for trees, with the major difference that we only ‘prune’ \((r_0, r_1, r_2)\) whenever the root is in a position where it has exactly two children who are themselves leaf nodes. Rules \( r_3, r_4 \) ensure that we can always prune, so that we only get stuck in the case that our input was not a full binary tree. Similarly to the previous system for trees, the triangle labels are necessary so that, in the case that the input graph is not a tree, we could ‘get stuck’ in a directed cycle.

For the rest of this section, by ‘input graph’, we mean any TLRG containing exactly one ‘root’ node, with edges and vertices all labelled \( \Box \). By ‘input
3.4. RECOGNISING TREES IN GP 2

FBT', we mean an ‘input graph’ that is also a FBT. The proof of complexity and correctness is similar to the previous system, so we won’t give all the detail.

**Theorem 3.3** (FBT Recognition). Given an input graph $G$, one may use the system $(\mathcal{L}, \mathcal{R})$ from $G$ to find a normal form for $G$, say $H$. $H$ is the single root-node graph labelled by $\Box$ iff $G^\ominus$ is an unlabelled FBT. Moreover, for input graphs of bounded degree, we terminate in linear time.

**Proof.** The linear derivation length comes from the fact that if $r_0$ appears in a derivation sequence from an input graph, then it must be the last derivation, and it must derive a normal form. Moreover, if $r_5$ appears in a derivation sequence, then it must appear at the only at the very start of a derivation sequence from an input graph. The remaining $r_1, r_2, r_3$ must appear in the middle of a derivation sequence from an input graph, at most a linear number of times. All of the rules in our GT system are of the form such that Corollary 3.1 can be applied, so we can conclude linear time complexity.

For correctness, one must show the analogous results to Corollary 3.2 and Lemma 3.9. That is, that FBTs derive only FBTs, and non-FBTs derive only non-FBTs, and that progress is always made, given an input FBT, until we reach the singleton FBT. The proof detail is not dissimilar from the existing proofs for trees, once one has taken care of the existing observations we have made about the location of rules $r_0$ and $r_5$ in derivation sequences starting from an input graph. 

3.4 Recognising Trees in GP 2

Our algorithm from Section 3.2 can be implemented in GP2. The program (Figure 3.6) expects an arbitrary labelled input graph with every node coloured grey, no ‘root’ nodes, and no additional ‘marks’. It will fail if the input is not a tree. Given an input graph of bounded degree, it will always terminate in linear time with respect to (w.r.t.) the number of nodes in the input graph.

![Figure 3.6: GP 2 Implementation](image-url)
To see that the program is correct follows mostly from our existing proofs. Grey nodes encode the □ label, and blue nodes, △. The ‘init’ rule will fail if the input graph is empty, otherwise, it will make exactly one node rooted, in at most linear time. The ‘Reduce!’ step is then exactly our previous GT system, which we have shown to be correct, and terminates in linear time. Finally, the ‘Check’ step checks for garbage in linear time. There is no need to check the host graph is not equal to the empty graph (Corollary 3.3).

We have performed empirical benchmarking to verify the complexity of the program, testing it with linked lists, binary trees, grid graphs, and star graphs (Figure 3.7). Formal definitions of each of these graph classes can be found in Section E.2. We have exclusively used ‘perfect’ binary trees, and ‘square’ grid graphs in our testing.

![Graph Classes](image)

Figure 3.7: Graph Classes

Star Graphs are not of bounded degree, so we saw quadratic time complexity as expected. The other graphs are of bounded degree, thus we observed linear time complexity (Figure 3.8).

![Measured Performance](image)

Figure 3.8: Measured Performance

We thus have empirical evidence, that in some sense, our complexity assumptions must be realistic, and that our proofs are not contradicted by the results. It remains future work to formalise complexity, in general, for GP 2.
Chapter 4

Confluence Analysis I

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

– Donald Knuth, Notes on the van Emde Boas construction of priority deques: An instructive use of recursion (1977)

Efficient testing of language membership is an important problem in graph transformation [15], [76], [77]. Our GT system for testing if a graph is a tree is actually not confluent, but if the input is a tree, then it has exactly one normal form, so it was in some sense confluent. We can formalise this with the new notion of ‘confluence up to garbage’.

4.1 Definitions and Closedness

In this section, we shall be working with standard GT systems, as defined in Appendix D, but without relabelling. That is, all graphs are totally labelled, including interface graphs. All the results in this section will actually generalise to systems with relabelling, or the systems defined in Chapter 2.

Definition 4.1. Let $T = (\mathcal{L}, \mathcal{R})$ be a GT system, and $D \subseteq G(\mathcal{L})$ be a set of abstract graphs. Then, a graph $G$ is called garbage iff $[G] \notin D$.

Definition 4.2. Let $T = (\mathcal{L}, \mathcal{R})$, and $D \subseteq G(\mathcal{L})$. $D$ is closed under $T$ iff for all $G, H$ such that $G \Rightarrow_R H$, if $[G] \in D$ then $[H] \in D$. $D$ is strongly closed under $T$ iff we have $[G] \in D$ iff $[H] \in D$.

This set of abstract graphs $D$ represents the ‘good input’, and the ‘garbage’ is the graphs that are not in this set. $D$ need not be explicitly generated by a graph grammar. For example, it could be defined by some (monadic second-order [78]) logical formula.
There are a couple of immediately obvious results:

**Proposition 4.1.** Strong closedness $\Rightarrow$ closedness.

**Proposition 4.2.** Given $D \subseteq G(L)$ closed under $T = (L, R)$, then for all graphs $G, H$ such that $G \Rightarrow^*_R H$, if $[G] \in D$, then $[H] \in D$.

**Example 4.1.** Consider the reduction rules in Figure 4.1. The language of acyclic graphs is closed under the GT system $((\{\text{\#}\}, \{\text{\#}\}), \{r_1\})$, and the language of trees (forests) is strongly closed under $((\{\text{\#}\}, \{\text{\#}\}), \{r_2\})$. △

![Figure 4.1: Example Reduction Rules](image)

We can now define (local) confluence up to garbage, allowing us to say that, ignoring the garbage graphs, a system is (locally) confluent. Note that we previously called this property (local) confluence modulo garbage, however we have decided this name was confusing because we are not interested in confluence ‘modulo’ an equivalence relation. Rather, we are just choosing to ignore certain start graphs.

**Definition 4.3.** Let $T = (L, R), D \subseteq G(L)$. If for all graphs $G, H_1, H_2$, such that $[G] \in D$, if $H_1 \Leftarrow^*_R G \Rightarrow^*_R H_2$ implies that $H_1, H_2$ are joinable, then $T$ is locally confluent on $D$ (or locally confluent up to garbage w.r.t. $D$).

**Definition 4.4.** Let $T = (L, R), D \subseteq G(L)$. If for all graphs $G, H_1, H_2$, such that $[G] \in D$, if $H_1 \Leftarrow^*_R G \Rightarrow^*_R H_2$ implies that $H_1, H_2$ are joinable, then $T$ is confluent on $D$ (or confluent up to garbage w.r.t. $D$).

**Definition 4.5.** Let $T = (L, R), D \subseteq G(L)$. If there is no infinite derivation sequence $G_0 \Rightarrow^*_R G_1 \Rightarrow^*_R G_2 \Rightarrow^*_R \cdots$ such that $[G_0] \in D$, then $T$ is terminating on $D$ (or terminating up to garbage w.r.t. $D$).

**Lemma 4.1.** Let $T = (L, R), D \subseteq G(L), E \subseteq D$. Then (local) confluence (termination) on $D$ implies (local) confluence (termination) on $E$.

**Proof.** Immediate consequence of set inclusion! □

**Corollary 4.1.** Let $T = (L, R), D \subseteq G(L)$. Then (local) confluence (termination) implies (local) confluence (termination) on $D$.

**Proof.** Local confluence (confluence, termination) is exactly local confluence (confluence, termination) on $G(L)$. □

**Example 4.2.** Looking again at $r_1$ and $r_2$ from our first example, it is easy to see that $r_1$ is in fact terminating and confluent up to garbage on the language of acyclic graphs. Similarly, $r_2$ is terminating and confluent up to garbage on the language of trees. △
4.1. DEFINITIONS AND CLOSEDNESS

Lemma 4.2. Let $T = (\mathcal{L}, \mathcal{R}), \mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$. Then, if $\mathcal{D}$ is closed under $T$, the induced ARS $(\mathcal{D}, \rightarrow)$ where $[G] \rightarrow [H]$ iff $G \Rightarrow_R H$, is closed and well-defined. Moreover, it is (locally) confluent (terminating) whenever $T$ is, up to garbage with respect to $\mathcal{D}$.

Proof. Since $\mathcal{D}$ is closed under $T$, by Proposition 4.2, the induced ARS $(\mathcal{D}, \rightarrow)$ where $[G] \rightarrow [H]$ iff $G \Rightarrow_R H$ is closed, and clearly it is well-defined due to the uniqueness of derivations up to isomorphism. Clearly this induced ARS is (locally) confluent (terminating) if $T$ is (locally) confluent (terminating) up to garbage with respect to $\mathcal{D}$.

Definition 4.6 (Closedness Problem).

Input: A GT system $T = (\mathcal{L}, \mathcal{R})$ and a graph grammar $\mathcal{G}$ over $\mathcal{L}$.

Question: Is $\mathcal{L}(\mathcal{G})$ is closed under $T$?

Theorem 4.1 (Undecidable Closedness). The closedness problem is undecidable in general, even for terminating GT systems $T$ with only one rule, and $\mathcal{G}$ an edge replacement grammar (Section D.9).

Proof. This was shown in 1998 by Fradet and Le Métayer [79].

Remark 4.1. Closedness and language recognition has actually been considered before by Bakewell, Plump, and Runciman, in the context of languages specified by reduction systems without non-terminals [80, 81].

Theorem 4.2 (Newman-Garbage Lemma). Let $T = (\mathcal{L}, \mathcal{R}), \mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$. If $T$ is terminating on $\mathcal{D}$ and $\mathcal{D}$ is closed under $T$, then it is confluent on $\mathcal{D}$ iff it is locally confluent on $\mathcal{D}$.

Proof. By Lemma 4.2, the induced ARS $(\mathcal{D}, \rightarrow)$ is well-defined, closed, and terminating. Thus, by the original Newman’s Lemma (Theorem C.3), the induced ARS is confluent iff it is locally confluent, as required.

The requirement of closure is reasonable, even in the presence of termination on all graphs. Of course, there exists systems that are both locally confluent confluent up to garbage and terminating confluent up to garbage, which are not closed, but are confluent up to garbage, as in Example 4.3.

Example 4.3. Choose some $\mathcal{D}$ of size at least 2, and a graph $G$ which has a non-isomorphic successor under some GT system $T$. Suppose $\mathcal{D}$ is closed under $T$, $T$ locally confluent and $T$ terminating on $\mathcal{D}$. Then construct $\mathcal{E} = \mathcal{D} \setminus \{[H]\}$ where $H \not\cong G$ is a successor of $G$ under $T$. Then, $T$ is locally confluent and terminating on $\mathcal{E}$, but not closed.

However, in general, closure is needed. In Example 4.4, we show a system that is locally confluent up to garbage and terminating, yet is not confluent up to garbage.
Example 4.4. Let $\mathcal{L} = (\{\square\}, \{\square\})$ and $\mathcal{R} = \{r_0, r_1, r_2, r_3\}$, where the rules are given in Figure 4.2. Let $D = L(\text{TREE})$ from Section 3.2.

Then, $(\mathcal{L}, \mathcal{R})$ is locally confluent on $D$ and terminating on $G(\mathcal{L})$, however is not confluent on $D$. This was possible because we are not closed on $D$. An example non-joinable pair is given in Figure 4.3, generated by moving three steps away from the start graph. In our diagrams, unlabelled edges are understood to be $\square$-labelled.

![](image.png)

Figure 4.2: Example Non-Closed Rules

![](image.png)

Figure 4.3: Example Non-Joinable Pair

In this section, we have defined garbage, (local) confluence (termination) up to garbage, and closedness. We have shown that if we have closedness and termination up to garbage, then local confluence up to garbage implies confluence up to garbage, and an example where not having closedness fails, even in the presence of termination. Moreover, we have shown that closedness is undecidable in general.

### 4.2 Non-Garbage Critical Pairs

In 1970, Knuth and Bendix showed that confluence checking of terminating term rewriting systems is decidable [46]. Moreover, it suffices to compute all ‘critical pairs’ and check their joinability [47]–[49]. Unfortunately, for (terminating) graph transformation systems, confluence is not decidable (Theorem 1.1), and joinability of critical pairs does not imply local confluence. In 1993,
Plump showed that ‘strong joinability’ of all critical pairs is sufficient but not necessary to show local confluence \cite{16,45}. We have summarised these results in Section D.7.

We would like to generalise Theorem D.7 to allow us to determine when we have local confluence up to garbage. For this, we need to define a notion of subgraph closure and non-garbage critical pairs. In this section, we shall be working with standard GT systems, as defined in Appendix D, but without relabelling. That is, all graphs are totally labelled, including interface graphs.

**Definition 4.7.** Let $D \subseteq G(L)$ be a set of abstract graphs. Then $D$ is **subgraph closed** iff for all graphs $G, H$, such that $H \subseteq G$, if $[G] \in D$, then $[H] \in D$. The **subgraph closure** of $D$, denoted $\overline{D}$, is the smallest set containing $D$ that is **subgraph closed**.

**Proposition 4.3.** Given $D \subseteq G(L)$, $\overline{D}$ always exists, and is **unique**. Moreover, $D = \overline{D}$ iff $D$ is **subgraph closed**.

**Proof.** The key observations are that the subgraph relation is transitive, and each graph has only finitely many subgraphs. Clearly, the smallest possible set containing $D$ is just the union of all subgraphs of the elements of $D$, up to isomorphism. This is the unique subgraph closure of $D$.

**Remark 4.2.** $\overline{D}$ always exists, however it need not be decidable, even when $D$ is! It is not obvious what conditions on $D$ ensure that $\overline{D}$ is decidable. Interestingly, the classes of regular and context-free string languages are actually closed under substring closure \cite{82}.

**Example 4.5.** $\emptyset$ and $G(L)$ are subgraph closed.

**Example 4.6.** The language of discrete graphs is subgraph closed.

**Example 4.7.** The subgraph closure of the language of trees is the language of forests. The subgraph closure of the language connected graphs is the language of all graphs.

**Definition 4.8.** Let $T = (L, R)$, $D \subseteq G(L)$. A **critical pair** (Definition D.31) $H_1 \leftarrow G \Rightarrow H_2$ is **non-garbage** iff $[G] \in \overline{D}$.

**Lemma 4.3.** Given a GT system $T = (L, R)$ and $D \subseteq G(L)$, then there are only finitely many **non-garbage critical pairs** up to isomorphism.

**Proof.** By Theorem D.7 and Lemma D.3, there are only finitely many critical pairs for $T$, up to isomorphism, and there exists a terminating procedure for generating them. Thus, there are only finitely many non-garbage critical pairs up to isomorphism.

**Corollary 4.2.** Given a GT system $T = (L, R)$ and $D \subseteq G(L)$ such that $\overline{D}$ is **decidable**, then one can find the finitely many **non-garbage critical pairs** in **finite time**.
Proof. By Lemma 4.3, we have only finitely many pairs to consider, and we can generate them in finite time. Since $\mathcal{D}$ has a computable membership function, we can test if the start graph in each pair is garbage in finite time. 

Corollary 4.3. Let $T = (\mathcal{L}, \mathcal{R})$, $\mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$ be such that $T$ is terminating on $\mathcal{D}$ and $\mathcal{D}$ is decidable. Then, one can decide if all the non-garbage critical pairs are strongly joinable (Definition D.34).

Proof. By Corollary 4.2, we can find the finitely many pairs in finite time, and since $T$ is terminating on $\mathcal{D}$ and finitely branching (Lemma 1.1), both sides of each pair have only finitely many successors (Lemma C.1), thus we can test for strong joinability in finite time.

Lemma 4.4. Let $T = (\mathcal{L}, \mathcal{R})$, $\mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$. Then, the non-garbage critical pairs are complete. That is, for each pair of parallelly independent (Definition D.30) direct derivations, $H_1 \iff_{r_1, g_1} G \Rightarrow_{r_2, g_2} H_2$ such that $[G] \in \mathcal{D}$, there is a critical pair $P_1 \iff_{r_1, o_1} K \Rightarrow_{r_2, o_2} P_2$ with extension diagrams (1), (2), and an inclusion morphism $m : K \to G$.

\[
\begin{align*}
P_1 & \iff K \Rightarrow P_2 \\
\downarrow \quad (1) & \downarrow \quad (2) & \downarrow \\
H_1 & \iff G \Rightarrow H_2
\end{align*}
\]

Figure 4.4: Pair Factorisation Diagram

Proof. By Lemma 6.22 in [5], critical pairs are complete when $\mathcal{D} = \mathcal{G}(\mathcal{L})$. If we only consider derivations from start graphs $G$ such that $[G] \in \mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$, clearly all factorings with critical pairs are such that $K$ can be embedded into $G$, so $[K] \in \mathcal{D}$. Thus, the non-garbage critical pairs are complete.

Theorem 4.3 (Non-Garbage Critical Pair Lemma). Let $T = (\mathcal{L}, \mathcal{R})$, $\mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$. If all its non-garbage critical pairs are strongly joinable, then $T$ is locally confluent on $\mathcal{D}$.

Proof. By the proof of Theorem 6.28 in [5], strong joinability of critical pairs implies local confluence due to completeness. But, the non-garbage critical pairs are complete with respect to $\mathcal{D}$, so we have the result.

Corollary 4.4. Let $T = (\mathcal{L}, \mathcal{R})$, $\mathcal{D} \subseteq \mathcal{G}(\mathcal{L})$. If $T$ is terminating on $\mathcal{D}$, $\mathcal{D}$ is closed under $T$, and all $T$’s non-garbage critical pairs are strongly joinable then $T$ is confluent on $\mathcal{D}$.

Proof. By the above theorem, $T$ is locally confluent up to garbage, so by the Newman-Garbage Lemma (Theorem 4.2), $T$ is confluent up to garbage as required.

Remark 4.3. Obviously, testing for local confluence up to garbage is undecidable in general, even when $\mathcal{D}$ is decidable and the system is terminating and strongly closed. What is remarkable though, is that local confluence up to garbage is actually undecidable general for a terminating non-length-increasing string rewriting systems and $\mathcal{D}$ a regular string language [83]!
4.3 Extended Flow Diagrams

In 1976, Farrow, Kennedy and Zucconi presented semi-structured flow graphs, defining a grammar with confluent reduction rules \[23\]. Plump has considered a restricted version of this language: extended flow diagrams (EFDs) \[45\]. The reduction rules for extended flow diagrams efficiently recognise the EFDs, despite not being confluent. That is, we have an efficient mechanism for testing for language membership, since we need not ‘backtrack’, just like in Theorem 3.2.

**Definition 4.9 (Efficient Recognition).** \( T = (\mathcal{L}, \mathcal{R}) \) efficiently recognises \( \mathcal{L} \) over \( \mathcal{P} \subseteq \mathcal{L} \) iff \( T \) recognises \( \mathcal{L} \), \( T \) is terminating, and \( T \) is confluent on \( \mathcal{L} \).

**Theorem 4.4 (Efficient Recognition Correctness).** Given an GT system \( T = (\mathcal{L}, \mathcal{R}) \) efficiently recognising a language \( \mathcal{L} \) over \( \mathcal{P} \subseteq \mathcal{L} \) and an input graph \( G \) over \( \mathcal{P} \), the following algorithm is correct: Compute a normal form of \( G \) by deriving successor graphs using \( T \) as long as possible. If the result graph is isomorphic to \( S \), the input graph is in the language. Otherwise, the graph is not in the language.

**Proof.** Suppose \( G \) is not \( \mathcal{L} \). Then, since \( T \) is terminating our algorithm must be able to find a normal form of \( G \), say \( H \), and because \( T \) recognises \( \mathcal{L} \), it must be the case that \( H \) is not isomorphic to \( S \), and so the algorithm correctly decides that \( G \) is not in \( \mathcal{L} \).

Now, suppose that \( G \) is in \( \mathcal{L} \). Then, because \( T \) is terminating, as before, we must be able to derive some normal form, \( H \). But then, since \( T \) is both confluent on \( \mathcal{L} \) and recognises \( \mathcal{L} \), it must be the case that \( H \) is isomorphic to \( S \), and so the algorithm correctly decides that \( G \) is in \( \mathcal{L} \).

Figure 4.5: EFD Grammar Rules
Definition 4.10. The language of extended flow diagrams is generated by $EFD = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$ where $\mathcal{L}_V = \{\bullet, \square, \Diamond\}$, $\mathcal{L}_E = \{t, f, \square\}$, $\mathcal{N}_V = \mathcal{N}_E = \emptyset$, $\mathcal{R} = \{\text{seq, while, ddec, dec1, dec2}\}$, and $S = \leftrightarrow\square\leftrightarrow$.

In the figure, the shorthand notation with the numbers under the nodes places such nodes in the interface graph of the rules. We assume that the interface graphs are discrete (have no edges).

Lemma 4.5. Every directed cycle in an extended flow diagram contains a $t$-labelled edge

Proof. Induction. □

Theorem 4.5 (Efficient EFD Recognition). $EFD^{-1} = (\mathcal{L}, \mathcal{R}^{-1})$ efficiently recognises $L(EFD)$.

Proof. By Theorem 2.3, $EFD^{-1}$ recognises $L(EFD)$, and one can see that it is terminating since each rule is size reducing.

We now proceed by performing critical pair analysis on $EFD^{-1}$. There are ten critical pairs, all but one of which are strongly joinable apart from one (Figure 4.6). Now observe that Lemma 4.5 tells us that EFDs cannot contain such cycles. With this knowledge, we define $D$ to be all graphs such that directed cycles contain at least one $t$-labelled edge. Clearly, $D$ is subgraph closed, and then by our Non-Garbage Critical Pair Lemma (Theorem 4.3), we have that $EFD^{-1}$ is locally confluent on $D$.

Next, it is easy to see that $D$ is closed on $EFD^{-1}$, so we can use Newman-Garbage Lemma (Theorem 4.2) to conclude confluence on $D$ and thus, by Lemma 4.1, $EFD^{-1}$ is confluent on $L(EFD)$.

Thus, $EFD^{-1}$ efficiently recognises $L(EFD)$, as required. □
Chapter 5

Confluence Analysis II

“The development of mathematics towards greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules.”

– Kurt Gödel, Opening of the paper introducing the undecidability theorem (1931)

In the last chapter, we have seen that the notion of confluence up to garbage was sufficient to give efficient language recognition, however it is not yet known if these results transfer to the setting with root nodes and partial labelling. In this chapter, we will introduce definitions of sequential and parallel independence for such rules, and show that the (Non-Garbage) Critical Pair Lemma holds. We do this by showing that ‘conventional’ totally labelled systems can simulate relabelling and root nodes.

5.1 Encoding PLRGs

We can encode the graphs and morphisms given in Chapter 2 as totally labelled graphs and morphisms (in Appendix D). Both of these are locally small categories, and we denote them $PLRG(L)$ and $TLG(L)$ respectively. We will refer to the graph transformation systems from Chapter 2 as R-GT systems, and those from Appendix D without relabelling as T-GT systems.

Without loss of generality, we will assume that $L = (L_V, L_E)$ is an arbitrary label alphabet such that $L_V \cap L_E = \emptyset$ and $\{\square, 0, 1\} \cap (L_V \cup L_E) = \emptyset$.

In this section we will see that this encoding is a actually fully faithful functor and that we have derivation compatibility, in a similar sense to Remark 2.10.
Definition 5.1 (Object Encoding). Given $G = (V, E, s, t, l, m, p) \in \text{PLRG}(\mathcal{L})$, define $e(G) = (V', E', s', t', l', m')$ where:

1. $V' = V$;
2. $E' = E \times \{0\} \cup l^{-1}(L_E) \times \{1\} \cup p^{-1}(\{0, 1\}) \times \{2\}$;
3. $s'((e, 0)) = s(e)$ and $s'((v, i)) = v$ for $i = 1, 2$;
4. $t'((e, 0)) = t(e)$ and $t'((v, i)) = v$ for $i = 1, 2$;
5. $l'(v) = \square$;
6. $m'((e, 0)) = m(e)$, $m'((v, 1)) = l(v)$, and $m'((v, 2)) = p(v)$.

We denote by $e(X)$ where $X$ is some collection of objects (or their isomorphism classes), the image of the objects under $e$. In an abuse of notation, we will also denote the the label alphabet of the encoded objects by $e(\mathcal{L}) = \{\square, L_E \cup L_V \cup \{0, 1\}\}$.

Proposition 5.1. $e(G) \in \text{TLG}(e(\mathcal{L}))$.

Proof. Clearly each graph has exactly one well-defined encoding, and it is totally labelled by construction. \qed

Example 5.1. Let $\mathcal{L} = \{\{x\}, \{y, z\}\}$. Then Figure 5.1 shows an example PLRG and its encoding as a totally labelled graph. \hfill $\triangle$

![Figure 5.1: Example Encoded PLRG](image)

Definition 5.2 (Arrow Encoding). Given $g \in \text{Hom}_{\text{PLRG}(\mathcal{L})}(G, H)$, define $e(g) = (g'_V, g'_E)$ where:

1. $g'_V(v) = g_V(v)$;
2. $g'_E((e, 0)) = (g_E(e), 0)$ and $g'_E((v, i)) = (g_V(v), i)$ for $i = 1, 2$.

Proposition 5.2. $e(g) \in \text{Hom}_{\text{TLG}(e(\mathcal{L}))}(e(G), e(H))$.

Proof. By construction. \qed

Now that we have setup our encoding function, we are ready to give our first important technical lemma, allowing us to then show that derivations are compatible (Theorem 5.1).

Lemma 5.1. $e$ is a fully faithful functor $e : \text{PLRG}(\mathcal{L}) \rightarrow \text{TLG}(e(\mathcal{L}))$.

Proof. By Propositions 5.1 and 5.2 each graph and morphism has a well-defined encoding. The first axiom as given in Definition 5.3 is given by Proposition 5.2 and the remaining 2 axioms are almost immediate. Thus we have a functor.

To see fullness, suppose that there is a morphism between two encoded graphs. Since morphisms are required to preserve labels, it is not possible to
5.1. ENCODING PLRGS

have confusion between the edges (that would introduce extra morphisms).
That is, root-loops have to be mapped to root-loops, non-root-loops to non-
root-loops, l-label-loops to l-label-loops for all \( l \in \mathcal{L}_V \), and m-label-loops to
m-label-loops for all \( m \in \mathcal{L}_E \). We can then, decode such a morphism to a
morphism between the original graphs in the original setting.

Faithfulness is explicit from the construction of encoded morphisms. Each
morphism is mapped to a distinct encoded morphism.

\textbf{Lemma 5.2} (Dangling Condition Compatibility). Given a TLRG \( G \) and an
inclusion \( i : K \to L \) where \( L \) is a TLRG and \( K \) a PLRG, then the set of
injective morphisms \( L \to G \) satisfying the dangling condition w.r.t. \( i \) (in the
sense of R-GT systems) is in bijective correspondence via \( e \) with the set of in
injective morphisms \( e(L) \to e(G) \) satisfying the dangling condition w.r.t. \( e(i) \)
(in the sense of T-GT systems).

\textit{Proof.} First, we show that the injective morphisms are in bijective correspond-
ence. Suppose that \( g : L \to G \) is an injective morphism, then by construction,
\( e(g) : e(L) \to e(G) \) is injective. The reverse direction is a consequence of the
elementary result that fully faithful functors reflect monomorphisms.

Suppose again that \( g : L \to G \) is an injective morphism. Then \( g \) satisfies
the dangling condition w.r.t. \( i \) iff no edge in \( G \setminus g(L) \) is incident to a node
in \( g(L \setminus K) \) iff no edge in \( e(G \setminus e(g(L))) \) is incident to a node in \( e(g(L \setminus K)) \) iff no edge in \( e(G) \setminus e(g)(e(L)) \) is incident to a node in \( e(g)(e(L) \setminus e(K)) \) iff \( e(g) \)
satisfies the dangling condition w.r.t. \( e(i) \). The correctness of the movement
of \( e \) comes from the explicit constructions. \(\square\)

\textbf{Theorem 5.1} (Compatible Derivations). Given a rule \( r = (L \leftarrow K \to R) \)
from a R-GT system, then for all TLRG \( G,H \), \( G \Rightarrow_r H \) iff \( e(G) \Rightarrow e(r) \ e(H) \),
where \( e(r) \) encodes the rule in the obvious way.

\textit{Proof.} By \textbf{Lemma 5.2} \( e \) reflects and preserves injective morphisms satisfying
the dangling condition. But, it is an elementary result that a fully faithful
functor reflects all limits and colimits, so by Theorems \textbf{1.3} and \textbf{2.1} we have the result!

\(\square\)

We can define the track morphism and closedness in the same way as for
T-GT systems. We have compatibility of tracks and also of closedness. Note,
in particular, the special case when \( D = G(\mathcal{L}) \).

\textbf{Corollary 5.1} (Track Compatibility). \( e(tr_{G \Rightarrow_r H}) = tr_{e(G) \Rightarrow e(r) e(H)} \).

\textit{Proof.} Induction on derivation length using \textbf{Theorem 5.1}. \(\square\)

\textbf{Corollary 5.2} (Closedness Compatibility). Let \( T = (\mathcal{L}, \mathcal{R}) \) be some R-GT
system, and \( D \subseteq G^\mathcal{G}(\mathcal{L}) \). Then \( D \) is \textbf{closed} under \( T \) iff \( e(D) \) is \textbf{closed}
under \( e(T) = (e(\mathcal{L}), e(\mathcal{R})) \).
5.2 Local Church-Rosser and Independence

The purpose of this chapter is to define sequential and parallel independence for R-GT systems, and show they are compatible with the T-GT encoding. Moreover, Theorems 5.2 and 5.3 together, constitute the Local Church-Rosser Theorem, and so we have lifted this result into our new setting.

**Definition 5.3.** The derivations $G_1 \Rightarrow r_1, g_1 H \Rightarrow r_2, g_2 G_2$ are sequentially independent iff $(h_1(R_1) \cap g_2(L_2)) \subseteq (h_1(K_1) \cap g_2(K_2))$.

**Proposition 5.3.** We have sequential independence of derivations iff their encoding is sequentially independent in the sense of Definition D.29.

**Proof.** Definition bashing. □

**Theorem 5.2 (Sequential Independence).** If $G_1 \Rightarrow r_1, g_1 H \Rightarrow r_2, g_2 G_2$ are sequentially independent, then there exists a graph $H'$ and sequentially independent steps $G \Rightarrow r_2 H' \Rightarrow r_1 G_2$.

**Proof.** By combining Theorem 5.1 and Proposition 5.3 the result is immediate from Lemma D.1. □

**Definition 5.4.** The derivations $H_1 \Leftarrow r_1, g_1 G \Rightarrow r_2, g_2 H_2$ are parallelly independent iff $(g_1(L_1) \cap g_2(L_2)) \subseteq (g_1(K_1) \cap g_2(K_2))$.

**Remark 5.1.** The derivations $H_1 \Leftarrow r_1, g_1 G \Rightarrow r_2, g_2 H_2$ are parallelly independent iff $H_1 \Rightarrow r_1^{-1}, h_1 G \Rightarrow r_2, g_2 H_2$ are sequentially independent.

**Proposition 5.4.** We have parallel independence of derivations iff their encoding is parallelly independent in the sense of Definition D.30.

**Proof.** Definition bashing. □

**Theorem 5.3 (Parallel Independence).** If $H_1 \Leftarrow r_1, g_1 G \Rightarrow r_2, g_2 H_2$ are parallelly independent, then there exists a graph $G'$ and direct derivations $H_1 \Rightarrow r_2, G' \Leftarrow r_1 H_2$ with $G \Rightarrow r_1 H_1 \Rightarrow r_2 G'$ and $G \Rightarrow r_2 H_2 \Rightarrow r_1 G'$ sequentially independent.

**Proof.** By combining Theorem 5.1 and Proposition 5.3 the result is immediate from Lemma D.2. □

5.3 Confluence up to Garbage

The ultimate goal of this chapter is to recover the (Non-Garbage) Critical Pair Lemma for R-GT systems. We will begin this section by noting that all the definitions from Section 4.4 can be trivially re-formulated for R-GT systems. We will not give all the definitions and results again here, but we do need to state the relationship between (local) confluence (termination) of a R-GT system and that of its encoding.
5.3. CONFLUENCE UP TO GARBAGE

We start by noting compatibility of confluence and termination.

**Lemma 5.3 (Compatible Properties).** Given a R-GT system \( T = (\mathcal{L}, \mathcal{R}) \) and \( D \subseteq \mathcal{G}^\oplus(\mathcal{L}) \), then \( T \) is (locally) confluent (terminating) on \( D \) iff \( e(T) = (e(\mathcal{L}), e(\mathcal{R})) \) is (locally) confluent (terminating) on \( e(D) \).

Recall from Remark 2.4 that there are two obvious definitions of subgraph. The alternative definition that insists on the inclusion being undefinedness preserving will actually be most useful to us in this context, since we want to insist that all subgraphs of a TLRG are themselves TLRGs. We will thus use that definition of subgraph throughout the remainder of this chapter.

**Definition 5.5.** Let \( D \subseteq \mathcal{G}^\oplus(\mathcal{L}) \) be a set of abstract TLRGs. Then \( D \) is **subgraph closed** iff for all TLRG \( G, H \), such that \( H \subseteq G \), if \( [G] \in D \), then \([H] \in D \). The **subgraph closure** of \( D \), denoted \( \overline{D} \), is the smallest set containing \( D \) that is subgraph closed.

**Proposition 5.5.** Given \( D \subseteq \mathcal{G}^\oplus(\mathcal{L}) \), \( D \) always exists, and is unique. Moreover, \( D = \overline{D} \) if \( D \) is subgraph closed.

We now must setup the definition of a (non-garbage) critical pair. We will show that this definition is compatible across encoding, and also that (strong) joinability is.

**Definition 5.6.** A pair of direct derivations \( G_1 \xleftarrow{r_1,g_1} H \xrightarrow{r_2,g_2} G_2 \) is a **critical pair** iff:
1. \( H = g_1(L_1) \cup g_2(L_2) \);
2. The steps are not **parallelly independent**;
3. If \( r_1 = r_2 \) then \( g_1 \neq g_2 \).

**Definition 5.7.** A critical pair \( H_1 \xleftarrow{G} H \rightarrow H_2 \) is non-garbage w.r.t. \( D \) iff \( [G] \in D \).

**Lemma 5.4 (Compatible Non-Garbage Critical Pairs).** Given a R-GT system \( T = (\mathcal{L}, \mathcal{R}) \) and \( D \subseteq \mathcal{G}^\oplus(\mathcal{L}) \), a pair of direct derivations \( G_1 \xleftarrow{r_1,g_1} H \rightarrow_{r_2,g_2} G_2 \) is a non-garbage critical pair w.r.t. \( D \) iff its encoding via \( e \) is a non-garbage critical pair w.r.t. \( e(D) \) in the sense of a T-GT system.

**Proof.** Due to Proposition 5.4 and Lemma 5.1.

**Corollary 5.3 (Compatible Critical Pairs).** Given a R-GT system \( T = (\mathcal{L}, \mathcal{R}) \), a pair of direct derivations \( G_1 \xleftarrow{r_1,g_1} H \rightarrow_{r_2,g_2} G_2 \) is a critical pair iff its encoding via \( e \) is a non-garbage critical pair w.r.t. \( e(\mathcal{G}^\oplus(\mathcal{L})) \) in the sense of a T-GT system.

**Theorem 5.4 (Compatible Strong Joinability).** Given a R-GT system \( T = (\mathcal{L}, \mathcal{R}) \), a critical pair is (strongly) joinable iff its encoding is.
CHAPTER 5. CONFLUENCE ANALYSIS II

Proof. Due to Theorem 5.1, Corollary 5.1 and Lemma 5.4.

Just like in the previous case, there are only finitely many (non-garbage) critical pairs, and so long as the sub-graph closure of $D$ is decidable, we can generate them all.

**Lemma 5.5.** Given a R-GT system $T = (L, R)$ and $D \subseteq G^\oplus(L)$, then there are only finitely many non-garbage critical pairs up to isomorphism.

**Proof.** Lemma 4.3 combined with Lemma 5.4.

**Corollary 5.4.** Given a R-GT system $T = (L, R)$ and $D \subseteq G^\oplus(L)$ such that $D$ is decidable, then one can find the finitely many non-garbage critical pairs in finite time.

**Proof.** By Lemma 5.5 we have have only finitely many pairs to consider. Continue as in the proof of Corollary 4.2.

**Corollary 5.5.** Let $T = (L, R)$, $D \subseteq G^\oplus(L)$ be such that $T$ is terminating on $D$ and $D$ is decidable. Then, one can decide if all the non-garbage critical pairs are strongly joinable (Definition D.34).

**Proof.** By Corollary 5.4 we can find the finitely many pairs in finite time. Continue as in the proof of Corollary 4.3.

**Remark 5.2.** Clearly $e(G^\oplus(L))$ is always decidable, so it is always possible to generate all the genuine critical pairs of a R-GT system.

We are now, finally, ready to reproduce the (Non-Garbage) Critical Pair Lemma.

**Theorem 5.5 (Non-Garbage Critical Pair Lemma).** Let $T = (L, R)$, $D \subseteq G^\oplus(L)$. If all its non-garbage critical pairs are strongly joinable, then $T$ is locally confluent on $D$.

**Proof.** By compatibility Lemma 5.4 and Theorem 5.4, we can simply apply the original Non-Garbage Critical Pair Lemma (Theorem 4.3).

**Corollary 5.6.** Let $T = (L, R)$, $D \subseteq G^\oplus(L)$. If $T$ is terminating on $D$, $D$ is closed under $T$, and all $T$’s non-garbage critical pairs are strongly joinable then $T$ is confluent on $D$.

**Proof.** By the above theorem, $T$ is locally confluent up to garbage, so by the Newman-Garbage Lemma, $T$ is confluent up to garbage.

5.4 Tree Recognition Revisited

It is possible to rephrase the results from Section 3.2 in terms of our new notion of garbage, where the rules $r_0, r_1, r_2$ are from Figure 3.3.
5.4. TREE RECOGNITION REVISITED

**Proposition 5.6.** Let \( \mathcal{L} = (\{\square, \triangle\}, \{\square\}) \) and \( \mathcal{R} = \{r_0, r_1, r_2\} \). Then, \( D = \{G \in \mathcal{G}^B(\mathcal{L}) \mid |G^G| \in \mathcal{L}(\text{TREE}), |p_G^{-1}(\{1\})| = 1\} \) is strongly closed under \( T = (\mathcal{L}, \mathcal{R}) \) and \( T \) is confluent on \( E = \{G \in D \mid I_G(V_G) = \{\square\}\} \).

**Proof.** Strong closedness is due to Lemma 3.6 and confluence up to garbage due to Theorem 3.2.

One would hope that we could then perform non-garbage critical pair analysis w.r.t. \( D \) (where \( D \) is as in Proposition 5.6) in order to demonstrate local confluence on \( D \). It turns out that every non-garbage critical pair is joinable, but unfortunately, one of them is not strongly joinable (Figure 5.2), so we are unable to make any conclusion about local confluence up to garbage using the Non-Garbage Critical Pair Lemma (Theorem 5.5).

Thus, just like Plump’s original Critical Pair Lemma, strong joinability of (non-garbage) critical pairs is sufficient, but not necessary to imply local confluence (up to garbage). As discussed in the next chapter, it remains future work to develop stronger (non-garbage) critical pair analysis theorems.
Chapter 6

Conclusion

“In a sense, mathematics has been most advanced by those who distinguished themselves by intuition rather than by rigorous proofs.”

– Felix Klein, Unknown source

We have reviewed the current state of graph transformation, with a particular focus on the ‘injective DPO’ approach with relabelling and graph programming languages, establishing issues with the current approach to rooted graph transformation. We developed a new type of graph transformation system that supports relabelling and root nodes, but where derivations are invertible, and looked at a case study, showing that rooted graph transformation systems can recognise trees in linear time. This work on tree recognition was presented at CALCO 2019 [22]. We have also defined some notions of equivalence for our new type of graph transformation system, and briefly discussed a possible theory of refinement.

In Chapter 4, we have introduced the new notion of confluence up to garbage for graph transformation systems, that allows us to have confluence, except in the cases we do not care about. Moreover, we have shown that it is sufficient to only analyse the non-garbage critical pairs to establish confluence up to garbage, and if said system is closed and terminating up to garbage, then we may conclude confluence up to garbage. We have shown closedness is undecidable in general, and applied our results to see that EFDs can be recognised by a system that is confluent up to garbage. Finally, in Chapter 5, we recovered notions of independence of derivations in our new notion of rooted GT systems with relabelling, along with the Local Church-Rosser Theorem and (Non-Garbage) Critical Pair Lemma, by encoding our rooted systems as conventional systems via a fully faithful functor.
6.1 Evaluation

We regard this project as a success, having achieved our four original goals as detailed in the Executive Summary. Our first goal was to review rooted DPO graph transformation with relabelling. We have done this in Chapter 1, looking at labelled GT systems with the DPO approach with injective matching, and how relabelling and root nodes have been implemented, providing further detail in Appendices C and D. We also briefly reviewed graph languages and DPO-based graph programming languages.

Our second goal was to address the problem that the current theory of rooted graph transformation does not have invertible derivations. We have fixed this problem in Chapter 2 by defining rootedness using a partial function onto a two-point set rather than pointing graphs with root nodes. We have shown rule application corresponds to NDPOs, how Dodds’ complexity theory applies in our system, and briefly discussed the equivalence of and refinement of GT systems.

Our third goal was to show a new example of how rooted graph transformation can be applied. We showed a new result that the graph class of trees can be recognised by a rooted GT system in linear time, given an input graph of bounded degree. Moreover, we have given empirical evidence by implementing the algorithm in GP2 and collecting timing results. The results are in the CALCO 2019 post-proceedings [22].

Our final goal was to develop new confluence analysis theory. We have defined a new notion of confluence up to garbage and non-garbage critical pairs, and shown that it is sufficient to require strong joinability of only the non-garbage critical pairs to establish confluence up to garbage, applying this theory to EFDs. Moreover, we have recovered the Local Church-Rosser Theorem and (Non-Garbage) Critical Pair Lemma for our new notion of rooted graph transformation with relabelling. We look to publish this work.

6.2 Future Work

Developing a fully-fledged theory of correctness and refinement for (rooted) GT systems remains future work, extending the work from Section 2.6. Additionally, extending this notion to GP2, or other graph transformation based languages, and looking at the automated introduction of root nodes in order to improve time complexity remains open. Overcoming the restriction of host graphs to be of bounded degree in Theorem 3.1 remains open too.

Further exploring the relationship between (local) confluence up to garbage and closedness remains open work. In fact, confluence analysis of GT systems remains an underexplored area in general. Developing a stronger version of the
Non-Garbage Critical Pair Lemma that allows for the detection of persistent nodes that need not be identified in the joined graph would allow conclusions of confluence up to garbage where it was previously not determined, remains future work. It is also unclear under what conditions one can decide if a graph is in the subgraph closure of a language, specified by a grammar, or otherwise.

Additional future work in the foundations of our new theory of rooted graph transformation would be to continue to establish the foundational theorems. We have shown the Local Church-Rosser Theorem and Critical Pair Lemma, however, the Parallelism and Concurrency Theorems [21], which have applications in database systems [84], algebraic specifications [85], and logic programming [6], remain future work. One may also want to recover a formalism of the Extension Theorem, which we didn’t need when proving the Critical Pair Lemma [86].

Finally, it remains open research, to explore the overlap between graph transformation systems and the study of ‘reversible computation’ [87]. Our new foundations of rooted graph transformation allows for the specification of both efficient and reversible GT systems. Since graph transformation is a uniform way of expressing many problems in computer science, it is only natural that its applications in reversible computation is explored.
Appendix A

Basic Mathematical Notions

There is not time to give an account of an axiomatization of set theory. Instead, we will simply state that we are using Von Neumann-Bernays-Gödel Set Theory (NBG) \cite{88, 89}. For the most part, a naive approach will suffice. We will use the word ‘collection’ to refer, informally, to something that may or may not be a set. We will avoid the use of the word ‘class’ to mitigate confusion with equivalence relations. When it comes to the definition of decidable (for sets of graphs) we will tacitly assuming a Gödel numbering \cite{40}.

A.1 Sets I

We split the ‘Sets’ section into two halves. This section is derived from Chapters 2 and 3 of \cite{90} and Chapter 3 of \cite{91}.

Definition A.1. We let $\emptyset$ denote the empty set. If $A$ is a set, then we write $a \in A$ to say that $a$ ‘belongs to’ $A$. We say that $B$ is a subset of $A$, $B \subseteq A$ iff $\forall x \in B, x \in A$. We say $A = B$ iff $A \subseteq B$ and $B \subseteq A$.

Definition A.2. If $A, B$ are sets, then we define:

1. Set union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$;
2. Set intersection: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$;
3. Set difference: $A \setminus B = \{x \mid x \in A \text{ and } x \notin B\}$;
4. Cartesian product: $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$;
5. Power set: $\mathcal{P}(A) = \{X \mid X \subseteq A\}, \mathcal{P}_1(A) = \mathcal{P}(A) \setminus \emptyset$.

Definition A.3. We define the disjoint union of sets $A, B$ to be their coproduct in the category of sets (Example B.1). It can be convenient to define this as $A + B = (A \times \{1\}) \cup (B \times \{2\})$, or simply to assume that $A \cap B = \emptyset$ and define $A + B = A \cup B$.

Definition A.4. Let $\mathbb{N} = \{0, 1, 2, \ldots\}$, and $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$.
A.2 Functions

This section is derived from Chapters 2 and 3 of [90] and Chapter 1 of [92].

We use the conventional order of composition.

**Definition A.5.** Let $A, B$ be sets. A function $f$ from domain $A$ to codomain $B$ is a rule which assigns to each $a \in A$ a unique $b \in B$. We write $b = f(a)$, $f : A \to B$, and call $a$ the argument of $f$. Formally, a function from $A$ to $B$ is a subset of $A \times B$ such that for each $a \in A$ there is exactly one element $(a,b)$ in $f$.

**Definition A.6.** Let $A, B, C, D$ be sets. If $f : A \to B$, $g : C \to D$ are functions, then $f$ and $g$ are equal ($f = g$) iff they are equal as sets.

**Definition A.7.** Let $A, B, C$ be sets. If $f : A \to B$, $g : B \to C$ are functions, then we form a new function $(g \circ f) : A \to C$ the composite of $f$ and $g$ by the rule $(g \circ f)(a) = g(f(a))$.

**Proposition A.1.** Composition of functions is associative. That is, given $f : A \to B$, $g : B \to C$, $h : C \to D$, then $h \circ (g \circ f) = (h \circ g) \circ f$.

**Definition A.8.** For any set $A$, the identity function on $A$, $id_A : A \to A$ is defined by $\forall a \in A, id_A(a) = a$.

**Proposition A.2.** If $f : B \to A$, then $id_A \circ f = f$. If $g : A \to C$, $g \circ id_A = g$.

**Definition A.9.** Let $f : A \to B$ be a function. Then a function $g : B \to A$ is the inverse of $f$ iff $g \circ f = id_A$ and $f \circ g = id_B$.

**Proposition A.3.** Let $f : A \to B$ be a function. Then, if an inverse exists, it is unique, and is denoted $f^{-1} : B \to A$.

**Definition A.10.** Let $f : A \to B$ be a function. Then $f$ is injective iff $\forall a, b \in A, f(a) = f(b)$ implies $a = b$. $f$ is surjective iff $\forall a \in A, \exists b \in B, f(a) = b$. If $f$ satisfies both properties, then it is bijective.

**Lemma A.1.** A function has an inverse iff it is a bijection.

**Definition A.11.** Let $f : A \to B$ be a function, $X \subseteq A$, and $Y \subseteq B$. Then the image of $A$ under $f$ is $f(A) = \{f(a) \mid a \in A\} \subseteq B$, and the preimage of $B$ is $f^{-1}(B) = \{a \in A \mid f(a) \in B\} \subseteq A$.

**Remark A.1.** This does not imply the existence of an inverse, but if it does exist, then preimage of $f$ coincides with the image of $f^{-1}$.

**Definition A.12.** Let $f : A \to B$ be a function, and $X \subseteq A$. Then the restriction of $f$ to $X$ is $f|_X : X \to B$ is defined by $\forall x \in X, f|_X(x) = f(x)$.

**Definition A.13.** A partial function $f : A \to B$ is a subset $f$ of $A \times B$ such that there is at most one element $(a,b)$ in $f$. 
A.3 Binary Relations

This section is derived from Chapter 2 of [90], Chapter 1 of [92] and Appendix A of [48].

Definition A.14. Let \( A \) be a set. Then a binary relation on \( A \) is a subset \( R \subseteq A \times A \). For any \( a, b \in A \), we write \( aRb \) iff \( (a, b) \in R \).

Definition A.15. Let \( A \) be a set. Then, the identity relation on \( A \) is \( \iota_A = \{(a, a) \mid a \in A\} \), and the universal relation on \( A \) is \( \omega_A = A \times A \).

Definition A.16. Let \( A \) be a set. Then we call a binary relation \( R \) on \( A \) functional iff for any \( a, b, c \in A \), \( aRb \) and \( aRc \) implies \( b = c \).

Definition A.17. Let \( A \) be a set, and \( R, S \) be binary relations on \( A \). Then the composition of \( R \) and \( S \) is \( S \circ R = \{(x, y) \in A \times A \mid \exists z \in A \text{ with } xRz \text{ and } zSy\} \). Define \( R^0 = \iota_A \), and \( \forall n \in \mathbb{N}^+, R^n = R \circ R^{n-1} \).

Definition A.18. Let \( A \) be a set. Then the inverse of a binary relation \( R \) on \( A \) is \( R^{-1} = \{(b, a) \in A \times A \mid aRb\} \).

Proposition A.4. When considered as binary relations, functions and partial functions are functional. Moreover, the definitions of composition and inverses coincide.

Definition A.19. A binary relation \( R \) on \( A \) is:
1. Reflexive iff \( \iota_A \subseteq R \);
2. Irreflexive iff \( \iota_A \cap R = \emptyset \);
3. Symmetric iff \( R = R^{-1} \);
4. Antisymmetric iff \( R \cap R^{-1} \subseteq \iota_A \);
5. Transitive iff \( R \circ R \subseteq R \);
6. Connex iff \( \omega_A \setminus \iota_A \subseteq R \cup R^{-1} \).

Definition A.20. A binary relation is a preorder iff it is reflexive and transitive. A symmetric preorder is called an equivalence.

Proposition A.5. The classes \( [a] = \{b \in A \mid a \sim b\} \) of an equivalence \( \sim \) on \( A \) partition \( A \) into a union of pairwise disjoint non-empty subsets.

Definition A.21. Given an equivalence \( \sim \), define \( A/\sim = \{[a] \mid a \in A\} \).

A.4 Orders

This section is derived from Chapter 1 of [92] and Appendix A of [48].

Definition A.22. An antisymmetric preorder \( \leq \) on \( X \) is called a partial order, and we call \((X, \leq)\) a partially ordered set (poset).
Definition A.23. A partial order satisfying the connex property is called a total order, giving a totally ordered set.

Definition A.24. A strict order is an irreflexive, transitive relation.

Proposition A.6. Every partial order \( \leq \) induces a strict order \( \leq \setminus \iota \), and every strict order \( < \) induces a partial order \( < \cup \iota \).

Definition A.25. Let \((X, \leq)\) be a poset, and \(\emptyset \neq Y \subseteq X\). Then:
1. \(a \in Y\) is minimal iff \(\forall y \in Y, y \leq a\) implies \(y = a\);
2. \(b \in Y\) is the minimum iff \(\forall y \in b \leq y\);
3. \(c \in X\) is a lower bound for \(Y\) iff \(\forall y \in Y, c \leq y\).

Proposition A.7. Let \((X, \leq)\) be a poset, and \(\emptyset \neq Y \subseteq X\). Then every minimum element of \(Y\) is minimal, \(Y\) and has at most one minimum.

Definition A.26. We say that the poset \((X, \leq)\) satisfies the minimal condition (well-founded) iff every non-empty subset of \(X\) has a minimal element. If \(\leq\) is also a total order, then we say it is well-ordered.

Definition A.27. Let \((X, \leq_X)\), \((Y, \leq_Y)\) be posets. Then a function \(\varphi : X \rightarrow Y\) is called monotone iff \(a \leq_X b\) implies \(\varphi(a) \leq_Y \varphi(b)\).

A.5 Sets II

This section is derived from Chapter 7 of [93], Part I of [94], Chapter 8 of [95], and Chapter 1 of [39].

Theorem A.1 (Well-Ordered Sets). Every set can be well-ordered, and every well-ordered set is isomorphic to an ordinal (see [93] for details).

Proposition A.8. We define the cardinality of \(A (|A|)\), to be the least ordinal \(\kappa\) such that there is some bijection \(f : A \rightarrow \kappa\). Every set has unique cardinality. All sets with cardinality \(\leq\) to that of \(\mathbb{N}\) are countable.

Theorem A.2 (Countable Sets). The Cartesian product of two countable sets is countable, and a countable union of countable sets is countable. If \(A\) is finite, then \(\mathcal{P}(A)\) is finite. The set \(\mathcal{P}(\mathbb{N})\) is uncountable.

Definition A.28. A partial function \(f : \mathbb{N} \rightarrow \mathbb{N}\) is computable iff there exists a Turing Machine that computes \(f\) (see [95] for details).

Definition A.29. A countable set \(A \subseteq \mathbb{N}\) has characteristic function \(\chi_A : \mathbb{N} \rightarrow \{0, 1\}\) defined by \(\forall x \in \mathbb{N}, \chi_A(x) = 1\) iff \(x \in A\). \(A\) is decidable or recursive iff \(\chi_A\) is computable. Otherwise, \(A\) is undecidable.

Definition A.30. \(A \subseteq \mathbb{N}\) is semidecidable or recursively enumerable iff it is the domain of a computable function.
Appendix B

Category Theory

The definitions and theorems in this appendix are derived Chapter 1 of [96], Appendix A of [5], Chapter 1 of [91], and Chapter 1 of [97].

B.1 Foundations

Definition B.1. A category $\mathcal{C}$ consists of the following data:

1. Objects: $A$, $B$, $C$, ...
2. Arrows: $f$, $g$, $h$, ...
3. For each arrow $f$, there are given objects $\text{dom}(f)$, $\text{cod}(f)$, and we write $f : A \rightarrow B$ to indicate that $A = \text{dom}(f)$, $B = \text{cod}(f)$;
4. Given arrows $f : A \rightarrow B$, $g : B \rightarrow C$, $g \circ f$ is an arrow such that $A = \text{dom}(g \circ f)$, $C = \text{cod}(g \circ f)$;
5. For each object $A$ there is given an arrow $1_A : A \rightarrow A$;

such that for all arrows $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$, we have $h \circ (g \circ f) = (h \circ g) \circ f$ and $f \circ 1_A = f = 1_B \circ f$. We write $A \in \mathcal{C}$ when $A$ is an object in $\mathcal{C}$.

Definition B.2. We call an arrow $f : A \rightarrow B$ monic (epic) iff it is left (right) cancellative. We call $f$ an isomorphism and write $A \cong B$ iff $f$ admits a two-sided inverse.

Definition B.3. A category $\mathcal{C}$ is called small iff the collection of all arrows is a set. Otherwise, it is called large. $\mathcal{C}$ is called locally small iff the collection of arrows between any two objects ($\text{Hom}_{\mathcal{C}}(X,Y)$) is a set.

Proposition B.1. Every small category is locally small. Moreover, a locally small category is small iff the collection of all objects is a set.

Definition B.4. A category is finitary iff there are only finitely many subobjects up to isomorphism, where $A$ is a subobject of $B$ iff there is a monomorphism $A \rightarrow B$. 
Example B.1. The collection of all sets as objects and all functions between them as arrows forms a locally small category. The collection of all finite sets and functions between them is a locally small, finitary category. \(\triangle\)

Definition B.5. A functor \(F : C \to D\) between categories \(C, D\) is a mapping such that:

1. \(F(f : A \to B) = F(f) : F(A) \to F(B)\);
2. \(F(1_A) = 1_{F(A)}\);
3. \(F(g \circ f) = F(g) \circ F(f)\).

Definition B.6. A (covariant) functor \(F : C \to D\) between two locally small categories \(C, D\) is faithul (full) if the induced function \(F_{X,Y} : \text{Hom}_C(X, Y) \to \text{Hom}_D(F(X), F(Y))\) is injective (surjective) for all objects \(X, Y\) in \(C\).

B.2 Pushouts and Pullbacks

\(\begin{array}{ccc}
A & \longrightarrow & B \\
\downarrow & & \downarrow \\
C & \longrightarrow & D
\end{array}\)

\(\begin{array}{ccc}
A' & \longrightarrow & B \\
\downarrow & & \downarrow \\
C & \longrightarrow & D
\end{array}\)

\(\begin{array}{ccc}
A & \longrightarrow & B \\
\downarrow & & \downarrow \\
C & \longrightarrow & D
\end{array}\)

\(\begin{array}{ccc}
A' & \longrightarrow & B \\
\downarrow & & \downarrow \\
C & \longrightarrow & D
\end{array}\)

Figure B.1: Pushout and Pullback

Definition B.7. Given arrows \(A \to B\) and \(A \to C\), an object \(D\) together with arrows \(B \to D\) and \(C \to D\) is a pushout iff:

1. Commutativity: \(A \to B \to D = A \to C \to D\).
2. Universal property: For all arrows \(B \to D'\), \(C \to D'\) such that \(A \to B \to D' = A \to C \to D'\), there is a unique arrow \(D \to D'\) such that \(B \to D \to D' = B \to D'\) and \(C \to D \to D' = C \to D'\).

Definition B.8. Given arrows \(B \to D\) and \(C \to D\), an object \(A\) together with arrows \(A \to B\) and \(A \to C\) is a pullback iff:

1. Commutativity: \(A \to B \to D = A \to C \to D\).
2. Universal property: For all arrows \(A' \to B\), \(A' \to C\) such that \(A' \to B \to D = A' \to C \to D\), there is a unique arrow \(A' \to A\) such that \(A' \to A \to B = A' \to B\) and \(A' \to A \to C = A' \to C\).

Definition B.9. A pushout that is a pullback is called a natural pushout.

Theorem B.1 (Limit Uniqueness). If they exist, a pushout (pullback), \(D\) (\(A\)) are unique up isomorphism.

Definition B.10. Given arrows \(A \to B\) and \(B \to D\), a (natural) pushout complement is an object \(C\) together with arrows \(A \to C\) and \(C \to D\) such that the resulting square is a (natural) pushout.
Appendix C

Abstract Reduction Systems

The definitions and theorems in this appendix are derived from Chapter 2 of [48], Section 2.2 of [98], and Section 1.1 of [99].

C.1 Foundations

Definition C.1. An abstract reduction system (ARS) is a pair \((A, \rightarrow)\) where \(A\) is a set and \(\rightarrow\) a binary relation on \(A\).

Definition C.2. Let \((A, \rightarrow)\) be an ARS. We define the notation:

1. Composition: \(\stackrel{n}{\rightarrow} := \rightarrow^n (n \geq 0)\);
2. Transitive closure: \(\stackrel{\rightarrow}{=\downarrow} := \bigcup_{n \geq 1} \stackrel{n}{\rightarrow}\);
3. Reflexive transitive closure: \(\stackrel{\rightarrow}{\downarrow} := \stackrel{\rightarrow}{=\downarrow} \cup \stackrel{0}{\rightarrow}\);
4. Reflexive closure: \(\stackrel{\rightarrow}{\leftarrow} := \rightarrow \cup \leftarrow\);
5. Inverse: \(\leftarrow := \rightarrow^{-1}\);
6. Symmetric closure: \(\leftrightarrow := \rightarrow \cup \leftarrow\).

Remark C.1. It is usual, in practice, that \(\rightarrow\) is decidable. This does not imply that \(\stackrel{\rightarrow}{\downarrow}\) is decidable, only that \(\stackrel{n}{\rightarrow}\) is decidable.

Definition C.3. Let \((A, \rightarrow)\) be an ARS. We say that:

1. \(x\) is reducible iff there is a \(y\) s.t. \(x \rightarrow y\);
2. \(x\) is in normal form iff \(x\) is not reducible;
3. \(y\) is a normal form of \(x\) iff \(x \rightarrow^{*} y\) and \(y\) is in normal form. If \(x\) has a unique normal form, it is denoted \(x\downarrow\);
4. \(y\) is a successor to \(x\) iff \(x \rightarrow^{*} y\), and a direct successor iff \(x \rightarrow y\);
5. \(x\) and \(y\) are joinable iff there is a \(z\) s.t. \(x \rightarrow^{*} z \leftarrow^{*} y\). We write \(x \downarrow y\).
C.2 Termination

**Definition C.4.** Let \((A, \rightarrow)\) be an ARS. Then \(\rightarrow\) is called terminating iff there is no infinite descending chain \(x_0 \rightarrow x_1 \rightarrow \ldots\).

**Remark C.2.** Other texts call a terminating reduction uniformly terminating or Noetherian, or say it satisfies the descending chain condition.

The principle of Noetherian induction (well-founded induction) is a generalisation of induction from \((\mathbb{N}, >)\) to any terminating reduction system.

**Definition C.5.** Let \((A, \rightarrow)\) be an ARS, and \(P\) is some property of the elements of \(A\). Then the inference rule for Noetherian Induction is:

\[
\forall x \in A, (\forall y \in A, x \xrightarrow{\rightarrow} y \Rightarrow P(y)) \Rightarrow P(x)
\]

**Theorem C.1** (Noetherian Induction). Let \((A, \rightarrow)\) The following are equivalent for an ARS:

1. The principle of Noetherian induction holds;
2. \(\rightarrow\) is well-founded (Definition A.26);
3. \(\rightarrow\) is terminating (Definition C.4).

**Definition C.6.** Let \((A, \rightarrow)\) be an ARS. Then \(\rightarrow\) is called

1. Finitely branching iff each \(a\) has only finitely many direct successors;
2. Globally finite iff each \(a\) has only finitely many successors;
3. Acyclic iff there is no \(a\) such that \(a \xrightarrow{\rightarrow} a\).

**Lemma C.1.** Let \((A, \rightarrow)\) be an ARS. Then:

1. If \(\rightarrow\) is finitely branching and terminating, then it is globally finite;
2. If \(\rightarrow\) is acyclic and globally finite, then it is terminating;
3. \(\rightarrow\) is acyclic iff \(\xrightarrow{\rightarrow}\) is a strict order.

**Definition C.7.** Let \((A, \rightarrow)\) be an ARS. Then \(\rightarrow\) is called normalising iff every element has a normal form (Definition C.3).

**Lemma C.2.** Let \((A, \rightarrow)\) be an ARS. If \(\rightarrow\) is terminating, then every element has at least one normal form.

**Lemma C.3.** A finitely branching reduction terminates iff there is a monotone (Definition A.27) embedding into \((\mathbb{N}, >)\).

**Remark C.3.** The union of two terminating reductions need not be terminating.
Definition C.8. Let \((A, \to)\) be an ARS. Then \(\to\) is called:

1. **Church-Rosser** iff \(x \leftrightarrow y\) implies \(x \downarrow y\);
2. **Confluent** iff \(y_1 \leftarrow x \to y_2\) implies \(y_1 \downarrow y_2\);
3. **Semi-confluent** iff \(y_1 \leftarrow x \to y_2\) implies \(y_1 \downarrow y_2\);
4. **Locally confluent** iff \(y_1 \leftarrow x \to y_2\) implies \(y_1 \downarrow y_2\);
5. **Convergent** iff it is both confluent and terminating.

**Theorem C.2** (Church-Rosser). Let \((A, \to)\) be an ARS. Then, the following are equivalent:

1. \(\to\) is **Church-Rosser**;
2. \(\to\) is semi-confluent;
3. \(\to\) is confluent.

**Lemma C.4**. Let \((A, \to)\) be an ARS. If \(\to\) is **confluent**, then:

1. Every element has at most one **normal form**;
2. If \(\to\) **normalising**, then \(x \leftrightarrow y\) iff \(x \downarrow = y \downarrow\).

**Theorem C.3** (Newman’s Lemma). A **terminating** reduction is **confluent** iff it is **locally confluent**.
Appendix D

Graph Transformation

We give a quick introduction to the theory of algebraic graph transformation derived from my earlier literature review [100], which is in turn derived from [5]. This Appendix should be read along side Chapter 1, since definitions are split between that Chapter and this Appendix, such as the definition of an unlabelled graph (Section 1.1).

Compared to [100], we generalise to partially labelled graphs using [12] and [101], in that we allow relabelling of totally labelled graphs. The additional sections on pushouts and pullbacks and critical pair analysis is derived from [5] and [101], the section on rooted graphs from [26], and the section on edge replacement grammars from [102] and [103]. The proof of Theorem D.3 is by Habel and Plump [12], and of Theorem D.4 is given by the proof of Theorem 1.1 which is due to Plump [16], [44], [45].

D.1 Partially Labelled Graphs

Definition D.1. A label alphabet \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \) consists of finite sets of node labels \( \mathcal{L}_V \) and edge labels \( \mathcal{L}_E \).

Definition D.2. A concrete partially labelled graph over a label alphabet \( \mathcal{L} \) is a concrete graph equipped with two partial label maps \( l : V \rightarrow \mathcal{L}_V, m : E \rightarrow \mathcal{L}_E \): \( G = (V, E, s, t, l, m) \).

\[
\begin{array}{ccc}
\mathcal{L}_E & \xleftarrow{m} & E \xrightarrow{l} \xrightarrow{s} V \xrightarrow{l} \mathcal{L}_V
\end{array}
\]

Figure D.1: Partially Labelled Graph Diagram

Remark D.1. By this definition, we do not work with the free monoid on the alphabet, as in string rewriting systems. Nodes and edges are labelled exactly with the elements from the respective alphabets.

Definition D.3. We say that a partially labelled graph \( G \) is totally labelled iff \( l_G \) is total.
Definition D.4. Given a common $\mathcal{L}$, a partially labelled graph morphism $g : G \to H$ is a graph morphism on the underlying concrete graphs, with the extra constraint that labels must be preserved, if defined. That is:

1. $\forall e \in E_G, g_V(s_G(e)) = s_H(g_E(e));$ [Sources]
2. $\forall e \in E_G, g_V(t_G(e)) = t_H(g_E(e));$ [Targets]
3. $\forall e \in E_G, m_G(e) = m_H(g_E(e));$ [Edge Labels]
4. $\forall v \in l_G^{-1}(\mathcal{L}_V), l_G(v) = l_H(g_V(v)).$ [Node Labels]

Definition D.5. Given a common $\mathcal{L}$, a partially labelled graph morphism $g : G \to H$ is undefinedness preserving iff $l_G = l_H \circ g_V$.

Definition D.6. Given a common $\mathcal{L}$, a partially labelled graph morphism $g : G \to H$ is injective/surjective iff the underlying graph morphism is injective/surjective.

Definition D.7. Given a common $\mathcal{L}$, we say $H$ is a subgraph of $G$ iff there exists an inclusion morphism $H \hookrightarrow G$. This happens iff $V_H \subseteq V_G, E_H \subseteq E_G, s_H = s_G|_{E_H}, t_H = t_G|_{E_H}, m_H = m_G|_{E_G}, l_H \subseteq l_G$.

Remark D.2. Given a totally labelled graph $G$, and $H$ partially labelled. If there exists a surjective morphism $G \to H$, then $H$ is totally labelled.

Definition D.8. We say that graphs $G, H$ are isomorphic iff there exists an injective, surjective graph morphism $g : G \to H$ such that $g^{-1} : H \to G$ is a graph morphism. We write $G \cong H$, and call $g$ an isomorphism. This naturally gives rise to equivalence classes $[G]$: the countably many partially labelled abstract graphs over some fixed $\mathcal{L}$.

D.2 Typed Graphs

Definition D.9. A typed graph is the tuple $G_T = (G, \text{type}_G)$ where $G$ is an unlabelled graph, and $\text{type}_G$ is a graph morphism $G \to TG$ where $TG$ is an unlabelled graph called a type graph. The vertices and edges of $TG$ are called the node alphabet and edge alphabet.

Definition D.10. Given two typed graphs $G_T, H_T$, a typed graph morphism is an unlabelled graph morphism $f : G \to H$ such that $\text{type}_H \circ f = \text{type}_G$.

Theorem D.1 (Typed-Labelled Graph Correspondence). There is a bijective correspondence between the totally labelled graphs over some fixed label alphabet $\mathcal{L}$ and the typed graphs over $\mathcal{L}$. 
D.3 Pushouts and Pullbacks

The following propositions hold in the category of unlabelled (totally labelled) graphs, but not necessarily for partially labelled graphs.

**Proposition D.1.** Every pushout satisfies the following:

1. **No junk:** Each item in $D$ has a preimage in $B$ or $C$.
2. **No confusion:** If $A \to B$, $A \to C$ monic, then $B \to D$, $C \to D$ monic and an item from $B$ is merged in $D$ with an item from $C$ only if the items have a common preimage in $A$.

**Proposition D.2.** A pushout is natural if $A \to B$ is monic.

**Theorem D.2 (Limit Existence).** Pushouts, pushout complements, and pullbacks always exist.

D.4 Rules and Derivations

Let $\mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E)$ be the ambient label alphabet, and graphs be concrete.

**Definition D.11.** A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of totally labelled graphs $L, R$ over $\mathcal{L}$, the partially labelled graph $K$ over $\mathcal{L}$, and inclusions $K \rightarrow L$ and $K \leftarrow R$.

**Definition D.12.** We define the inverse rule to be $r^{-1} = \langle R \leftarrow K \rightarrow L \rangle$.

**Definition D.13.** If $r = \langle L \leftarrow K \rightarrow R \rangle$ is a rule, then $|r| = \max\{|L|, |R|\}$.

**Definition D.14.** Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and a totally labelled graph $G$, we say that an injective morphism $g : L \to G$ satisfies the dangling condition iff no edge in $G \setminus g(L)$ is incident to a node in $g(L \setminus K)$.

\[
\begin{array}{cc}
L & \leftarrow K & \rightarrow R \\
G & \leftarrow D & \rightarrow H \\
\downarrow g & \downarrow d & \downarrow h \\
\end{array}
\]

**Figure D.2: Direct Derivation**

**Definition D.15.** To apply a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ to some totally labelled graph $G$, find an injective graph morphism $g : L \to G$ satisfying the dangling condition, then:

1. Delete $g(L \setminus K)$ from $G$, and for each unlabelled node $v$ in $K$, make $g_V(v)$ unlabelled, giving the intermediate graph $D$;
2. Add disjointly $R \setminus K$ to $D$, keeping their labels, and for each unlabelled node $v$ in $K$, label $g_V(v)$ with $l_R(v)$, giving the result graph $H$.
APPENDIX D. GRAPH TRANSFORMATION

If the dangling condition fails, then the rule is not applicable using the match \( g \). We can exhaustively check all matches to determine applicability.

**Definition D.16.** We write \( G \Rightarrow_{r,g} M \) for a successful application of \( r \) to \( G \) using match \( g \), obtaining result \( M \cong H \). We call Figure D.2 a direct derivation, and the injective morphism \( h \) the comatch.

**Theorem D.3 (Derivation Theorem).** It turns out that deletions are natural pushout complements and gluings are natural pushouts in the category of partially labelled graphs. Moreover, direct derivations are natural double pushouts, \( D \) and \( H \) are unique up to isomorphism, and \( H \) is totally labelled. Moreover, derivations \( G \Rightarrow_{r,g} H \) are invertible.

**Definition D.17.** Given a rule set \( R \), we define \( R^{-1} = \{ r^{-1} \mid r \in R \} \).

**Definition D.18.** For a given set of rules \( R \), we write \( G \Rightarrow_{R} H \) iff \( H \) is derived from \( G \) in one or more direct derivations, and \( G \Rightarrow_{R}^{*} H \) iff \( G \cong H \) or \( G \Rightarrow_{R}^{*} H \).

### D.5 Transformation Systems

**Definition D.20.** A graph transformation system \( T = (\mathcal{L}, R) \), consists of a label alphabet \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \), and a finite set \( R \) of rules over \( \mathcal{L} \).

**Proposition D.3.** Given a graph transformation system \( T = (\mathcal{L}, R) \), then one can always decide if \( G \Rightarrow_{R} H \).

**Definition D.21.** Given a graph transformation system \( T = (\mathcal{L}, R) \), we define the inverse system \( T^{-1} = (\mathcal{L}, R^{-1}) \).

**Definition D.22.** Given a label alphabet \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \), \( \mathcal{P} = (\mathcal{P}_V, \mathcal{P}_E) \) is a subalphabet of \( \mathcal{L} \) iff \( \mathcal{P}_V \subseteq \mathcal{L}_V \) and \( \mathcal{P}_E \subseteq \mathcal{L}_E \). We define the other standard set operations pairwise too, such as union, intersection, and difference of alphabets.

**Definition D.23.** Given a graph transformation system \( T = (\mathcal{L}, R) \), a subalphabet of non-terminals \( \mathcal{N} \), and a start graph \( S \) over \( \mathcal{L} \), then a graph grammar is the system \( G = (\mathcal{L}, \mathcal{N}, R, S) \).

**Definition D.24.** Given a graph grammar \( G \) as defined above, we say that a graph \( G \) is terminally labelled iff \( l(V) \cap \mathcal{N}_V = \emptyset \) and \( m(E) \cap \mathcal{N}_E = \emptyset \). Thus, we can define the graph language generated by \( G \):

\[
L(G) = \{ [G] \mid S \Rightarrow_{\mathcal{R}}^{*} G, \text{G terminally labelled} \}
\]

**Proposition D.4.** Given a graph grammar \( G = (\mathcal{L}, \mathcal{N}, R, S) \), \( G \Rightarrow_{r} H \) iff \( H \Rightarrow_{r^{-1}} G \), for some \( r \in R \) (simply use the comatch). Moreover, \([G] \in L(G)\) iff \( G \Rightarrow_{\mathcal{R}^{-1}}^{*} S \) and \( G \) is terminally labelled.
Remark D.3. Graph languages need not be finite. In fact, graph grammars are as powerful as unrestricted string grammars. As such, many questions like if the language is empty, are undecidable in general.

D.6 Confluence and Termination

Let \( T = (\mathcal{L}, \mathcal{R}) \) be a graph transformation system.

Definition D.25. The graphs \( H_1, H_2 \) are joinable iff there is a graph \( M \) such that \( H_1 \Rightarrow^* \mathcal{R} M \Leftrightarrow^* \mathcal{R} H_2 \).

Definition D.26. \( T \) is locally confluent iff for all graphs \( G, H_1, H_2 \) such that \( H_1 \Leftarrow^* \mathcal{R} G \Rightarrow^* \mathcal{R} H_2 \), \( H_1 \) and \( H_2 \) are joinable.

Definition D.27. \( T \) is confluent iff for all graphs \( G, H_1, H_2 \) such that \( H_1 \Leftarrow^* \mathcal{R} G \Rightarrow^* \mathcal{R} H_2 \), \( H_1 \) and \( H_2 \) are joinable.

Definition D.28. \( T \) is terminating iff there is no infinite derivation sequence \( G_0 \Rightarrow^* \mathcal{R} G_1 \Rightarrow^* \mathcal{R} G_2 \Rightarrow^* \mathcal{R} \cdots \).

Theorem D.4 (Property Undecidability). Testing if \( T \) has (local) confluence or is terminating is undecidable in general.

D.7 Critical Pair Analysis

Throughout this section, we fix some common label alphabet \( \mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E) \), and also require that the interface in all rules to be totally labelled.

Definition D.29. The derivations \( G_1 \Rightarrow^* \mathcal{R} H \Rightarrow^* \mathcal{R} G_2 \) are sequentially independent iff \( (h_1(R_1) \cap g_2(L_2)) \subseteq (h_1(K_1) \cap g_2(K_2)) \).

Lemma D.1. The derivations \( G_1 \Rightarrow^* \mathcal{R} H \Rightarrow^* \mathcal{R} G_2 \) are sequentially independent iff there exist morphisms \( R_1 \rightarrow D_2 \) and \( L_2 \rightarrow D_1 \) with \( R_1 \rightarrow D_1 \rightarrow H = R_1 \rightarrow H \) and \( L_1 \rightarrow D_2 \rightarrow H = L_2 \rightarrow H \).

Theorem D.5 (Sequential Independence). If \( G_1 \Rightarrow^* \mathcal{R} H \Rightarrow^* \mathcal{R} G_2 \) are sequentially independent, then there exists a graph \( H' \) and sequentially independent steps \( G \Rightarrow^* \mathcal{R} H' \Rightarrow^* \mathcal{R} G_2 \).

Definition D.30. The derivations \( H_1 \Leftarrow^* \mathcal{R} G \Rightarrow^* \mathcal{R} H_2 \) are parallelly independent iff \( (g_1(L_1) \cap g_2(L_2)) \subseteq (g_1(K_1) \cap g_2(K_2)) \).
Lemma D.2. The derivations $H_1 \xleftarrow{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are **parallelly independent** iff there exist morphisms $L_1 \rightarrow D_2$ and $L_2 \rightarrow D_1$ with $L_1 \rightarrow D_2 \rightarrow G = L_1 \rightarrow G$ and $L_2 \rightarrow D_1 \rightarrow G = L_2 \rightarrow G$.

Remark D.4. The derivations $H_1 \xleftarrow{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are **parallelly independent** iff $H_1 \Rightarrow_{r_1} L_1 \Rightarrow_{r_2} L_2$ and $G \Rightarrow_{r_2} H_2 \Rightarrow_{r_1} G'$ sequentially independent.

Theorem D.6 (Parallel Independence). If $H_1 \xleftarrow{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are parallelly independent, then there exists a graph $G'$ and direct derivations $H_1 \Rightarrow_{r_2} G' \xleftarrow{r_1} H_2$ with $G \Rightarrow_{r_1} H_1 \Rightarrow_{r_2} G'$ and $G \Rightarrow_{r_2} H_2 \Rightarrow_{r_1} G'$ sequentially independent.

Definition D.31. A pair of direct derivations $G_1 \xleftarrow{r_1,g_1} H \Rightarrow_{r_2,g_2} G_2$ is a **critical pair** iff $H = g_1(L_1) \cup g_2(L_2)$, the steps are not parallelly independent, and if $r_1 = r_2$ then $g_1 \neq g_2$.

Lemma D.3. Every graph transformation system has, up to isomorphism, only finitely many critical pairs.

Definition D.32. Let $G \Rightarrow H$ be a **direct derivation**. Then the **track morphism** is defined to be the partial morphism $\text{tr}_{G \Rightarrow H} = in' \circ in^{-1}$, where $in$ and $in'$ are the bottom left and right morphisms in Figure D.2, respectively. We define $\text{tr}_{G \Rightarrow H}$ inductively as the composition of track morphisms.

Definition D.33. The set of **persistent nodes** of a critical pair $\Phi : H_1 \xleftarrow{G} H_2$ is $\text{Persist}_\Phi = \{v \in G_V \mid \text{tr}_{G \Rightarrow H_1}(\{v\}), \text{tr}_{G \Rightarrow H_2}(\{v\}) \neq \emptyset\}$. That is, those nodes that are are not deleted by the application of either rule.

Definition D.34. A critical pair $\Phi : H_1 \xleftarrow{G} H_2$ is **strongly joinable** iff it is **joinable** without deleting any of the persistent nodes, and the persistent nodes are identified. That is, there is a graph $M$ and derivations $H_1 \Rightarrow_{r_2} M \xleftarrow{r_2} H_2$ such that $\forall v \in \text{Persist}_\Phi, \text{tr}_{G \Rightarrow H_1 \Rightarrow_{r_2} M}(\{v\}) = \text{tr}_{G \Rightarrow H_2 \Rightarrow_{r_2} M}(\{v\}) \neq \emptyset$.

Theorem D.7 (Critical Pair Lemma). A graph transformation system $T$ is **locally confluent** if all its critical pairs are strongly joinable.

Remark D.5. The reverse direction of this theorem is false, as this would contradict the undecidability of checking for confluence. If we find a non-joinable critical pair, we have non-confluence, and if we have all all critical pairs joinable, but not all strongly, we can draw no conclusions.
D.8 Rooted Graph Transformation

We fix some common label alphabet $\mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E)$, and allow rules to have a partially labelled interface again.

**Definition D.35.** Let $G$ be a partially labelled graph, and $P_G \subseteq V_G$ be a set of root nodes. Then a rooted partially labelled graph is the tuple $\hat{G} = (G, P_G)$.

**Definition D.36.** Given two rooted partially labelled graphs $\hat{G}, \hat{H}$, a partially labelled graph morphism $g : G \rightarrow H$ is a rooted labelled graph morphism $\hat{G} \rightarrow \hat{H}$ iff $g(V)(P_G) \subseteq P_H$. A morphism $g : \hat{G} \rightarrow \hat{H}$ is injective/surjective iff the underlying graph morphism is injective/surjective. Inclusion morphisms and subgraphs are defined in the obvious way.

**Definition D.37.** We say that rooted partially labelled graphs $\hat{G}, \hat{H}$ are isomorphic iff there exists an injective, surjective morphism $g : \hat{G} \rightarrow \hat{H}$ such that $g^{-1} : \hat{H} \rightarrow \hat{G}$ is also a morphism, and we write $\hat{G} \cong \hat{H}$. This naturally gives rise to equivalence classes $[\hat{G}]$: the countably many rooted partially labelled abstract graphs over some fixed $\mathcal{L}$.

**Definition D.38.** Direct derivations on rooted totally labelled graphs are defined analogously as for totally labelled graphs, but with the following modifications to the rule application process (Definition D.15):

1. The root nodes of the intermediate graph are $P_G \setminus g(V)(P_L \setminus P_K)$.
2. The root nodes of the result graph are $P_D \cup h(V)(P_R \setminus P_K)$.

We write $\hat{G} \Rightarrow_{r,g} \hat{M}$ for a successful application of $r$ to $\hat{G}$ using match $g$, obtaining result $\hat{H} \cong \hat{M}$. We call this a direct derivation. Definitions D.18 and D.19 are analogous.

**Theorem D.8** (Rooted Derivation Uniqueness). The result graph of a direct derivation is unique up to isomorphism and is totally labelled.

**Definition D.39.** A rooted graph transformation system $\hat{T} = (\mathcal{L}, \hat{R})$, consists of a label alphabet $\mathcal{L}$, and a finite set $\hat{R}$ of rules over $\mathcal{L}$.

**Proposition D.5.** Given a rooted graph transformation system $\hat{T} = (\mathcal{L}, \hat{R})$, then one can always decide if $\hat{G} \Rightarrow_{\hat{R}} \hat{H}$.

**Definition D.40.** Given a rooted graph transformation system $\hat{T} = (\mathcal{L}, \hat{R})$, a subalphabet of non-terminals $\mathcal{N}$, and a start graph $\hat{S}$ over $\mathcal{L}$, then a rooted graph grammar is the system $\hat{G} = (\mathcal{L}, \mathcal{N}, \hat{R}, \hat{S})$.

**Definition D.41.** Given a rooted graph grammar $\hat{G}$ as defined above, we say that a graph $\hat{G}$ is terminally labelled iff $l(V) \cap \mathcal{N}_V = \emptyset$ and $m(E) \cap \mathcal{N}_E = \emptyset$. Thus, we can define the graph language:

$$L(\hat{G}) = \{ [\hat{G}] \mid \hat{S} \Rightarrow^*_{\hat{R}} G, G \text{ terminally labelled} \}$$
D.9 Edge Replacement Grammars

One can describe edge replacement grammars in terms of DPO-grammars.

**Definition D.42.** An edge replacement (DPO-)grammar is a tuple $G = (\mathcal{L}, N, R, S)$ where $\mathcal{L}_V = \{\blacksquare\}$, $N_V = \emptyset$, $S = \blacksquare\rightarrow\blacksquare$ for some $a \in N_E$, and the rules in $R$ are of the form given in Figure D.5 for some $a \in N_E$ and arbitrary graph $R$ over $\mathcal{L}$.

$\begin{array}{c}
\blacksquare \xrightarrow{a} \blacksquare \\
1 & 2 \\
\blacksquare \xleftarrow{} \blacksquare & \blacksquare \rightarrow R
\end{array}$

Figure D.5: Edge Replacement Rule

**Theorem D.9** (Decidable Membership Problem). The membership problem for edge replacement grammars is in $NP$. Moreover, there exists a grammar such that the problem is $NP$-complete.

**Corollary D.1.** There exists a recursively enumerable set of graphs that cannot be generated by an edge replacement grammar. Moreover, the edge replacement grammars generate a recursive class of languages.
Appendix E

Graph Theory

In standard literature, ‘graph theory’ is the mathematical study of ‘graphs’, where in this context a graph is a finite set of vertices with (directed) edges between them, without parallel edges. We will present this theory in terms of the more general notion of a (labelled) graph from Appendix D. The definitions and theorems in this appendix have been adapted from [104], [105], Chapter 1 of [106], and Chapter 3 of [107].

E.1 Basic Definitions

Definition E.1. Given a concrete graph $G$, $v \in V_G$, we define the:

1. **Incoming degree**: $\text{indeg}_G(v) = |t_G^{-1}(\{v\})|$
2. **Outgoing degree**: $\text{outdeg}_G(v) = |s_G^{-1}(\{v\})|$
3. **Degree**: $\text{deg}_G(v) = \text{indeg}_G(v) + \text{outdeg}_G(v)$
4. **Neighbourhood**: $N_G(v) = s_G(t_G^{-1}(\{v\})) \cup t_G(s_G^{-1}(\{v\}))$
5. **Closed neighbourhood**: $N_G[v] = N_G(v) \cup \{v\}$

Definition E.2. Given a concrete graph $G$, $v \in V_G$, we:

1. Say $v \in V_G$ is a **leaf node** iff $\text{outdeg}_G(v) = 0$
2. Say $u, v \in V_G$ are **adjacent** iff $\{u, v\} \subseteq N[u] \cap N[v]$
3. Say $e \in E_G$ is **proper** iff $s_G(e) \neq t_G(e)$

Definition E.3. We say two proper edges $e, f \in E_G$ are **parallel** iff $[s_G(e) = s_G(f)$ and $t_G(e) = t_G(f)]$ or $[s_G(e) = t_G(f)$ and $s_G(e) = t_G(f)]$

Definition E.4. Let $G$ be a concrete graph. Then:

1. An **undirected walk** of length $k$ is a non-empty, finite sequence of alternating vertices and edges in $G$: $(v_0, e_0, v_1, e_1, \ldots, e_{k-1}, v_k)$, such that for each $e_i$ $(0 \neq i < k)$, $[s_G(e_i) = v_i$ and $t_G(e_i) = v_{i+1}]$ or $[s_G(e_i) = v_i$ and $t_G(e_i) = v_i]$. 

69
2. A walk is an undirected walk such that for each \( e_i \) \((0 \leq i < k)\), \( s_G(e_i) = v_i \) and \( t_G(e_i) = v_{i+1} \);
3. We call a (undirected) walk closed iff \( v_0 = v_k \);
4. If the vertices \( v_i \) of a walk are all distinct (except possibly \( v_0 = v_k \)), we call the walk a path;
5. A closed walk is called a cycle; a graph with no cycles is acyclic. Similarly, a closed undirected walk is called an undirected cycle.

**Definition E.5.** A graph is called connected iff there is an undirected walk between every pair of distinct vertices. A connected component of a concrete graph \( G \) is a maximal connected subgraph.

**Theorem E.1** (Graph Decomposition). Every concrete graph \( G \) has a unique decomposition into connected components.

**Definition E.6.** Given a concrete graph \( G, v \in V_G \) we define the:
1. **Children:** \( \text{children}_G(v) = t_G(s_G^{-1}(\{v\})) \);
2. **Parents:** \( \text{parents}_G(v) = s_G(t_G^{-1}(\{v\})) \).

\( u \) is a child of \( v \) iff \( u \in \text{children}_G(v) \), and a parent iff \( u \in \text{parents}_G(v) \).

**Proposition E.1.** Given a concrete graph \( G, v \in V_G \). Then:
1. \( \text{children}_G(v) \subseteq N_G(v) \) and \( \text{parents}_G(v) \subseteq N_G(v) \);
2. \(|\text{children}_G(v)| \leq \text{outdeg}_G(v)\) and \(|\text{parents}_G(v)| \leq \text{indeg}_G(v)\).

**E.2 Classes of Graphs**

**Definition E.7.** We say a graph class is of bounded degree iff there exists some \( b \in \mathbb{N} \) such that for all graphs \( G \) in the class, \( \forall v \in V_G, \deg(v) \leq b \).

**Definition E.8.** A graph is called discrete iff it has no edges.

**Definition E.9.** A tree is a non-empty connected graph without undirected cycles such that every node has at most one incoming edge. Moreover:
1. A linked list is a tree such that every node has outgoing degree at most 1;
2. A binary tree is a tree such that every node has outgoing degree at most 2;
3. A full binary tree is a binary tree such that every node has either 0 or 2 children;
4. A perfect binary tree is a full binary tree such that every maximal path is the same length;
5. A forest is a graph where each connected component is a tree.
**E.2. CLASSES OF GRAPHS**

**Proposition E.2.** A **tree** is a graph containing a node from which there is a unique directed path to each node in the graph.

**Definition E.10.** A **$n \times m$-grid graph** is a graph with underlying unlabelled graph isomorphic to $(V,E,s,t)$ where $V = \mathbb{Z}_n \times \mathbb{Z}_m$, $E = (\mathbb{Z}_2 \times V) \setminus \{(0,i,m-1),(1,n-1,j)| i \in \mathbb{Z}_n, j \in \mathbb{Z}_m\}$, $s(d,i,j) = (i,j)$, and $t(d,i,j) = (i+d,j+1-d)$. We call such a graph **square** iff $n = m$.

**Definition E.11.** An **$n$-star graph** is a graph with underlying unlabelled graph isomorphic to $(V,E,s,t)$ where $V = \mathbb{Z}_{n+1}$, $E = \mathbb{Z}_n$, and:

$$s(i) = \begin{cases} 
    n & \text{if } i \equiv 0 \mod 2 \\
    i & \text{otherwise} 
\end{cases} \quad t(i) = \begin{cases} 
    n & \text{if } i \equiv 1 \mod 2 \\
    i & \text{otherwise} 
\end{cases}$$

An example linked list, perfect binary tree, square grid graph, and star graph can be found in Figure 3.7.
Bibliography


