Department of Computer Science

Submitted in part fulfilment for the degree of BSc.

Efficient Graph Rewriting

Graham Campbell

April 2019

Supervisor: Dr Detlef Plump

Acknowledgements

I should like to thank my supervisor, Detlef Plump, for his support and guidance this year and for giving me the benefit of his invaluable technical understanding of rewriting systems. Our (regularly overrunning!) meetings were of great benefit, and I hope to continue to collaborate after I move to the School of Mathematics, Statistics and Physics at Newcastle.

Contents

List of Figures

List of Theorems

Executive Summary

Introduction, 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] [4] [5]. Applications in Computer Science are wide-reaching including compiler construction, software engineering, natural language processing, modelling of concurrent systems, and logical and functional programming [6] [7] [8]. There are a number of GT languages and tools [9] [10] [11] [12] [13] [14].

The declarative nature of graph rewriting rules comes at a cost. In general, to match the left-hand graph of a fixed rule within a host graph requires polynomial time. To improve matching performance, Dörr [15] 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 by Bak and Plump in GP 2, allowing programs to rival the performance of traditional implementations in languages such as C [16].

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 [17] [18]. 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 [19] [20].

Testing for 'confluence' is not possible in general [21], however we can sometimes use 'critical pair' analysis to show confluence. Confluence remains poorly understood, and while there are techniques for classifying 'conflicts' [22] [23], 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.

Outline, Results and Evaluation

We regard this project as a success, having achieved our four original goals. Each of our goals have been addressed by the first four chapters, respectively. We started by 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 [24] 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 Local Church-Rosser and Parallelism theorems hold [25] [18]. 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 GP 2 and collecting timing results. We have submitted our program and results for publication [26]. 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 modulo 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 modulo garbage. We have applied this theory to Extended Flow Diagrams [27] and the encoding of partially labelled (rooted) GT systems as standard GT systems, performing non-garbage critical pair analysis on the encoded system. Further exploring the relationship between confluence modulo garbage and weak garbage separation 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.

1 Theoretical Background

Before reading the main text, the reader should first skim read Appendix A in order to set up notation and definitions.

In this chapter, we will review the rewriting of totally labelled graphs with relabelling, and Bak and Plump's modifications adding 'root' nodes [16]. 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.

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'. △

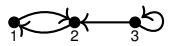


Figure 1.1: Example Concrete Graph

Definition 1.3. Given two concrete graphs G and H, a **graph morphism** $g : G \to H$ is a pair of maps $g = (g_V : V_G \to V_H, g_E : E_G \to E_H)$ 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.

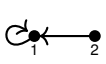
$$\begin{array}{ccc} E_G & \xrightarrow{s_G} & V_G \\ \downarrow{\scriptstyle g_E} & & \downarrow{\scriptstyle g_V} \\ E_H & \xrightarrow{s_H} & V_H \end{array} \qquad \begin{array}{ccc} E_G & \xrightarrow{t_G} & V_G \\ \downarrow{\scriptstyle g_E} & & \downarrow{\scriptstyle g_V} \\ E_H & \xrightarrow{t_H} & V_H \end{array}$$

Figure 1.2: Graph Morphism Commuting Diagrams

Definition 1.4. A graph morphism $g : G \to H$ 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 $G \to H$, three of which are injective, none of which are surjective. There are actually also four morphisms $H \to G$, three of which are surjective. △

$G =$ 

$H =$ 

Figure 1.3: Example Concrete Graphs

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

Proposition 1.1. The **quotient** (Definition A.20) 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' [28], 'node replacement' [29], and 'algebraic' approaches [3] [4]. The two major approaches to algebraic graph transformation are the so called '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 [5, p.9-14] [4]. For this reason, we will focus only on the DPO approach with injective matching. Moreover, DPO graph grammars can generate every recursively enumerable set of graphs [30].

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 C.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 C.2. Note that typed (attributed) (hyper)graphs have a rich theory [31] [32] [33] [34] [35] [20].

A review of graph transformation of labelled graphs using the DPO approach with injective matching can be found in Appendix C. 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 C.

We are in fact using a restricted version of the theory presented by Habel and Plump [17], 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 $(\{1,2\},\{\square\})$ where x,y are to be determined:

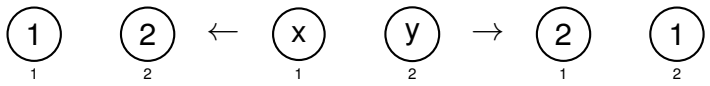


Figure 1.4: Relabelling Non-Example

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. △

1.4 Rooted Graph Transformation

Rooted graph transformation first appeared when Dörr [15] proposed to equip rules and host graphs with distinguished (root) nodes, and to match roots in rules with roots in host graphs. More recently, Bak and Plump [16] [36] have used rooted graph transformation in conjunction with the theory of partially labelled graph transformation in GP 2.

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 [37] [19] [16] [38]. An excellent account of this is available in Part II of Dodds' Thesis [24].

We can define rooted graphs in a pointed style, just as for typed graphs. An account of the theoretical modifications is provided in Section C.9, using Bak's approach [16]. Note that Dodds [19] [24] previously implemented root nodes via an augmentation of the label alphabet, however Bak's approach makes for more concise theory, and has been implemented in GP 2.

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 C.2. Moreover, given a match, one can decide if it is applicable in $O(|r|)$ time. That is, the RAP requires $O(|r|)$ time. [24]

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 containing a bounded number of root nodes, and a rule r of bounded size with L containing a single root node, then the time complexity of GMP reduces to constant time [24].

Example 1.5. Figure 1.5 '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 [16], showcasing root nodes in GP 2. △

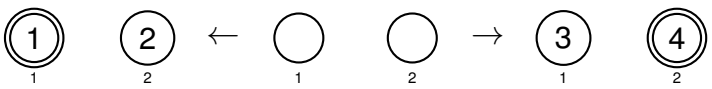


Figure 1.5: Example Rooted Rule

We will revisit time complexity in Section 2.7, 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 [39].

1.5 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 B). These systems were studied for the first time by Newman in the early 40s [40]. 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 B.4), **finitely branching** (Definition B.6), **confluent** (Definition B.4) ARS (Definition B.1). △

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

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

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

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

Lemma 1.1. Consider the **ARS** $(\mathcal{G}(\mathcal{L}), \to)$ induced by a (rooted) **GTS**. Then $\to$ is a **binary relation** (Definition A.13) on $\mathcal{G}(\mathcal{L})$. Moreover, it is **finitely branching** (Definition B.6) and **decidable** (Definition A.28).

Proof. By Proposition 1.3, $\mathcal{G}(\mathcal{L})$ is a countable set, and so $\to$ is a countable set (by Theorem A.2), and is well-defined since derivations are unique up to isomorphism (Theorem C.9). Finally, we have only finitely many rules, and for each rule, there can only exist finitely many matches $L \to G$, so there can only ever be finitely many result graphs H (up to isomorphism) $G \Rightarrow_{\mathcal{R}} H$ for any given G. □

Theorem 1.1 (Property Undecidability)**.** Consider the **ARS** $(\mathcal{G}(\mathcal{L}), \to)$ induced by a (rooted) **GTS**. Then testing if $\to$ 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 [41]. Undecidability of (local) confluence checking was shown by Plump in 1993 [21], even for terminating GT systems [42]. □

1.6 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 $T = (\mathcal{L}, \mathcal{R})$, consider the **state space** $\Sigma = \mathcal{G}(\mathcal{L}) \cup \{\bot\}$ and the induced ARS $(\mathcal{G}(\mathcal{L}), \to_{\mathcal{R}})$. We may define the **semantic function** $f_T : \mathcal{G}(\mathcal{L}) \to \Sigma$ by $f_T([G]) = \{[H] \mid [H]$ is a normal form of $[G]$ with respect to $\to_{\mathcal{R}}\} \cup \{\bot \mid$ there is an infinite reduction sequence starting from $[G]\}$ and $f_T(\bot) = \{\bot\}$. △

There are a number of GT languages and tools, such as AGG [10], GMTE [43], Dactl [44], GP 2 [14], GReAT [11], GROOVE [12], GrGen.Net [13], Henshin [45], PROGRES [9], and PORGY [46]. Habel and Plump [47] 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 ($\mathcal{R}$);
2. Sequential composition ($P1;\ P2$);
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. $\rightarrow_{P1;P2} := \rightarrow_{P2} \circ \rightarrow_{P1}$.
3. $\rightarrow_{P{\downarrow}} := \{([G],[H]) \mid [G] \rightarrow_P^* [H]$ and $[H]$ is in normal form[1]$\}$. $\triangle$

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.

GP 2 is an experimental rule-based language for problem solving in the domain of graphs, developed at York, the successor of GP [48] [14]. GP 2 is of interest because it has been designed to support formal reasoning on programs [49], with a semantics defined in terms of partially labelled graphs, using the injective DPO approach with relabelling [50] [17]. Poskitt and Plump have set up the foundations for verification of GP 2 programs [51] [52] [53] [54] using a Hoare-Style [55] system (actually for GP [56] [48]), Hristakiev and Plump have developed static analysis for confluence checking [57] [23], and Bak and Plump have extended the language, adding root nodes [16] [36]. Plump has shown computational completeness [58].

GP 2 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 $(\mathbb{Z} \cup Char^*)^* \times \mathcal{B}$. 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. [14]

The formal semantics of GP 2 is given in the style of Plotkin's structural operational semantics [59]. Inference rules inductively define a small-step transition relation $\rightarrow$ on configurations. The inference rules and definition of the semantic function $[\![.]\!] : ComSeq \rightarrow \mathcal{G} \rightarrow \mathcal{P}(\mathcal{G} \cup \{fail, \bot\})$ were first defined in [14]. Up-to-date versions can be found in [36].

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

2 A New Theory

Graph transformation with relabelling as described in Sections 1.2, 1.3, C.5 and C.6 has desirable properties. It was shown by Habel and Plump in 2002 [17] that derivations are natural double pushouts (Theorem C.4) and thus are invertible. Unfortunately, Bak and Plump's modifications to add root nodes (Sections 1.4 and C.9) 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 (Figure 2.1). This asymmetry is unfortunate, because derivations are no longer invertible.

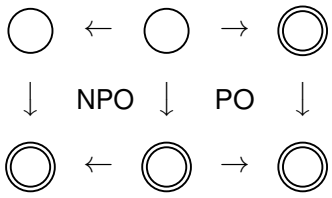


Figure 2.1: Example Rooted Derivation

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).

2.1 Graphs and Morphisms

Fix some common label alphabet (Definition C.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. $p : V \to \mathbb{Z}_2$[1] is a **partial** function, determining vertex rootedness.

Definition 2.2. 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**.

[1] $\mathbb{Z}_2$ is the quotient $\mathbb{Z}/2\mathbb{Z} = \{0, 1\}$.

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. $\triangle$

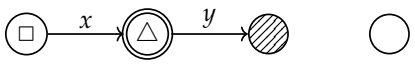


Figure 2.2: Example Graph

Definition 2.3. 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]
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]
5. $\forall v \in p_G^{-1}(\mathbb{Z}_2),\ p_G(v) = p_H(g_V(v))$. [Rootedness]

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

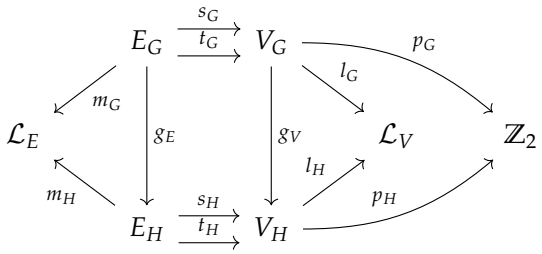


Figure 2.3: Graph Morphism Commuting Diagrams

Proposition 2.1. Our new notion of graphs and morphisms is a **locally small category** (Definition A.29), just like the previous notions.

Definition 2.4. 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 **injective** and **surjective**, and $g^{-1} : H \to G$ is a graph morphism.

Definition 2.5. We say H is a **subgraph** of G iff there exists an **inclusion morphism** $H \hookrightarrow G$. This happens iff $V_H \subseteq V_G$, $E_H \subseteq E_G$, $s_H = s_G|_{E_H}$, $t_H = t_G|_{E_H}$, $m_H = m_G|_{E_H}$, $l_H \subseteq l_G$, $p_H \subseteq p_G$ (treating functions as sets).

Definition 2.6. We say that graphs G, H are **isomorphic** iff there exists a **graph isomorphism** $g : G \to H$. 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 $\mathcal{L}$.

Proposition 2.2. $\mathcal{G}^{\oplus}(\mathcal{L})$ is a **countable set**.

Definition 2.7. If G is a **graph**, then $|G| = |V_G| + |E_G|$.

2.2 Rules and Derivations

Fixing some common $\mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E)$, we define rules and derivations.

Definition 2.8. A **rule** $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of left/right **TLRGs** L, R, the interface **graph** K, and **inclusions** $K \hookrightarrow L$ and $K \hookrightarrow R$.

Example 2.2. See Figure 3.2. △

Definition 2.9. We define the **inverse rule** to be $r^{-1} = \langle R \leftarrow K \rightarrow L \rangle$.

Definition 2.10. If $r = \langle L \leftarrow K \rightarrow R \rangle$ is a **rule**, then $|r| = max\{|L|, |R|\}$.

Definition 2.11. Given a **rule** $r = \langle L \leftarrow K \rightarrow R \rangle$ and a **TLRG** G, we say that an **injective** morphism $g : L \hookrightarrow G$ satisfies the **dangling condition** iff no edge in $G \setminus g(L)$ is incident to a node in $g(L \setminus K)$.

Definition 2.12. To **apply** a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ to some **TLRG** G, find an **injective** graph morphism $g : L \hookrightarrow G$ 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 2.13. 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 $G \Rightarrow_r M$.

Definition 2.14. 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 2.15. 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.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Figure 2.4: Commuting Squares

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 C.11).

Proof. The constructions are exactly as in Lemma 1 of [17], with the rootedness function defined analogously to the node labelling function. Trivial modifications to the proof give the result. □

Lemma 2.2. Let $b : K \to R$, $d : K \to D$ be **graph morphisms** such that d is **injective** and $\forall v \in V_R, |l_R(\{v\}) \cup l_D(d_V(b_V^{-1}(\{v\})))| \leq 1$. 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 C.10).

Proof. The constructions are exactly as in Lemma 2 of [17], with the rootedness function defined analogously to the node labelling function. □

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 C.12) iff $l_D(d_V(V_K \setminus l_K^{-1}(\mathcal{L}_V))) = \emptyset = p_D(d_V(V_K \setminus p_K^{-1}(\mathbb{Z}_2)))$.

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 [17] with the obvious modifications. Similar for the other direction. □

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 [17] with the obvious modifications. □

Theorem 2.1 (Derivation Uniqueness)**.** Given a rule $\langle L \leftarrow K \rightarrow R \rangle$ and an **injective graph morphism** $g : L \to G$, then there exists a **natural DPO** diagram as above iff g satisfies the **dangling condition**. In this case, D and H are unique up to isomorphism. This exactly corresponds to Definition 2.12. Moreover, if $G \Rightarrow_r H$, then G is a **TLRG** iff H is a **TLRG**.

Proof. Proceed as in Theorem 1 of [17] with the obvious modifications. Totality of labelling is given by Theorem 2 of [17], and totality of rootedness is given by replacing all occurrences of the labelling function with the rootedness function in the proof. □

Corollary 2.1. Derivations are **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 $g : L \to G$, and a comatch $h : R \to H$, and so by symmetry, we have the result. □

This symmetry is unique to this new approach to rooted graph transformation. In Bak's approach (Appendix C.9), 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. Finally, we can now show our root node invariance result:

Lemma 2.5. Let G be a **TLRG**, and $r = \langle L \leftarrow K \rightarrow R \rangle$ 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. □

By comparison, in Bak's system, non-root nodes could be matched against root nodes.

Corollary 2.2. Let G be a **TLRG**, and $r = \langle L \leftarrow K \rightarrow R \rangle$ a rule such that L and R both contain k root nodes, for some fixed $k \in \mathbb{N}$. Then any **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 2.12, so, $|p_G^{-1}(\{1\})| = |p_H^{-1}(\{1\})|$. □

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.

Definition 2.16. 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 $f(K_1) = g(K_1) = K_2$. Write $r_1 \cong r_2$.

Proposition 2.3. The above notion of **rule isomorphism** is an **equivalence**, and gives rise to **abstract rules** $[r]$.

Definition 2.17. 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 $K' = (V_K, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. We say two rules r_1, r_2 are **normalisation equivalent** iff $r_1{\downarrow} \cong r_2{\downarrow}$. We write $r_1 \simeq r_2$.

Proposition 2.4. Clearly, this gives us a **coarser** notion of **equivalence** for rules than the notion of **isomorphism**.

Example 2.3. Consider the rules over $(\{\square, \triangle\}, \{\square, \triangle\})$ 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'. △

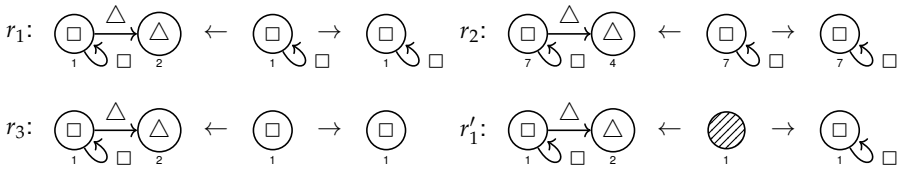


Figure 2.5: Example (Non-)Isomorphic Rules

Theorem 2.2 (Well Behaved 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 $G \Rightarrow_{r\downarrow} H$.

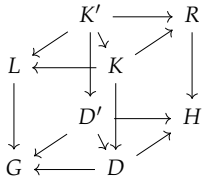


Figure 2.6: Derivations Diagram

Proof. Consider some fixed graph G. The set of injective morphisms $g : L \rightarrow G$ 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 2.12, $G \Rightarrow_{r,g} H$ iff $G \Rightarrow_{r\downarrow,g} H$. □

Remark 2.3. Normal forms for rules is not actually a new observation, and is the foundation of rule schemata in GP 2 [14]. Moreover, maximising the number of edges in the interface of rules leads to a reduction of the number of critical pairs (Section C.8) of a GT system [57].

2.5 Transformation Systems

We can now define graph transformation systems using our new definitions of graphs and rules. Next, we will look at equivalence and complexity.

Definition 2.18. 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 2.19. 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 2.20. 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 $\boldsymbol{G} = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$.

Definition 2.21. Given a **graph grammar** $\boldsymbol{G}$ 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 $\boldsymbol{G}$:

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

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

Lemma 2.6. A GT system $T = (\mathcal{L}, \mathcal{R})$ induces a **decidable**, **finitely branching** ARS $(\mathcal{G}^{\oplus}(\mathcal{L}), \to)$ where $[G] \to [H]$ iff $G \Rightarrow_{\mathcal{R}} H$, just like in Section 1.5.

Remark 2.4. This does not, in general, imply that $\to^*$ is decidable. We say that T is (**locally**) **confluent** (**terminating**) iff its **induced ARS** is.

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 2.22. Two **GT systems** T_1, T_2 over a common alphabet are:

1. **Isomorphic** ($T_1 \cong T_2$) iff $\mathcal{R}_1/{\cong} = \mathcal{R}_2/{\cong}$; [2]
2. **Normalisation equivalent** ($T_1 \simeq_N T_2$) iff $\mathcal{R}_1/{\simeq} = \mathcal{R}_2/{\simeq}$;
3. **Step-wise equivalent** ($T_1 \simeq_S T_2$) iff the induced **ARSs**[3] are identical;
4. **Semantically equivalent** ($T_1 \simeq_F T_2$) iff the **semantic functions** (modify Example 1.8 in the obvious way) are identical.

Proposition 2.5. Each of the above notions are equivalences.

Proposition 2.6. This notion of isomorphism gives rise to **abstract graph transformation systems** $[T]$ over some fixed label alphabet $\mathcal{L}$. Let $\mathcal{T}(\mathcal{L})$ denote the collection of all such classes. Then, $\mathcal{T}(\mathcal{L})$ is a **countable set**.

Remark 2.5. 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**. Moreover, the inclusion is strict, in general. That is, $T_1 \cong T_2 \Rightarrow T_1 \simeq_N T_2 \Rightarrow T_1 \simeq_S T_2 \Rightarrow T_1 \simeq_F T_2$.

[2] This is a quotient (Definition A.20) by rule isomorphism (Definition 2.16).

[3] Induced ARSs are as defined in Lemma 2.6.

Proof. Let T_1, T_2 be GT systems over some $\mathcal{L}$, with rule sets $\mathcal{R}_1$, $\mathcal{R}_2$. 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 $\mathcal{R}_1$ correspond to those of $\mathcal{R}_2$. 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 $\mathcal{R}_1$ and $\mathcal{R}_2$ 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 $\to_{\mathcal{R}_1}, \to_{\mathcal{R}_2}$ 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. □

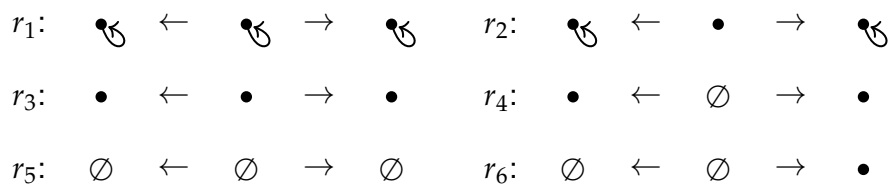


Figure 2.7: Example Rules Demonstrating Non-Equivalence

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 [60] [61] and Wirth [62]. 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.

2.7 Complexity Theorems

The Graph Matching Problem (Definition 1.6) and Rule Application Problem (Definition 1.7) can be considered in this our setting. When we say 'bounded degree', we mean the degree of each node has a constant upper bound. We will see that if we have an input graph with bounded degree and a bounded number of root nodes, and a finite set of 'fast' rules, then we can perform matching in constant time.

Definition 2.23. We call a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ **fast** iff every connected component (Definition D.5) of L contains a root node.

Just like in Lemma 1.2, we need to set up some assumptions about the complexity of various problems. We will again be using the assumptions from Figure C.2, assuming that the rootedness of any node can be accessed in constant time and that we can access the set of root nodes in a graph in $O(|X|)$ time, given that there are $|X|$ root nodes.

Lemma 2.7. 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 [24, 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, with the major difference being the bounded number of root nodes in G, allowing us to conclude $O(|r|)$ time rather than $O(|V_G|)$ time. □

Lemma 2.8. 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 2.24. We say that a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is **root non-increasing** iff $|p_L^{-1}(\{1\})| \geq |p_R^{-1}(\{1\})|$.

Definition 2.25. A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is **degree non-increasing** iff $\forall v \in (V_R \setminus V_K), \deg_R(v) \leq N$ and $\forall v \in V_K, \deg_L(v) \geq \deg_R(v)$, where N is our upper bound on the degree of nodes.

Theorem 2.5 (Fast Derivations)**.** Given a **TLRG** G of **bounded degree** containing a **bounded** number of root nodes, and a GT system $T = (\mathcal{L}, \mathcal{R})$ where each rule is **fast**, then one can decide in **constant time** the **direct successors** (Definition B.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 2.3. Given G, T as above, where each rule is additionally **root non-increasing** and **degree non-increasing**, and T terminating with maximum derivation length $N \in \mathbb{N}$, then one can find a **normal form** (Definition B.3) of G in $O(N)$ time, up to isomorphism.

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 2.3, we need only consider the maximum length of derivations when reasoning about time complexity, as mentioned at the end of Section 1.4.

3 Recognising Trees

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 have submitted a version of this chapter for publication as part of a co-authored paper [26] looking at linear time algorithms in GP 2.

3.1 Generating Trees

Writing a graph grammar that generates all unlabelled trees (Definition D.8) 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.

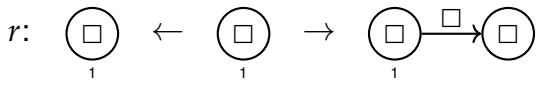


Figure 3.1: Tree Grammar Rules

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

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

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 $[G] \in \boldsymbol{L}(\boldsymbol{TREE})$ (together with Proposition C.4):

Proposition 3.1. $[G] \in \boldsymbol{L}(\boldsymbol{TREE})$ iff $G \Rightarrow_{r^{-1}} S$. Moreover, this procedure always terminates, since the system is acyclic and globally finite.

It is easy to see via critical pair analysis (Section C.8) that this system is confluent, since it has no 'critical pairs'. Unfortunately, it is not 'fast' due to the fact that in each derivation, we must consider the entire host graph when finding a match. In the next session, we will see that rooted graph transformation rules can actually recognise trees in linear time.

3.2 Linear Time Recognition

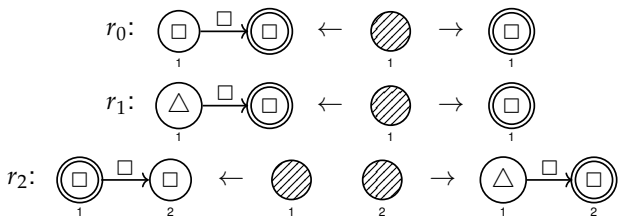
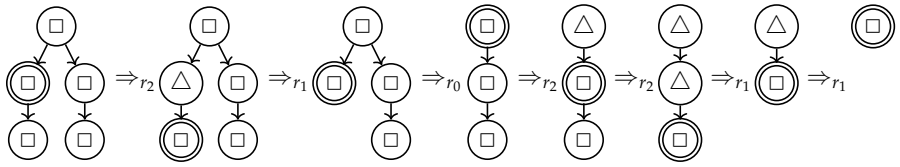


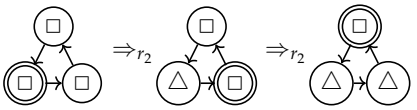
Figure 3.2: Tree Recognition Rules

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.

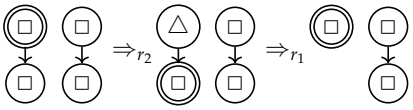
Example 3.2. Figure 3.3 shows a reduction of a tree and non-trees. △



(a) Tree Reduction



(b) 3-Cycle Reduction



(c) Forrest Reduction

Figure 3.3: Example Reductions

Definition 3.1. Given a graph G, we define $G^{\ominus}$ to be exactly G, but with every node unrooted, and everything labelled by $\square$. That is, $G^{\ominus} = (V_G, E_G, s_G, t_G, V_G \times \{\square\}, E_G \times \{\square\}, V_G \times \{0\})$.

Definition 3.2. 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 D.8).

Lemma 3.1. The system $(\mathcal{L}, \mathcal{R})$ is terminating. Moreover, derivations have length at most $2|V_G|$.

Proof. Let $\#G = |V_G|$, $\square G = |\{v \in V_G \mid l_G(v) = \square\}|$, for any TLRG G. If $G \Rightarrow_{r_0} H$ or $G \Rightarrow_{r_1} H$, then $\#G > \#H$ and $\square G > \square H$. If $G \Rightarrow_{r_2} H$ then $\#G = \#H$ and $\square G > \square H$. Suppose there were an infinite sequence

of derivations $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} \cdots$, then there would be an infinite descending chain of natural numbers $\#G_0 + \square G_0 > \#G_1 + \square G_1 > \#G_2 + \square G_2 > \cdots$, which contradicts the well-ordering of $\mathbb{N}$. 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 3.2. 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 3.1. 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 3.3. 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. In each application of r_0, r_1, r_2, the number of root nodes is invariant (Corollary 2.2), and so the result holds by induction. 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 3.1. In Bak's system (and hence GP 2), Lemma 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 3.4. If G is an input graph and $G \Rightarrow^*_{\mathcal{R}} H$. 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 $G \Rightarrow^*_{\mathcal{R}} H \Rightarrow_{\mathcal{R}} H'$ 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 3.2. Let G be an input tree and $G \Rightarrow^*_{\mathcal{R}} H$. Then the root-node in H has no $\triangle$-node children.

Proof. By Lemma 3.3, H has exactly one root node, and by Lemma 3.4, 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 3.1). □

Lemma 3.5. Let G be an input tree and $G \Rightarrow^*_{\mathcal{R}} H$. Then, either $|V_H| = 1$ or H is not in normal form.

Proof. By Lemma 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 $G \Rightarrow^*_{\mathcal{R}} H$. If $|V_H| = 1$, then H is in normal form by the proof to Lemma 3.3. Otherwise, by Corollary 3.1 H is a tree and $|V_H| > 1$. Now, the root-node in H (Lemma 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 3.2, there must be a $\square$-node child. So H is not in normal form. □

We now present the main result of this chapter:

Theorem 3.1 (Tree 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 $\square$ iff $[G^{\ominus}] \in \boldsymbol{L}(\boldsymbol{TREE})$. Moreover, for input graphs of bounded degree, we terminate in linear time.

Proof. By Lemma 3.1, our system is terminating and derivations have maximum length $2\#G$. By Corollary 3.1 and Lemma 3.5, G is a tree iff we can derive the singleton tree without backtracking. Finally, by Corollary 2.3, the algorithm terminates in linear time, since our ruleset satisfies the necessary conditions. □

3.3 GP 2 Implementation

Our algorithm can be implemented in GP 2. The program (Figure 3.4) expects an arbitrary labelled input graph with every node coloured grey, no 'root' nodes, and no additional 'marks'. It will fail iff 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.

To see that the program is correct follows mostly from our existing proofs. Grey nodes encode the $\square$ label, and blue nodes, $\triangle$. 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 (Lemma 3.3).

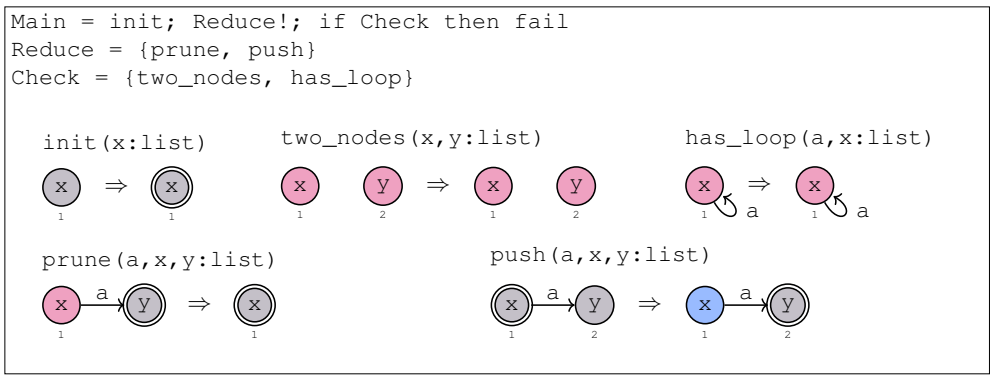
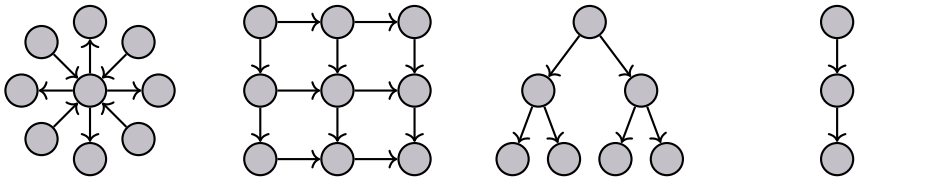


Figure 3.4: GP 2 Implementation

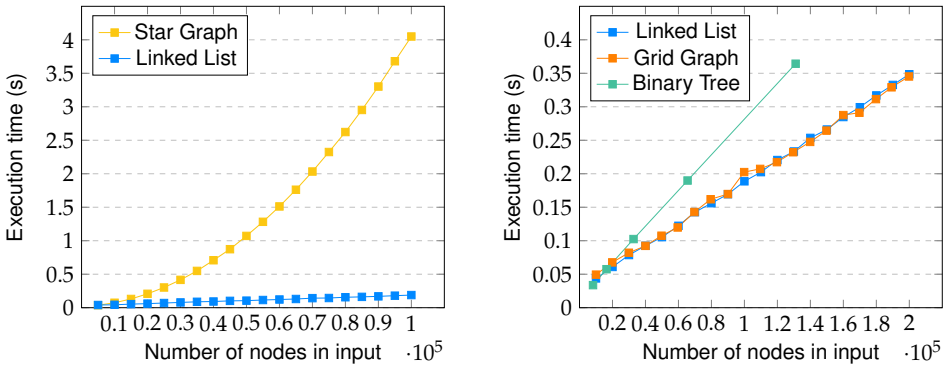
We have performed empirical benchmarking to verify the complexity of the program, testing it with linked lists, binary trees, grid graphs, and star graphs (Figure 3.5). Formal definitions of each of these graph classes can be found in Section D.2. We have exclusively used ‘perfect’ binary trees, and ‘square’ grid graphs in our testing.



(a) Star Graph (b) Grid Graph (c) Binary Tree (d) Linked List

Figure 3.5: 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 3.6).



(a) Star Graphs and Linked Lists (b) Bounded Degree Input Graphs

Figure 3.6: Measured Performance

4 Confluence Analysis

Efficient testing of language membership is an important problem in graph transformation [63] [64] [20]. 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 modulo garbage'. The name is attributed to Plump, however it appears in no published work.

4.1 Confluence Modulo Garbage

In this section, we shall be working with standard GT systems, as defined in Appendix C, 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 $D \subseteq \mathcal{G}(\mathcal{L})$ 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 $D \subseteq \mathcal{G}(\mathcal{L})$. T is **weakly garbage separating** with respect to D iff for all G, H such that $G \Rightarrow_{\mathcal{R}} H$, if $[G] \in D$ then $[H] \in D$. T is **garbage separating** 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 [65]) logical formula.

There are a couple of immediately obvious results:

Proposition 4.1. Garbage separation $\Rightarrow$ **weak garbage separation**.

Proposition 4.2. Given $T = (\mathcal{L}, \mathcal{R})$ **weakly garbage separating** with respect to $D \subseteq \mathcal{G}(\mathcal{L})$, then for all graphs G, H such that $G \Rightarrow^*_{\mathcal{R}} H$, if $[G] \in D$, then $[H] \in D$.

Example 4.1. Consider the reduction rules in Figure 4.1. The GT system $((\{\square\}, \{\square\}), \{r_1\})$ is **weakly garbage separating** w.r.t. the language of acyclic graphs, and $((\{\square\}, \{\square\}), \{r_2\})$ **garbage separating** w.r.t. the language of trees or the language of forests. $\triangle$

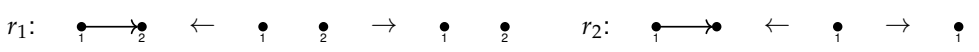


Figure 4.1: Example Reduction Rules

We can now define (local) confluence modulo garbage, allowing us to say that, ignoring the garbage graphs, a system is (locally) confluent.

Definition 4.3. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If for all graphs G, H_1, H_2, such that $[G] \in D$, if $H_1 \Leftarrow_{\mathcal{R}} G \Rightarrow_{\mathcal{R}} H_2$ implies that H_1, H_2 are **joinable**, then T is **locally confluence modulo garbage** with respect to D.

Definition 4.4. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If for all graphs G, H_1, H_2, such that $[G] \in D$, if $H_1 \Leftarrow^*_{\mathcal{R}} G \Rightarrow^*_{\mathcal{R}} H_2$ implies that H_1, H_2 are **joinable**, then T is **confluence modulo garbage** with respect to D.

Definition 4.5. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If there is no infinite derivation sequence $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} \cdots$ such that $[G_0] \in D$, then T is **terminating modulo garbage** with respect to D.

Lemma 4.1. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$, $E \subseteq D$. Then (**local**) **confluence** (**termination**) **modulo garbage** with respect to D implies (**local**) **confluence** (**termination**) **modulo garbage** with respect to E.

Proof. Immediate consequence of set inclusion! □

Corollary 4.1. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. Then (**local**) **confluence** (**termination**) implies (**local**) **confluence** (**termination**) **modulo garbage**.

Proof. Local confluence (confluence, termination) is exactly local confluence (confluence, termination) modulo garbage with respect to $\mathcal{G}(\mathcal{L})$. □

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 modulo garbage** w.r.t. the language of acyclic graphs. Similarly, r_2 is **terminating** and **confluent modulo garbage** w.r.t. the language of trees. △

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

Proof. Since T is weakly garbage separating, by Proposition 4.2, the induced ARS $(D, \to)$ where $[G] \to [H]$ iff $G \Rightarrow_{\mathcal{R}} H$ 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) modulo garbage with respect to D. □

We can now show an analogy to **Newman's Lemma** (Theorem B.4).

Theorem 4.1 (Newman-Garbage Lemma)**.** Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If T is **terminating modulo garbage** and **weakly garbage separating**, then it is **confluent modulo garbage** iff it is **locally confluent modulo garbage**.

Proof. By Lemma 4.2, the induced ARS $(D, \to)$ is well-defined, closed, and terminating. Thus, by Theorem B.4 it is confluent iff it is locally confluent, as required. □

4.2 Non-Garbage Critical Pairs

In 1970, Knuth and Bendix showed that confluence checking of terminating term rewriting systems is decidable [66]. Moreover, it suffices to compute all 'critical pairs' and check their joinability [67] [68] [69]. 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 [21] [42]. We have summarised these results in Section C.8.

We would like to generalise Theorem C.8 to allow us to determine when we have local confluence modulo 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 C, but without relabelling. That is, all graphs are totally labelled, including interface graphs.

Definition 4.6. Let $D \subseteq \mathcal{G}(\mathcal{L})$ 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**.

Lemma 4.3. Given $D \subseteq \mathcal{G}(\mathcal{L})$, $\overline{D}$ 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 4.1. $\overline{D}$ 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 [70].

Example 4.3. $\emptyset$ and $\mathcal{G}(\mathcal{L})$ are subgraph closed. △

Example 4.4. The language of discrete graphs is subgraph closed. △

Example 4.5. 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 4.7. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. A **critical pair** (Definition C.34) $H_1 \Leftarrow G \Rightarrow H_2$ is **non-garbage** iff $[G] \in \overline{D}$.

Lemma 4.4. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. Then there are only finitely many **non-garbage critical pairs** up to isomorphism. If $\overline{D}$ is **decidable**, then one can find all the **non-garbage critical pairs** in **finite time**.

Proof. By Theorem C.8 and Remark C.4, 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. It remains to show that we can decide if a critical pair is garbage. Since $\overline{D}$ has a computable membership function, we can test if the start graph in each pair is garbage in finite time. □

Corollary 4.2. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$ be such that T is **terminating modulo garbage** and $\overline{D}$ is **decidable**. Then, one can **decide** if all the **non-garbage critical pairs** are **strongly joinable** (Definition C.37).

Proof. By Lemma 4.4, we can find the finitely many pairs in finite time, and since T is terminating modulo garbage and finitely branching (Lemma 1.1), both sides of each pair have only finitely many successors (Lemma B.1), thus we can test for strong joinability in finite time. □

Lemma 4.5. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. Then, the **non-garbage critical pairs** are **complete**. That is, for each pair of **parallelly independent** (Definition C.33) direct derivations, $H_1 \Leftarrow_{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ such that $[G] \in D$, there is a **critical pair** $P_1 \Leftarrow_{r_1,o_1} K \Rightarrow_{r_2,o_2} P_2$ with extension diagrams (1), (2), and an inclusion morphism $m : K \to G$.

$$\begin{array}{ccccc} P_1 & \Longleftarrow & K & \Longrightarrow & P_2 \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ H_1 & \Longleftarrow & G & \Longrightarrow & H_2 \end{array}$$

Figure 4.2: Pair Factorisation Diagram

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

Theorem 4.2 (Non-Garbage Critical Pair Lemma)**.** Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If all its **non-garbage critical pairs** are **strongly joinable**, then T is **locally confluent modulo garbage** with respect to 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 D, so we have the result. □

Corollary 4.3. Let $T = (\mathcal{L}, \mathcal{R})$, $D \subseteq \mathcal{G}(\mathcal{L})$. If T is **terminating modulo garbage**, **weakly garbage separating**, and all its **non-garbage critical pairs** are **strongly joinable** then T is **confluent modulo garbage**.

Proof. By the above theorem, T is **locally confluent modulo garbage**, so by the Newman-Garbage Lemma (Theorem 4.1), T is **confluent modulo garbage** as required. □

4.3 Extended Flow Diagrams

In 1976, Farrow, Kennedy and Zucconi presented 'semi-structured flow graphs', defining a grammar with confluent reduction rules [27]. Plump has considered a restricted version of this language: 'extended flow diagrams' [42]. The reduction rules for 'extended flow diagrams' are not confluent, however we will see that they are confluent modulo garbage and terminating. Thus we have an efficient mechanism for testing for language membership, since we need not 'backtrack', just like in Theorem 3.1.

Definition 4.8. The language of **extended flow diagrams** is generated by $\boldsymbol{EFD} = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$ where $\mathcal{L}_V = \{\bullet, \square, \diamond\}$, $\mathcal{L}_E = \{t, f, \square\}$, $\mathcal{N}_V = \mathcal{N}_E = \emptyset$, $\mathcal{R} = \{seq, while, ddec, dec1, dec2\}$, and $S = \bullet\!\!\rightarrow\!\square\!\rightarrow\!\bullet$.

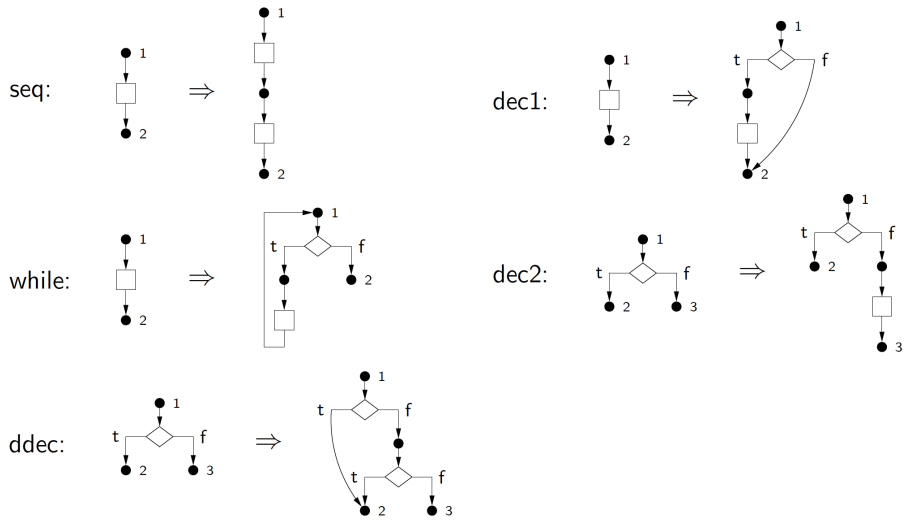


Figure 4.3: EFD Grammar Rules

Lemma 4.6. $EFD^{-1} = (\mathcal{L}, \mathcal{R}^{-1})$ is **terminating**. Moreover, it is **garbage separating** w.r.t. $\boldsymbol{L(EFD)}$.

Proof. Termination is clear since the rules are size reducing. Weak garbage separation can be seen by induction. □

Lemma 4.7. Every **directed cycle** in a graph in the **subgraph closure** of $\boldsymbol{L(EFD)}$ contains a t-labelled edge.

Proof. Induction. □

Now that we have all the intermediate results we need, we are ready to see that EFD^{-1} is not confluent, but is confluent modulo garbage. Moreover, that non-garbage critical pair analysis is sufficient to prove this!

Theorem 4.3 (EFD Recognition)**.** $EFD^{-1} = (\mathcal{L}, \mathcal{R}^{-1})$ is **confluent modulo garbage** w.r.t. $\boldsymbol{L(EFD)}$, but not **confluent**.

Proof. By Lemma 4.6 and the Newman-Garbage Lemma (Theorem 4.1), it suffices to show local confluent modulo garbage. EFD^{-1} has ten critical pairs [71], all but one of which are strongly joinable. Thus, we do not have confluence, however by Lemma 4.7, the non-joinable critical pair (Figure 4.4) is garbage, so by the Non-Garbage Critical Pair Lemma (Theorem 4.2), we have local confluence modulo garbage, as required. □

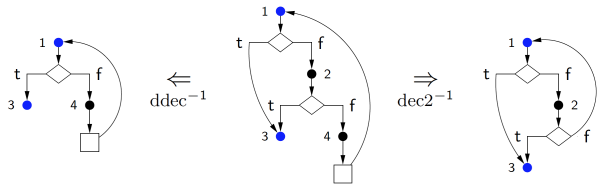


Figure 4.4: Non-Joinable Critical Pair

Remark 4.2. This special case of **weak garbage separation** with respect to the language we are recognising has actually been considered before by Bakewell [72]. He called this property **closedness**.

4.4 Encoding Partial Labelling

We now turn our attention to encoding partially labelled graphs and morphisms as totally labelled graphs and morphisms. The reason for doing this is that if we can show that our encoded rules are confluent modulo garbage, this must mean our original rules with relabelling were. Thus, we can attempt to determine local confluence of a system with relabelling by performing non-garbage critical pair analysis of the encoded rules!

Let $\mathcal{L} = (\mathcal{L}_V, \mathcal{L}_E)$ be an arbitrary label alphabet, and suppose without loss of generality (w.l.o.g.) that $\mathcal{L}_V \cap \mathcal{L}_E = \emptyset$ and $\{\square\} \notin \mathcal{L}_V \cup \mathcal{L}_E$. We will start by showing that partially labelled graphs (Definition C.2), morphisms, and rules can be encoded by totally labelled systems.

Definition 4.9. Let G be a **partially labelled graph** over $\mathcal{L}$, and w.l.o.g., suppose $V_G \cap E_G = \emptyset$. Define $e(G) = (V, E, s, t, l, m)$ where:

1. $V = V_G$
2. $E = E_G \cup l^{-1}(\mathcal{L}_V)$
3. $s(e) = \begin{cases} s_G(e) & \text{if } e \in E_G \\ e & \text{otherwise} \end{cases}$
4. $t(e) = \begin{cases} t_G(e) & \text{if } e \in E_G \\ e & \text{otherwise} \end{cases}$
5. $l(v) = \square$
6. $m(e) = \begin{cases} m_G(e) & \text{if } e \in E_G \\ l_G(e) & \text{otherwise} \end{cases}$

Proposition 4.3. $e(G)$ is a **totally labelled graph** over the **encoded label alphabet** $e(\mathcal{L}) = (\{\square\}, \mathcal{L}_V \cup \mathcal{L}_E)$.

Example 4.6. Let $\mathcal{L} = (\{x\}, \{y, z\})$. Then Figure 4.5 shows an example partially labelled graph and its encoding as a totally labelled graph. △

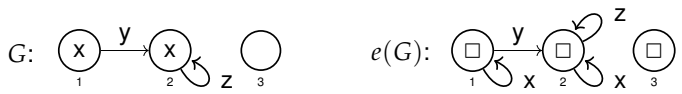


Figure 4.5: Example Encoded Partially Labelled Graph

Definition 4.10. Given two **partially labelled graphs** G, H, and a **morphism** $g : G \to H$, define $e(g) = (g'_V, g'_E)$ where:

1. $g'_V(v) = g_V(v)$
2. $g'_E(e) = \begin{cases} g_E(e) \text{ if } e \in E_G \\ g_V(e) \text{ otherwise} \end{cases}$

Proposition 4.4. Clearly, g is source/target/label preserving, and thus it is a **morphism** between the **totally labelled graphs** $e(G)$ and $e(H)$.

Lemma 4.8. e is an **injective functor** from the category of **partially labelled graphs** to the category of **totally labelled graphs**.

Proof. Clearly each graph and morphism has a distinct encoding, which is well-defined by Propositions 4.3 and 4.4. Clearly composition of morphisms behaves properly too. □

Theorem 4.4 (Partial Labelling Simulation)**.** Given a **rule** $r = \langle L \leftarrow K \rightarrow R \rangle$ where L and R are totally labelled graphs, and K partially labelled, then for all **totally labelled graphs** G, H, $G \Rightarrow_r H$ iff $e(G) \Rightarrow_{e(r)} e(H)$.

Proof. Firstly, given a fixed graph G, every injective morphism $g : L \to G$ satisfying the dangling condition can be encoded. Its encoding must also be injective (since encoding is an injective functor), and it is easy to check it must also satisfy the dangling condition. To see the other inclusion, suppose there was an injective morphism satisfying the dangling condition in the encoded system $g' : e(L) \to e(G)$. Then, we must be able to decode the morphism, to give an injective morphism. Again, it is easy to check the decoded morphism satisfies the dangling condition.

Finally, it is routine to check that the encoding of result graph for each match is exactly the same as the encoded result graph, derived using the encoded system, by using the explicit definition of rule application. □

Corollary 4.4. Given a **GT system** $(\mathcal{L}, \mathcal{R})$, $(e(\mathcal{L}), e(\mathcal{R}))$ is **weakly garbage separating** with respect to $e(\mathcal{G}(\mathcal{L}))$.

Proof. By the theorem, the encoded system can only derive encoded totally labelled graphs from encoded totally labelled graphs. □

Corollary 4.5. The **GT system** $(\mathcal{L}, \mathcal{R})$ is (**locally**) **confluent** (**terminating**) iff $(e(\mathcal{L}), e(\mathcal{R}))$ is (**locally**) **confluent** (**terminating**) **modulo garbage** with respect to $e(\mathcal{G}(\mathcal{L}))$.

Proof. By the theorem, we have a correspondence between derivations and derivations in the encoded system, so it is immediate that these notions line up with the notions in the encoded system. □

4.5 Tree Recognition Revisited

It is possible to rephrase the results from Section 3.2 in terms of our new notion of garbage:

Proposition 4.5. Let $\mathcal{L} = (\{\square, \triangle\}, \{\square\})$, $\mathcal{R} = \{r_0, r_1, r_2\}$, where the rules are as in Figure 3.2. Then, $T = (\mathcal{L}, \mathcal{R})$ is **garbage separating** w.r.t. to $D = \{[G] \in \mathcal{G}^{\oplus}(\mathcal{L}) \mid [G^{\ominus}] \in \boldsymbol{L}(\boldsymbol{TREE}), |p_G^{-1}(\{1\})| = 1\}$ and **confluent modulo garbage** w.r.t. $E = \{[G] \in D \mid l_G(V_G) = \{\square\}\}$.

Proof. Garbage separation is due to Lemma 3.2 and confluence modulo garbage due to Theorem 3.1. □

Finally, in Section 4.4 we have seen that we can encode a GT system with relabelling as a standard GT system (where interface graphs are totally labelled). One can pull a similar trick to encode rootedness of nodes, using looped edges with special labels. We give an encoding of the tree recognition rules from Figure 3.2: $T' = ((\{\square\}, \{R, N, M, \triangle\}), \{e_0, e_1, e_2\})$, where the rules are defined in Figure 4.6.

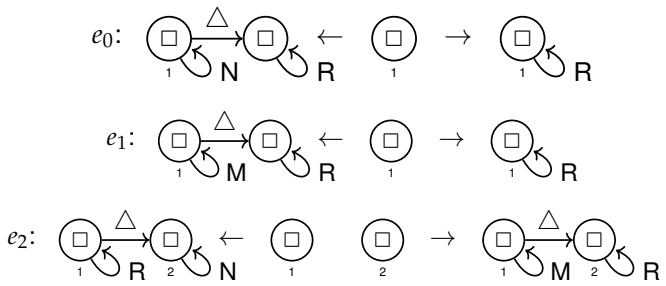


Figure 4.6: Encoded Tree Recognition Rules

One would hope that we could then perform non-garbage critical pair analysis on the encoding of D (where D is as in Proposition 4.5) in order to demonstrate local confluence modulo garbage of the original system. Every non-garbage critical pair is joinable, but unfortunately, one of them is not strongly joinable (Figure 4.7), so we are unable to make any conclusion about local confluence modulo garbage using the Non-Garbage Critical Pair Lemma (Theorem 4.2).

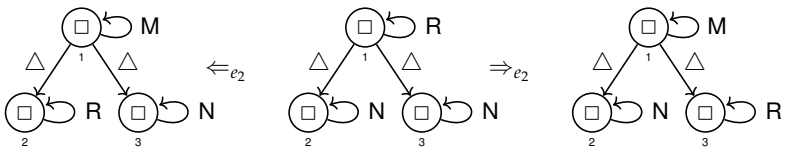


Figure 4.7: Non-Strongly Joinable Encoded Critical Pair

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

5 Conclusion

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 has been submitted for publication as part of [26]. We have also defined some notions of equivalence for our new type of graph transformation system, and briefly discussed a possible theory of refinement.

Finally, in Chapter 4, we have introduced the new notion of confluence modulo 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 modulo garbage. We have applied this to see that Extended Flow Diagrams (EFDs) can be recognised by a system that is confluent modulo garbage, and that we can rephrase the question of confluence of less well understood systems in terms of confluence modulo garbage of an encoded standard graph transformation system.

5.1 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 B and C. We also briefly reviewed 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 GP 2 and collecting timing results. We have submitted our program and results for publication [26].

Our final goal was to develop new confluence analysis theory. We have defined a new notion of confluence modulo 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 modulo garbage. We have applied this theory to EFDs and the encoding of partially labelled (rooted) GT systems as standard GT systems, performing non-garbage critical pair analysis on the encoded system. We look to publish this work.

5.2 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 GP 2, 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 2.5 remains open too.

Further exploring the relationship between (local) confluence modulo garbage and weak garbage separation 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 modulo garbage where it was previously not determined, remains future work.

Additional future work in the foundations of our new theory of rooted graph transformation would be to attempt to establish if the Local Church-Rosser and Parallelism Theorems hold [25], which have applications in database systems [73] and algebraic specifications [74]. It has been shown by Habel and Plump that this is the case with only relabelling [18]. It is likely that our new system with root nodes is $\mathcal{M},\mathcal{N}$-adhesive. Moreover, showing an analogy to the Extension Theorem and Critical Pair Lemma [75] would be excellent. Based on Section 4.4, we think that this is possible.

Finally, it remains open research, to explore the overlap between graph transformation systems and the study of ‘reversible computation’ [76]. 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.

A Mathematical Prelude

A.1 Sets I

There is not time to develop ZF(C) Set Theory and the foundational logic required. For the most part, a naive approach will suffice. We split the 'Sets' section into two halves. This section is derived from [77].

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

A.2 Functions

This section is derived from Chapter 3 of [77] and Chapter 1 of [78]. We use the conventional order of composition.

Definition A.4. Let A, B be sets. A **function** f from A to 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.5. 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.6. 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.7. For any set A, the **identity function** on A, $I_A : A \to A$ is defined by $\forall a \in A, I_A(a) = a$.

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

Definition A.8. Let $f : A \to B$ be a **function**. Then a **function** $g : B \to A$ is the **inverse** of f iff $g \circ f = I_A$ and $f \circ g = I_B$

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.9. 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.10. 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.11. 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.12. 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 [77], Chapter 1 of [78] and Appendix A of [68].

Definition A.13. Let A be a set. Then a **binary relation** on A is a subset R of $A \times A$. For any $a, b \in A$, we write aRb iff $(a, b) \in R$.

Definition A.14. 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.15. 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.16. Let A be a set, and R, S be **binary relations** on A. Then the **composition** of R and S is $S \circ R = \{(x, y) \in A \times A \mid \exists z \in A$ with xRz and $zSy\}$. Define $R^0 = \iota_A$, and $\forall n \in \mathbb{N}^+, R^n = R \circ R^{n-1}$.

Definition A.17. 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.18. 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.19. 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.20. Given an **equivalence** $\sim$, define $A/\sim = \{[a] \mid a \in A\}$.

A.4 Orders

This section is derived from Chapter 1 of [78] and Appendix A and of [68].

Definition A.21. An **antisymmetric preorder** $\leq$ on X is called a **partial order**, and we call $(X, \leq)$ a **partially ordered set** (**poset**).

Definition A.22. A **partial order** satisfying the **connex** property is called a **total order**, giving a **totally ordered set**.

Definition A.23. 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.24. 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.25. 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.26. Let $(X, \leq_X)$, $(Y, \leq_Y)$ be posets. Then a function $\varphi : X \to Y$ 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 [79], Part I of [80], Chapter 8 of [81], and Chapter 1 of [82].

Theorem A.1 (Well-Ordered Sets)**.** Every set can be **well-ordered**, and every **well-ordered** set is **isomorphic** to an **ordinal** (see [79] for details).

Proposition A.8. We define the **cardinality** of A ($|A|$), to be the **least ordinal** κ such that there is some **bijection** $f : A \to \kappa$. 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.27. A **partial function** $f : \mathbb{N} \to \mathbb{N}$ is **computable** iff there exists a **Turing Machine** that computes f (see [81] for details).

Definition A.28. A **countable** set $A \subseteq \mathbb{N}$ has **characteristic function** $\chi_A : \mathbb{N} \to \{0,1\}$ defined by $\forall x \in \mathbb{N}, \chi_A(x) = 1$ iff $x \in A$. A is **decidable** or **recursive** iff χ_A is **computable**. Otherwise, A is **undecidable**.

A.6 Categories

This section is derived from Chapter 1 of [83] and Chapter 1 of [84].

Definition A.29. A **category** consists of the following data:

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

such that for all $f : A \to B, g : B \to C, h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$ and $f \circ 1_A = f = 1_B \circ f$. Moreover, it is **locally small** iff the collection of arrows between any two objects is a **set**.

Definition A.30. A **functor** $F : \boldsymbol{C} \to \boldsymbol{D}$ between categories $\boldsymbol{C}$, $\boldsymbol{D}$ is a mapping such that:

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

B Abstract Reduction Systems

The definitions and theorems in this appendix are derived from Chapter 2 of [68], Section 2.2 of [85], and Section 1.1 of [86].

B.1 Basic Definitions

Definition B.1. An **abstract reduction system** (ARS) is a pair $(A, \to)$ where A is a **set** and $\to$ a **binary relation** on A.

Definition B.2. Let $(A, \to)$ be an ARS. We define the notation:

1. **Composition**: $\xrightarrow{n} := \to^n$ $(n \geq 0)$.
2. **Transitive closure**: $\xrightarrow{+} := \bigcup_{n \geq 1} \xrightarrow{n}$.
3. **Reflexive transitive closure**: $\xrightarrow{*} := \xrightarrow{+} \cup \xrightarrow{0}$.
4. **Reflexive closure**: $\xrightarrow{=} := \to \cup \xrightarrow{0}$.
5. **Inverse**: $\leftarrow := \to^{-1}$.
6. **Symmetric closure**: $\leftrightarrow := \to \cup \leftarrow$.

Remark B.1. It is usual that $\to$ is **decidable**. This **does not** imply that $\xrightarrow{+}$ is **decidable**, only that $\xrightarrow{n}$ is **decidable**.

Definition B.3. Let $(A, \to)$ be an ARS. We say that:

1. x is **reducible** iff there is a y s.t. $x \to y$.
2. x is **in normal form** iff x is not reducible.
3. y is **a normal form** of x iff $x \xrightarrow{*} y$ 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 $x \xrightarrow{+} y$, and a **direct successor** iff $x \to y$.
5. x and y are **joinable** iff there is a z s.t. $x \xrightarrow{*} z \xleftarrow{*} y$. We write $x \downarrow y$.

Definition B.4. Let $(A, \to)$ be an ARS. Then $\to$ is called:

1. **Church-Rosser** iff $x \xleftrightarrow{*} y$ implies $x \downarrow y$.
2. **Semi-confluent** iff $y_1 \leftarrow x \xrightarrow{*} y_2$ implies $y_1 \downarrow y_2$.
3. **Confluent** iff $y_1 \xleftarrow{*} x \xrightarrow{*} y_2$ implies $y_1 \downarrow y_2$.
4. **Terminating** iff there is no infinite descending chain $x_0 \to x_1 \to \ldots$.
5. **Normalising** iff every element has a normal form.
6. **Convergent** iff it is both confluent and terminating.

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

B.2 Noetherian Induction

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

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

$$\frac{\forall x \in A, (\forall y \in A, x \xrightarrow{+} y \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in A, P(x)}$$

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

1. The principle of **Noetherian induction** holds.
2. $\to$ is **well-founded** (Definition A.25).
3. $\to$ is **terminating** (Definition B.4).

Definition B.6. Let $(A, \to)$ be an ARS. Then $\to$ 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 $a \xrightarrow{+} a$.

Lemma B.1. Let $(A, \to)$ be an ARS. Then:

1. If $\to$ is **finitely branching** and **terminating**, then it is **globally finite**.
2. If $\to$ is **acyclic** and **globally finite**, then it is **terminating**.
3. $\to$ is **acyclic** iff $\xrightarrow{+}$ is a **strict order**.

B.3 Confluence and Termination

Theorem B.2 (Church-Rosser)**.** Let $(A, \to)$ be an ARS. Then, $\to$ has the **Church-Rosser** property iff it is **semi-confluent** iff it is **confluent**.

Theorem B.3 (Normal Forms)**.** Let $(A, \to)$ be an ARS. If $\to$ is **confluent**, then every element has at most one **normal form**. Moreover, if $\to$ is **confluent** and **normalising**, then $x \xleftrightarrow{*} y$ iff $x{\downarrow} = y{\downarrow}$.

Definition B.7. Let $(A, \to)$ be an ARS. Then $\to$ is called **locally confluent** iff $y_1 \leftarrow x \to y_2$ implies $y_1 \downarrow y_2$.

Theorem B.4 (Newman's Lemma)**.** A **terminating** relation is **confluent** iff it is **locally confluent**.

Lemma B.2. A **finitely branching** reduction **terminates** iff there is a **monotone** (Definition A.26) embedding into $(\mathbb{N}, >)$.

C Graph Transformation

We give a quick introduction to the theory of algebraic graph transformation derived from my earlier literature review [87], which is in turn derived from [5]. We generalise to **partially labelled graphs** using [17] and [71], in that we allow **relabelling** of **totally labelled graphs**.

The additional section on pushouts and pullbacks is derived from [5] and [71], the section on critical pair analysis is derived from [71], and the section on rooted graphs is derived from [16]. The definition of an unlabelled graph can be found in Section 1.1. The proof of Theorem C.4 is given by [17], and of Theorem C.5 is given by the proof of Theorem 1.1.

C.1 Partially Labelled Graphs

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

$$\mathcal{L}_E \xleftarrow[m]{} E \underset{t}{\overset{s}{\rightrightarrows}} V \xrightarrow{l} \mathcal{L}_V$$

Figure C.1: Partially Labelled Graph Diagram

Remark C.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 C.3. We say that a **partially labelled graph** G is **totally labelled** iff l_G is total.

Definition C.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 C.5. 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 C.6. 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 C.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 C.7. 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 $\mathcal{L}$.

C.2 Typed Graphs

Definition C.8. A **typed graph** is the tuple $G_T = (G, type_G)$ where G is an **unlabelled graph**, and $type_G$ 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 C.9. Given two **typed graphs** G_T, H_T, a **typed graph morphism** is an **unlabelled graph morphism** $f : G \to H$ such that $type_H \circ f = type_G$.

Theorem C.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 $\mathcal{L}$.

C.3 Performance Assumptions

We will assume that graphs are stored in a format such that the time complexities of various problems are as given in the table [24].

Input	Output	Time
label l	The set X of nodes with label l.	$O(\|X\|)$
node v	Values $deg(v)$, $indeg(v)$, $outdeg(v)$.	$O(1)$
node v, label l	No. nodes with source v, label l.	$O(1)$
node v, label l	No. nodes with target v, label l.	$O(1)$
node v, label l	Set X of nodes with source v, label l.	$O(\|X\|)$
node v, label l	Set X of nodes with target v, label l.	$O(\|X\|)$
graph G	$\|V_G\|$ and $\|E_G\|$.	$O(1)$

Figure C.2: Complexity Assumptions Table

C.4 Pushouts and Pullbacks

Pushouts and pullbacks are limits, in the sense of category theory. Our definitions are for any category (Definition A.29), and the propositions hold in the category of unlabelled concrete graphs, but not necessarily in others.

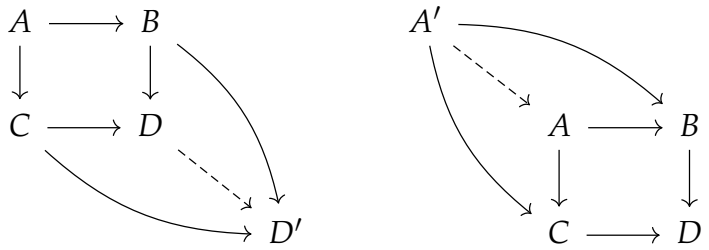


Figure C.3: Pushout and Pullback

Definition C.10. Given graph morphisms $A \to B$ and $A \to C$, a graph D together with graph morphisms $B \to D$ 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 morphisms $B \to D'$, $C \to D'$ such that $A \to B \to D' = A \to C \to D'$, there is a unique morphism $D \to D'$ such that $C \to D \to D' = B \to D'$ and $C \to D \to D' = C \to D'$.

Proposition C.1. Every pushout satisfies the following:

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

Definition C.11. Given graph morphisms $B \to D$ and $C \to D$, a graph A together with graph morphisms $A \to B$ 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 morphisms $A' \to B$, $A' \to C$ such that $A' \to B \to D = A' \to C \to D$, there is a unique morphism $A' \to A$ such that $A' \to A \to B = A' \to B$ and $A' \to A \to C = A' \to C$.

Definition C.12. A **pushout** that is a **pullback** is called a **natural pushout**.

Proposition C.2. A pushout is **natural** if $A \to B$ is injective.

Theorem C.2 (Limit Uniqueness)**.** In any category, if they exist, in a **pushout** (**pullback**), D (A) are unique up isomorphism.

Definition C.13. Given graph morphisms $A \to B$ and $B \to D$, a (natural) **pushout complement** is a graph C together with morphisms $A \to C$ and $C \to D$ such that the resulting square is a (natural) **pushout**.

Theorem C.3 (Limit Existence)**.** In the category of unlabelled graphs, **pushouts**, **pushout complements**, and **pullbacks** always exist.

C.5 Rules and Derivations

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

Definition C.14. 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** $K \hookrightarrow L$ and $K \hookrightarrow R$.

Definition C.15. We define the **inverse rule** to be $r^{-1} = \langle R \leftarrow K \rightarrow L \rangle$.

Definition C.16. If $r = \langle L \leftarrow K \rightarrow R \rangle$ is a **rule**, then $|r| = max\{|L|, |R|\}$.

Definition C.17. Given a **rule** $r = \langle L \leftarrow K \rightarrow R \rangle$ and a **totally labelled graph** G, we say that an **injective** morphism $g : L \hookrightarrow G$ satisfies the **dangling condition** iff no edge in $G \setminus g(L)$ is incident to a node in $g(L \setminus K)$.

Definition C.18. To **apply** a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ to some **totally labelled graph** G, find an **injective** graph morphism $g : L \hookrightarrow G$ 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.

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 C.19. 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 C.4 a **direct derivation**, and the injective morphism h the **comatch**.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow{\scriptstyle g} & & \downarrow{\scriptstyle d} & & \downarrow{\scriptstyle h} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Figure C.4: Direct Derivation

Theorem C.4 (Derivation Uniqueness)**.** 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 C.20. Given a rule set $\mathcal{R}$, we define $\mathcal{R}^{-1} = \{r^{-1} \mid r \in \mathcal{R}\}$.

Definition C.21. 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 C.22. 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 $G \Rightarrow^{+}_{\mathcal{R}} H$.

C.6 Transformation Systems

Definition C.23. 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 $\mathcal{L}$.

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

Definition C.24. Given a **graph transformation system** $T = (\mathcal{L}, \mathcal{R})$, we define the inverse system $T^{-1} = (\mathcal{L}, \mathcal{R}^{-1})$.

Definition C.25. 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 $\mathcal{P}_E \subseteq \mathcal{L}_E$.

Definition C.26. 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 $\boldsymbol{G} = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$.

Definition C.27. Given a **graph grammar** $\boldsymbol{G}$ 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 $\boldsymbol{G}$:

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

Proposition C.4. Given a **graph grammar** $\boldsymbol{G} = (\mathcal{L}, \mathcal{N}, \mathcal{R}, S)$, $G \Rightarrow_r H$ iff $H \Rightarrow_{r^{-1}} G$, for some $r \in \mathcal{R}$ (simply use the comatch). Moreover, $[G] \in \boldsymbol{L}(\boldsymbol{G})$ iff $G \Rightarrow^*_{\mathcal{R}^{-1}} S$ and G is terminally labelled.

Remark C.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.

C.7 Confluence and Termination

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

Definition C.28. The graphs H_1, H_2 are **joinable** iff there is a graph M such that $H_1 \Rightarrow^*_{\mathcal{R}} M \Leftarrow^*_{\mathcal{R}} H_2$.

Definition C.29. 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 C.30. 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 C.31. T is **terminating** iff there is no infinite derivation sequence $G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} G_3 \Rightarrow_{\mathcal{R}} \cdots$.

Theorem C.5 (Property Undecidability)**.** Testing if T has (**local**) **confluence** or is **terminating** is **undecidable** in general.

C.8 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 C.32. The derivations $G_1 \Rightarrow_{r_1,g_1} H \Rightarrow_{r_2,g_2} G_2$ are **sequentially independent** iff $(h_1(R_1) \cap g_2(L_2)) \subseteq (h_1(K_1) \cap g_2(K_2))$.

Lemma C.1. The derivations $G_1 \Rightarrow_{r_1,g_1} H \Rightarrow_{r_2,g_2} G_2$ are **sequentially independent** iff there exist morphisms $R_1 \to D_2$ and $L_2 \to D_1$ with $R_1 \to D_1 \to H = R_1 \to H$ and $L_1 \to D_2 \to H = L_2 \to H$.

Theorem C.6 (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 $\overline{H}$ and **sequentially independent** steps $G \Rightarrow_{r_2} \overline{H} \Rightarrow_{r_1} G_2$.

Definition C.33. The derivations $H_1 \Leftarrow_{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are **parallelly independent** iff $(g_1(L_1) \cap g_2(L_2)) \subseteq (g_1(K_1) \cap g_2(K_2))$.

Lemma C.2. The derivations $H_1 \Leftarrow_{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are **parallelly independent** iff there exist morphisms $L_1 \to D_2$ and $L_2 \to D_1$ with $L_1 \to D_2 \to G = L_1 \to G$ and $L_2 \to D_1 \to G = L_2 \to G$.

Lemma C.3. 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**.

Theorem C.7 (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 $\overline{G}$ and **direct derivations** $H_1 \Rightarrow_{r_2} \overline{G} \Leftarrow_{r_1} H_2$ with $G \Rightarrow_{r_1} H_1 \Rightarrow_{r_2} \overline{G}$ and $G \Rightarrow_{r_2} H_2 \Rightarrow_{r_1} \overline{G}$ **sequentially independent**.

Definition C.34. A pair of **direct derivations** $G_1 \Leftarrow_{r_1,g_1} H \Rightarrow_{r_2,g_2} G_2$ 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 $g_1 \neq g_2$.

Definition C.35. Let $G \Rightarrow H$ be a **direct derivation**. Then the **track morphism** is defined to be the partial morphism $tr_{G \Rightarrow H} = in' \circ in^{-1}$. We define $tr_{G \Rightarrow^* H}$ inductively as the composition of track morphisms.

Definition C.36. The set of **persistent nodes** of a critical pair $\Phi : H_1 \Leftarrow G \Rightarrow H_2$ is $Persist_\Phi = \{v \in G_V \mid tr_{G \Rightarrow H_1}(\{v\}), tr_{G \Rightarrow H_2}(\{v\}) \neq \emptyset\}$.

Definition C.37. A critical pair $\Phi : H_1 \Leftarrow G \Rightarrow H_2$ is **strongly joinable** iff there exists a graph M, a derivation $H_1 \Rightarrow_{\mathcal{R}}^* M \Leftarrow_{\mathcal{R}}^* H_2$ and:

$$\forall v \in Persist_\Phi, tr_{G \Rightarrow H_1 \Rightarrow^* \overline{G}}(\{v\}) = tr_{G \Rightarrow H_2 \Rightarrow^* \overline{G}}(\{v\}) \neq \emptyset$$

Theorem C.8 (Critical Pair Lemma)**.** A graph transformation system T is **locally confluent** if all its **critical pairs** are **strongly joinable**.

Remark C.4. Every graph transformation system has, up to isomorphism, only finitely many critical pairs. Thus, the reverse direction of this theorem is false, as this would contradict the undecidability of checking for confluence.

C.9 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 C.38. 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 $\widehat{G} = (G, P_G)$.

Definition C.39. Given two **rooted partially labelled graphs** $\widehat{G}, \widehat{H}$, a **partially labelled graph morphism** $g : G \to H$ is a **rooted labelled graph morphism** $\widehat{G} \to \widehat{H}$ iff $g_V(P_G) \subseteq P_H$. A morphism $g : \widehat{G} \to \widehat{H}$ is **injective**/**surjective** iff the underlying graph morphism is injective/surjective. **Inclusion morphisms** and **subgraphs** are defined in the obvious way.

Definition C.40. We say that **rooted partially labelled graphs** $\widehat{G}, \widehat{H}$ are **isomorphic** iff there exists an **injective**, **surjective** morphism $g : \widehat{G} \to \widehat{H}$ such that $g^{-1} : \widehat{H} \to \widehat{G}$ is also a morphism, and we write $\widehat{G} \cong \widehat{H}$. This naturally gives rise to **equivalence classes** $[\widehat{G}]$: the **countably** many rooted partially labelled abstract graphs over some fixed $\mathcal{L}$.

Definition C.41. 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 C.18):

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 $\widehat{G} \Rightarrow_{r,g} \widehat{M}$ for a successful application of r to $\widehat{G}$ using match g, obtaining result $\widehat{H} \cong \widehat{M}$. We call this a **direct derivation**. Definitions C.21 and C.22 are analogous.

Theorem C.9 (Rooted Derivation Uniqueness)**.** The **result graph** of a **direct derivation** is **unique up to isomorphism** and is **totally labelled**.

Definition C.42. A **rooted graph transformation system** $\widehat{T} = (\mathcal{L}, \widehat{\mathcal{R}})$, consists of a label alphabet $\mathcal{L}$, and a **finite** set $\widehat{\mathcal{R}}$ of rules over $\mathcal{L}$.

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

Definition C.43. Given a **rooted graph transformation system** $\widehat{T} = (\mathcal{L}, \widehat{\mathcal{R}})$, a subalphabet of **non-terminals** $\mathcal{N}$, and a **start graph** $\widehat{S}$ over $\mathcal{L}$, then a **rooted graph grammar** is the system $\widehat{\boldsymbol{G}} = (\mathcal{L}, \mathcal{N}, \widehat{\mathcal{R}}, \widehat{S})$.

Definition C.44. Given a **rooted graph grammar** $\widehat{\boldsymbol{G}}$ as defined above, we say that a graph $\widehat{G}$ 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**:

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

D 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 C. The definitions and theorems in this appendix have been adapted from [88], [89], Chapter 1 of [90], and Chapter 3 of [91].

D.1 Basic Definitions

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

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

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

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

Definition D.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 D.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: $\langle v_0, e_0, v_1, e_1, \ldots, e_{k-1}, v_k \rangle$, 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 $t_G(e_i) = v_i$].
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 D.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 D.1 (Graph Decomposition)**.** Every concrete graph G has a unique decomposition into **connected components**.

Definition D.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 D.1. Given a concrete graph G, $v \in V_G$. Then:

1. $\text{children}_G(v) \subseteq \text{N}_G(v)$ and $\text{parents}_G(v) \subseteq \text{N}_G(v)$.
2. $|\text{children}_G(v)| \leq \text{outdeg}_G(v)$ and $|\text{parents}_G(v)| \leq \text{indeg}_G(v)$.

D.2 Classes of Graphs

Definition D.7. A graph is called **discrete** iff it has no edges.

Definition D.8. 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 **perfect binary tree** is a **binary tree** such that every node has either 0 or 2 children and every **maximal path** is the same length.
4. A **forest** is a graph where each **connected component** is a **tree**.

Definition D.9. A $n \times m$-**grid graph** is a graph with underlying unlabelled graph isomorphic to (V, E, s, t) where $V = \mathbb{Z}_n \times \mathbb{Z}_m$, $E = (\mathbb{Z}_2 \times V) \setminus \{(0, i, m-1), (1, n-1, j) \mid i \in \mathbb{Z}_n, j \in \mathbb{Z}_m\}$, $s(d, i, j) = (i, j)$, and $t(d, i, j) = (i + d, j + 1 - d)$. We call such a graph **square** iff $n = m$.

Definition D.10. 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:

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

An example linked list, perfect binary tree, square grid graph, and star graph can be found in Figure 3.5.

Bibliography

[1] H. Ehrig, M. Pfender and H. J. Schneider, 'Graph-grammars: An algebraic approach', in *14th Annual Symposium on Switching and Automata Theory (SWAT 1973)*, IEEE, 1973, pp. 167–180. DOI: 10.1109/SWAT.1973.11.

[2] H. Ehrig, 'Introduction to the algebraic theory of graph grammars (a survey)', in *Graph-Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 73, Springer, 1979, pp. 1–69. DOI: 10.1007/BFb0025714.

[3] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel and M. Löwe, 'Algebraic approaches to graph transformation. Part I: Basic concepts and double pushout approach', in *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. 1, World Scientific, 1997, pp. 163–245. DOI: 10.1142/3303.

[4] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, 'Algebraic approaches to graph transformation. Part II: Single pushout approach and comparison with double pushout approach', in *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. 1, World Scientific, 1997, pp. 247–312. DOI: 10.1142/3303.

[5] H. Ehrig, K. Ehrig, U. Prange and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. DOI: 10.1007/3-540-31188-2.

[6] A. Corradini, F. Rossi and F. Parisi-Presicce, 'Logic programming as hypergraph rewriting', in *TAPSOFT '91*, S. Abramsky and T. S. E. Maibaum, Eds., ser. Lecture Notes in Computer Science, vol. 493, Springer, 1991, pp. 275–295. DOI: 10.1007/3-540-53982-4_16.

[7] *Graph Transformation*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2505. DOI: 10.1007/3-540-45832-8.

[8] *Graph Transformations*, ser. Lecture Notes in Computer Science. Springer, 2004, vol. 3256. DOI: 10.1007/b100934.

[9] A. Schürr, A. Winter and A. Zündorf, 'The PROGRES approach: Language and environment', in *Handbook of Graph Grammars and Computing by Graph Transformation*, H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, Eds., vol. 2, World Scientific, 1999, pp. 487–550. DOI: 10.1142/4180.

[10] O. Runge, C. Ermel and G. Taentzer, 'AGG 2.0 — new features for specifying and analyzing algebraic graph transformations', in *Proc. Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011)*, ser. Lecture Notes in Computer Science, vol. 7233, Springer, 2012, pp. 81–88. DOI: 10.1007/978-3-642-34176-2_8.

[11] A. Agrawal, G. Karsai, S. Neema, F. Shi and A. Vizhanyo, 'The design of a language for model transformations', *Software and System Modeling*, vol. 5, no. 3, pp. 261–288, 2006. DOI: 10.1007/s10270-006-0027-7.

[12] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon and M. Zimakova, 'Modelling and analysis using GROOVE', *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 1, pp. 15–40, 2012. DOI: 10.1007/s10009-011-0186-x.

[13] E. Jakumeit, S. Buchwald and M. Kroll, 'GrGen.NET - the expressive, convenient and fast graph rewrite system', *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 3–4, pp. 263–271, 2010. DOI: 10.1007/s10009-010-0148-8.

[14] D. Plump, 'The design of GP 2', in *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 82, 2011, pp. 1–16. DOI: 10.4204/EPTCS.82.1.

[15] H. Dörr, *Efficient Graph Rewriting and its Implementation*, ser. Lecture Notes in Computer Science. Springer, 1995, vol. 922. DOI: 10.1007/BFb0031909.

[16] C. Bak and D. Plump, 'Rooted graph programs', *Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, Electronic Communications of the EASST, vol. 54, 2012. DOI: 10.14279/tuj.eceasst.54.780.

[17] A. Habel and D. Plump, 'Relabelling in graph transformation', in *Graph Transformation*, A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 2505, Springer, 2002, pp. 135–147. DOI: 10.1007/3-540-45832-8_12.

[18] ——, '$\mathcal{M},\mathcal{N}$-adhesive transformation systems', in *Graph Transformations*, H. Ehrig, G. Engels, H.-J. Kreowski and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 7562, Springer, 2012, pp. 218–233. DOI: 10.1007/978-3-642-33654-6_15.

[19] M. Dodds and D. Plump, 'Graph transformation in constant time', in *Proc. International Conference on Graph Transformation (ICGT 2006)*, ser. Lecture Notes in Computer Science, vol. 4178, Springer, 2006, pp. 367–382. DOI: 10.1007/11841883_26.

[20] D. Plump, 'Checking graph-transformation systems for confluence', *ECEASST*, vol. 26, 2010. DOI: 10.14279/tuj.eceasst.26.367.

[21] ——, 'Hypergraph rewriting: Critical pairs and undecidability of confluence', in *Term Graph Rewriting*, M. R. Sleep, M. J. Plasmeijer and M. C.J. D. van Eekelen, Eds., John Wiley and Sons, 1993, pp. 201–213.

[22] H. Ehrig, L. Lambers and F. Orejas, 'Efficient conflict detection in graph transformation systems by essential critical pairs', *Electronic Notes in Theoretical Computer Science*, vol. 211, pp. 17–26, 2008. DOI: 10.1016/j.entcs.2008.04.026.

[23] I. Hristakiev, 'Confluence analysis for a graph programming language', PhD thesis, Department of Computer Science, University of York, UK, 2018. [Online]. Available: https://etheses.whiterose.ac.uk/20255/.

[24] M. Dodds, 'Graph transformation and pointer structures', PhD thesis, Department of Computer Science, University of York, UK, 2008. [Online]. Available: https://www.cs.york.ac.uk/plasma/publications/pdf/DoddsThesis.08.pdf.

[25] H. Ehrig, U. Golas, A. Habel, L. Lambers and F. Orejas, '$\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 1: Parallelism, concurrency and amalgamation', *Mathematical Structures in Computer Science*, vol. 24, no. 4, p. 240 406, 2014. DOI: 10.1017/S0960129512000357.

[26] G. Campbell, B. Courtehoute and D. Plump, 'Linear-time graph algorithms in GP 2', Department of Computer Science, University of York, UK, Submitted for publication, 2019. [Online]. Available: https://cdn.gjcampbell.co.uk/2019/Linear-Time-GP2-Preprint.pdf.

[27] R. Farrow, K. Kennedy and L. Zucconi, 'Graph grammars and global program data flow analysis', in *17th Annual Symposium on Foundations of Computer Science (SFCS 1976)*, IEEE, 1976, pp. 42–56. DOI: 10.1109/SFCS.1976.17.

[28] F. Drewes, H.-J. Kreowski and A. Habel, 'Hyperedge replacement graph grammars', in *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. 1, World Scientific, 1997, pp. 95–162. DOI: 10.1142/3303.

[29] J. Engelfriet and G. Rozenberg, 'Node replacement graph grammars', in *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. 1, World Scientific, 1997, pp. 1–94. DOI: 10.1142/3303.

[30] T. Uesu, 'A system of graph grammars which generates all recursively enumerablesets of labelled graphs', *Tsukuba Journal of Mathematics*, vol. 2, pp. 11–26, 1978.

[31] M. Löwe, M. Korff and A. Wagner, 'Term graph rewriting', M. R. Sleep, M. J. Plasmeijer and M. C.J. D. van Eekelen, Eds., John Wiley and Sons, 1993, ch. An Algebraic Framework for the Transformation of Attributed Graphs, pp. 185–199.

[32] A. Corradini, U. Montanari and F. Rossi, 'Graph processes', *Fundamenta Informaticae*, vol. 26, no. 3,4, pp. 241–265, 1996.

[33] M. Berthold, I. Fischer and M. Koch, 'Attributed graph transformation with partial attribution', Technical University of Berlin, Germany, Tech. Rep., 2000.

[34] R. Heckel, J. M. Küster and G. Taentzer, 'Confluence of typed attributed graph transformation systems', in *Graph Transformations*, ser. Lecture Notes in Computer Science, vol. 2505, Springer, 2002, pp. 161–176.

[35] H. Ehrig, U. Prange and G. Taentzer, 'Fundamental theory for typed attributed graph transformation', in *Graph Transformations*, H. Ehrig, G. Engels, F. Parisi-Presicce and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 3256, Springer, 2004, pp. 161–177. DOI: 10.1007/978-3-540-30203-2_13.

[36] C. Bak, 'GP 2: Efficient implementation of a graph programming language', PhD thesis, Department of Computer Science, University of York, UK, 2015. [Online]. Available: https://etheses.whiterose.ac.uk/12586/.

[37] R. Geiß, G. V. Batz, D. Grund, S. Hack and A. Szalkowski, 'GrGen: A fast SPO-based graph rewriting tool', in *Graph Transformations*, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 4178, Springer, 2006, pp. 383–397. DOI: 10.1007/11841883_27.

[38] G. Campbell, J. Romo and D. Plump, 'Fast graph programs', Department of Computer Science, University of York, UK, Tech. Rep., 2018. [Online]. Available: https://www.cs.york.ac.uk/plasma/publications/pdf/CampbellRomoPlump.EPSRC.18.pdf.

[39] J. Hartmanis and R. E. Stearns, 'On the computational complexity of algorithms', *Transactions of the American Mathematical Society*, vol. 117, pp. 285–306, 1965.

[40] M. H. A. Newman, 'On theories with a combinatorial definition of "equivalence"', *Annals of Mathematics*, vol. 43, no. 2, pp. 223–243, 1942. DOI: 10.2307/1968867.

[41] D. Plump, 'Termination of graph rewriting is undecidable', *Fundamenta Informaticae*, vol. 33, no. 2, pp. 201–209, 1998. DOI: 10.3233/FI-1998-33204.

[42] ——, ‘Confluence of graph transformation revisited’, in *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, A. Middeldorp, V. van Oostrom, F. van Raamsdonk and R. de Vrijer, Eds., ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3838, pp. 280–308. DOI: 10.1007/11601548_16.

[43] M. A. Hannachi, I. Bouassida Rodriguez, K. Drira and S. E. Pomares Hernandez, ‘GMTE: A tool for graph transformation and exact/inexact graph matching’, in *Graph-Based Representations in Pattern Recognition*, W. G. Kropatsch, N. M. Artner, Y. Haxhimusa and X. Jiang, Eds., ser. Lecture Notes in Computer Science, vol. 7877, Springer, 2013, pp. 71–80. DOI: 10.1007/978-3-642-38221-5_8.

[44] J. R. W. Glauert, J. R. Kennaway and M. R. Sleep, ‘Dactl: An experimental graph rewriting language’, in *Graph Grammars and Their Application to Computer Science*, H. Ehrig, H.-J. Kreowski and G. Rozenberg, Eds., ser. Lecture Notes in Computer Science, vol. 532, Springer, 1991, pp. 378–395. DOI: 10.1007/BFb0017401.

[45] T. Arendt, E. Biermann, S. Jurack, C. Krause and G. Taentzer, ‘Henshin: Advanced concepts and tools for in-place EMF model transformations’, in *Model Driven Engineering Languages and Systems (MODELS 2010)*, ser. Lecture Notes in Computer Science, vol. 6394, Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

[46] M. Fernández, H. Kirchner, I. Mackie and B. Pinaud, ‘Visual modelling of complex systems: Towards an abstract machine for PORGY’, in *Proc. Computability in Europe (CiE 2014)*, ser. Lecture Notes in Computer Science, vol. 8493, Springer, 2014, pp. 183–193. DOI: 10.1007/978-3-319-08019-2_19.

[47] A. Habel and D. Plump, ‘Computational completeness of programming languages based on graph transformation’, in *Foundations of Software Science and Computation Structures*, F. Honsell and M. Miculan, Eds., ser. Lecture Notes in Computer Science, vol. 2030, Springer, 2001, pp. 230–245. DOI: 10.1007/3-540-45315-6_15.

[48] D. Plump, ‘The graph programming language GP’, in *Algebraic Informatics*, S. Bozapalidis and G. Rahonis, Eds., ser. Lecture Notes in Computer Science, vol. 5725, Springer, 2009, pp. 99–122. DOI: 10.1007/978-3-642-03564-7_6.

[49] ——, ‘Reasoning about graph programs’, *Electronic Proceedings in Theoretical Computer Science*, vol. 225, pp. 35–44, 2016. DOI: 10.4204/eptcs.225.6.

[50] A. Habel, J. Müller and D. Plump, ‘Double-pushout graph transformation revisited’, 5, vol. 11, Cambridge University Press, 2001, pp. 637–688. DOI: 10.1017/S0960129501003425.

[51] C. M. Poskitt and D. Plump, 'Hoare-style verification of graph programs', *Fundamenta Informaticae*, vol. 118, no. 1-2, pp. 135–175, 2012. DOI: 10.3233/FI-2012-708. [Online]. Available: https://www.cs.york.ac.uk/plasma/publications/pdf/PoskittPlump.FundInf.12.pdf.

[52] ——, 'Verifying total correctness of graph programs', in *Revised Selected Papers, Graph Computation Models (GCM 2012)*, ser. Electronic Communications of the EASST, vol. 61, 2013.

[53] C. Poskitt, 'Verification of graph programs', PhD thesis, Department of Computer Science, University of York, UK, 2013. [Online]. Available: https://etheses.whiterose.ac.uk/4700/.

[54] C. M. Poskitt and D. Plump, 'Verifying monadic second-order properties of graph programs', in *Graph Transformation*, H. Giese and B. König, Eds., ser. Lecture Notes in Computer Science, vol. 8571, Springer, 2014, pp. 33–48. DOI: 10.1007/978-3-319-09108-2_3.

[55] C. A. R. Hoare, 'An axiomatic basis for computer programming', *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259.

[56] G. Manning and D. Plump, 'The GP programming system', Electronic Communications of the ECEASST, vol. 10, 2008. DOI: 10.14279/tuj.eceasst.10.150.

[57] I. Hristakiev and D. Plump, 'Checking graph programs for confluence', in *Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 10748, Springer, 2018, pp. 92–108. DOI: 10.1007/978-3-319-74730-9_8.

[58] D. Plump, 'From imperative to rule-based graph programs', *Journal of Logical and Algebraic Methods in Programming*, vol. 88, pp. 154–173, 2017. DOI: 10.1016/j.jlamp.2016.12.001.

[59] G. Plotkin, 'A structural approach to operational semantics', *The Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 17–139, 2004. DOI: 10.1016/j.jlap.2004.05.001.

[60] E. W. Dijkstra, 'A constructive approach to the problem of program correctness', *BIT Numerical Mathematics*, vol. 8, no. 3, pp. 174–186, 1968. DOI: 10.1007/BF01933419.

[61] E. W. Dijkstra, 'Chapter I: Notes on structured programming', in *Structured Programming*, O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Eds., Academic Press, 1972, pp. 1–82.

[62] N. Wirth, 'Program development by stepwise refinement', *Communications of the ACM*, vol. 14, no. 4, pp. 221–227, 1971. DOI: 10.1145/362575.362577.

[63] S. Arnborg, B. Courcelle, A. Proskurowski and D. Seese, 'An algebraic theory of graph reduction', *Journal of the ACM*, vol. 40, no. 5, pp. 1134–1164, 1993. DOI: 10.1145/174147.169807.

[64] H. L. Bodlaender and B. van Antwerpen-de Fluiter, 'Reduction algorithms for graphs of small treewidth', *Information and Computation*, vol. 167, no. 2, pp. 86–119, 2001. DOI: 10.1006/inco.2000.2958.

[65] B. Courcelle, 'The monadic second-order logic of graphs: Definable sets of finite graphs', in *Graph-Theoretic Concepts in Computer Science*, J. van Leeuwen, Ed., ser. Lecture Notes in Computer Science, vol. 344, Springer, 1989, pp. 30–53. DOI: 10.1007/3-540-50728-0_34.

[66] D. Knuth and P. Bendix, 'Simple word problems in universal algebras', in *Computational Problems in Abstract Algebras*, Pergamon Press, 1970, pp. 263–297.

[67] G. Huet, 'Confluent reductions: Abstract properties and applications to term rewriting systems', *Journal of the ACM*, vol. 27, no. 4, pp. 797–821, 1980. DOI: 10.1145/322217.322230.

[68] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[69] Terese, *Term Rewriting Systems*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 55.

[70] J. Berstel, *Transductions and Context-Free Languages*. Vieweg+Teubner, 1979. DOI: 10.1007/978-3-663-09367-1.

[71] D. Plump, 'Computing by graph transformation: 2018/19', Department of Computer Science, University of York, UK, Lecture Slides, 2019.

[72] A. Bakewell, D. Plump and C. Runciman, 'Specifying pointer structures by graph reduction', Department of Computer Science, University of York, UK, Tech. Rep., 2003. [Online]. Available: https://www.cs.york.ac.uk/plasma/publications/pdf/BakewellPlumpRuncimanReport.03.pdf.

[73] H. Ehrig and H.-J. Kreowski, 'Applications of graph grammar theory to consistency, synchronization and scheduling in database systems', *Information Systems*, vol. 5, pp. 225–238, 1980.

[74] F. Parisi-Presicce, 'Modular system design applying graph grammars techniques', in *Automata, Languages and Programming*, G. Ausiello, M. Dezani-Ciancaglini and S. R. Della Rocca, Eds., ser. Lecture Notes in Computer Science, vol. 372, Springer, 1989, pp. 621–636. DOI: 10.1007/BFb0035788.

[75] H. Ehrig, U. Golas, A. Habel, L. Lambers and F. Orejas, '$\mathcal{M}$-adhesive transformation systems with nested application conditions. Part 2: Embedding, critical pairs and local confluence', *Fundamenta Informaticae*, vol. 118, no. 1–2, pp. 35–63, 2012. DOI: 10.3233/FI-2012-705.

[76] M. P. Frank, 'Introduction to reversible computing: Motivation, progress, and challenges', in *CF '05 Proceedings of the 2nd conference on Computing frontiers*, SCM, 2005, pp. 385–390.

[77] W. A. Sutherland, *Introduction to metric and topological spaces*, 2nd ed. Oxford University Press, 2009.

[78] J. Howie, *Fundamentals of Semigroup Theory*, ser. LMS monographs. Clarendon Press, 1995.

[79] W. A. R. Weiss, 'An introduction to set theory', Department of Mathematics, University of Toronto, Lecture Notes, 2008. [Online]. Available: https://www.math.toronto.edu/weiss/set_theory.pdf.

[80] J. R. Munkres, *Topology*, 2nd ed. Pearson, 2018.

[81] J. C. Martin, *Introduction to Languages and the Theory of Computation*, 4th ed. McGraw-Hill, 2011.

[82] K. Weihrauch, *Computable Analysis: An Introduction*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2013. DOI: 10.1007/978-3-642-56999-9.

[83] S. M. Lane, *Categories for the Working Mathematician*, 2nd ed., ser. Graduate Texts in Mathematics. Springer, 1978. DOI: 10.1007/978-1-4757-4721-8.

[84] S. Awodey, *Category Theory*, 2nd ed., ser. Oxford Logic Guides. Oxford University Press, 2010.

[85] D. Plump, 'Computing by graph rewriting', Habilitation Thesis, Universität Bremen, Fachbereich Mathematik und Informatik, 1999.

[86] R. V. Book and F. Otto, *String-rewriting Systems*, ser. Monographs in Computer Science. Springer, 1993. DOI: 10.1007/978-1-4613-9771-7.

[87] G. Campbell, 'Algebraic graph transformation: A crash course', Department of Computer Science, University of York, UK, Tech. Rep., 2018. [Online]. Available: https://cdn.gjcampbell.co.uk/2018/Graph-Transformation.pdf.

[88] P. Johnson, 'MAS341 graph theory', School of Mathematics and Statistics, University of Sheffield, UK, Lecture Notes, 2018. [Online]. Available: https://ptwiddle.github.io/MAS341-Graph-Theory-2017/lecturenotes/lecturenotes.pdf.

[89] V. Lozin, 'Graph theory notes', Institute of Mathematics, University of Warwick, UK, Lecture Notes, 2018. [Online]. Available: https://homepages.warwick.ac.uk/~masgax/Graph-Theory-notes.pdf.

[90] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*, 2nd ed., ser. Springer Monographs in Mathematics. Springer, 2009. DOI: 10.1007/978-1-84800-998-1.

[91] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer, 2008. DOI: 10.1007/978-1-84800-070-4.