Confluence Analysis for a Graph Programming Language

Ivaylo Stanislavov Hristakiev

PhD

University of York

Computer Science

September 2017

Abstract

GP 2 is a high-level domain-specific language for programming with graphs. Users write a set of graph transformation rules and organise them with imperative-style control constructs to perform a desired computation on an input graph. As rule selection and matching are non-deterministic, there might be different graphs resulting from program execution. Confluence is a property that establishes the global determinism of a computation despite possible local non-determinism. Conventional confluence analysis is done via so-called critical pairs, which are conflicts in minimal context. A key challenge is extending critical pairs to the setting of GP 2.

This thesis concerns the development of confluence analysis for GP 2. First, we extend the notion of conflict to GP 2 rules, and prove that non-conflicting rule applications commute. Second, we define symbolic critical pairs and establish their properties, namely that there are only finitely many of them and that they represent all possible conflicts. We give an effective procedure for their construction. Third, we solve the problem of unifying GP 2 list expressions, which arises during the construction of critical pairs, by giving a unification procedure which terminates with a finite and complete set of unifiers (under certain restrictions). Last but not least, we specify a confluence analysis algorithm based on symbolic critical pairs, and show its soundness by proving the Local Confluence Theorem. Several existing programs are analysed for confluence to demonstrate how the analysis handles several GP 2 features at the same time, and to demonstrate the merit of the used techniques.

Contents

List of Figures

Acknowledgements

First and foremost, I would like to thank my parents and family, for being there for me forever and always, for being patient with me throughout the eight years of university studies. Without their support, I would probably not be standing where I am today.

I am thankful to my supervisor, Dr Detlef Plump, for his support, guidance and boundless patience. Detlef's enthusiasm and interest in my work kept me on track even in my darkest moments, even when my I could not see the light. Our discussions were a great source of positivity and ideas that ultimately culminated in the completion of this research. His attention to detail, technical understanding and foresight had an enormous impression on myself and others. I also thank my examiners, Prof Colin Runciman and Prof Andy Schür, for their interest in my research and the useful feedback that ensured a very interesting viva!

I express my thanks to the Engineering and Physical Sciences Research Council who provided me with financial support during my studies. I am grateful to the Department of Computer Science at the University of York, both its staff and students, for providing a positive environment for conducting my research, and for providing the opportunities for me to engage with and teach numerous topics in the broad field to Computer Science. I thank the Programming Languages and Systems (PLASMA) group for providing a stimulating research environment. More specifically, I give my thanks to the following people: Chris Bak, for going out his way in welcoming me to the PLASMA group, later engaging in interesting discussions regarding my research field and also giving me the opportunity to contribute to the practical implementation of GP 2; Rudy Braquehais, for being a great friend throughout the duration of my research, which made my time at York that more enjoyable and fun; my office colleagues Matt Windsor, Tim Atkinson, José Calderón, among others, for enriching my life and providing a stimulating environment to discuss, grow and reflect.

Last but not least, I would like to thank all my University of York friends, for making the eight years of university life pass seamlessly.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. Some parts of this thesis have been previously published in journal, conference, and workshop papers; the details of which are given in Chapter 1.

Ivaylo Stanislavov Hristakiev, September 2017

Chapter 1

Introduction

1.1 Motivation

Graphs are natural way of specifying objects and their relationships in an intuitive way. They are used in many fields of computer science to model system state such as data and control flow diagrams, UML diagrams, Petri nets, hardware and software architectures [HT06, Men05]. Graphs and graph theory have been subject to extensive research [BG02].

Graph transformation and graph grammars add to the static description of graphs by allowing us to express their dynamic evolution by means of graph transformation rules. Graph grammars have been applied to many fields of computer science such as model-driven software engineering [EE08, EEHP09], database design [GPdBG94], compiler construction [Aßm96], network architectures, distributed systems [TGM98] and many more [EEKR99].

A rule describes local changes to a graph, and a sequence of such rule applications contain the intended computational thinking. Rules are similar to pattern-matching and have a formal mathematical definition. Graph transformation has been subject to formal verification and analysis.

Graph transformation is non-deterministic - several rules may be applicable at the same time. This means that for any starting graph, there may be several different result graphs (graphs to which none of the rules apply). However, sometimes it may be the case that there is only one possible result due to how the rules interact with each other. This special property is called *confluence* and comes from the area of term and term graph rewriting.

A proof that a set of rules is confluent is done by constructing so-called critical pairs, conflicts in minimal context. Critical pairs can be detected and analysed statically, i.e. before the program is executed, and allow us to reason about all conflicts that may arise during computation.

Graph transformation has been extended in several ways. When equipped with sequential composition and as-long-as-possible iteration, it becomes computationally complete [HP01] which has served as basis for implementing graph transformation languages, one of them being the language GP 2 [Plu12] developed at York. The language GP 2 is designed to be minimal and computationally complete with intuitive formal syntax and semantics. The language's features include application conditions on rules that enable or forbid a rule from being applicable, combining rules into programs (using composition, iteration, conditional branching, etc), label expressions, and others. All of these extensions increase the richness of interactions between rules, and hence the increased complexity when proving confluence.

Confluence analysis for GP 2 would be desirable for several reasons. If a given program is proven confluent, then only a single result graph should be computed, greatly increasing efficiency by disabling backtracking. Furthermore, GP 2 has the explicit notion of *failure*, and if one possible execution of a confluent program produces failure, then all such executions will produce failure. Both of these improvements improve the execution run-time without compromising the program semantics. What is more, sometimes the *correctness* of GP 2 programs is tied to their confluent behaviour.

1.2 Research Hypothesis and Contributions

The hypothesis of this thesis is the following:

The language GP 2 can be effectively equipped with a confluence analysis system, facilitating proofs about confluence of many interesting graph programs

To support this hypothesis, such an analysis system must satisfy some criteria. In particular, it should be:

- *sound*: every confluence result must be valid with respect to the semantics of the language;
- *realistic*: in that it does not require impractical assumptions or restrictions on programs;
- *automatable*: all components of the system must be fully specified or implemented

These criteria will provide the basis for evaluating the work to be presented in the following chapters. To this end, the major contributions of this thesis are as follows.

Independence and Conflicts. We begin by introducing the notions of independence and conflict for rule schemata, central to the study of confluence. We lift the notions of independence and conflict to rule schemata, and prove the Local Church-Rosser Theorem which establishes that independent derivations are commutative and thus lead to the same result regardless of application order. This line of work is important not only because it is a paving stone for defining critical pairs, but also because without proving the commutativity of independent derivations, the confluence analysis based on critical pairs would be unsound.

Critical Pairs. Next, we develop critical pair analysis for sets of rule schemata by introducing symbolic critical pairs, which are pairs of derivations at the level of schemata, i.e. labelled with expressions, that are minimal and in conflict. We define our critical pairs at the level of graphs labelled with expressions to avoid an infinite number of such pairs. Then we give an algorithm for their construction, and showed the set of critical pairs is complete and finite (under suitable but *realistic* restrictions). The construction algorithm is based on computing graph overlaps, and unifying overlapped labels.

Unification Algorithm for List Expressions. We present our rule-based unification algorithm for solving systems of label equations. We show that the unification algorithm terminates, is *sound* and also complete meaning that every unifier of the input system of equations is an instance of some unifier in the computed set of solutions. Completeness of the algorithm is necessary for proving that our critical pairs are complete.

Confluence Algorithm for graph programs. We continue by introducing our notion of strong joinability of critical pairs, based on rewriting of graphs labelled with expressions. This is necessary as graphs in critical pairs are, in general, labelled with expressions rather than concrete values. Then, we establish the Local Confluence Theorem for GP 2 schema rewriting, which in effect allows for the specification of a *sound* confluence analysis *algorithm* based on critical pairs. We discuss the practical aspects of the algorithm by looking at two refinements that reduce its search space and improve its accuracy.

1.3 Thesis Structure

The content of this thesis is structured as follows.

Chapter 2 presents the fundamental theory of graph transformation, specifically in the double-pushout approach, the framework used as basis for the language GP 2. Then, it examines related research into confluence, first in its basic form concerning unlabelled graphs, and later how extends over some of the mentioned graph transformation features.

Chapter 3 provides the notions of *conflict* and *independence*, and proves the Local Church-Rosser Theorem for the case of GP 2. The result states that independent transformations can be interchanged and always lead to the same result.

Chapter 4 introduces the notion of *symbolic critical pairs* for GP 2, which are conflicting pairs of transformations in minimal context, gives an algorithm for their construction, and proves they have certain properties, namely that they represent all possible conflicts (*completeness*) and that there are only finitely many such critical pairs (*finiteness*).

Chapter 5 is concerned with how labels of critical pairs are computed. It gives a unification algorithm that solves equations between GP list expressions by providing a set of substitutions. These substitutions are used to construct the set of critical pairs. Important properties such as soundness, termination and completeness are shown.

Chapter 6 introduces our approach to using critical pairs in the context of a confluence checker. We explain our notion of symbolic rewriting, which allows us to rewrite the graphs found in critical pairs, in order to obtain joinability. We prove the Local Confluence Theorem which asserts the local confluence of a set of rule schemata given all their critical pairs are strongly joinable. Furthermore, we present our confluence analysis algorithm, based on joinability analysis of critical pairs, and study two refinements that improve its efficiency and accuracy.

Chapter 7 presents several case studies for the purposes of applying the developed techniques to existing graph programs.

Appendix A lists some definitions and properties of the class of partially labelled graphs and their morphisms in order to support some proofs in Chapter 4 and Chapter 6

Appendix B contains the full specification of the SELECT algorithm that is used in Chapter 5

Appendix C contains the proof of some lemmata that are either too lengthy or too technical to be contained in the main text.

1.4 Publication History

The results of this thesis have been published as the following papers. The author of this thesis was the primary contributor in all cases.

[HP15] Ivaylo Hristakiev and Detlef Plump. A unification algorithm for GP 2. In *Graph Computation Models (GCM 2014), Revised Selected Papers*, volume 71 of Electronic Communications of the EASST, 2015.

- Introduced the unification algorithm for left-linear schemata, on which Chapter 5 is based. The proof of completeness of the algorithm was only given in the long version made available shortly after, and later distributed as [HP17b]

[HP16a] Ivaylo Hristakiev and Detlef Plump. Attributed graph transformation via rule schemata: Church-Rosser theorem. In *Revised Selected Papers of Software Technologies: Applications and Foundations (STAF 2016) Collocated Workshops*, volume 9946 of *Lecture Notes in Computer Science*, Springer, 2016, pages 145–160

- Introduced our approach to independence and conflicts of rule schemata, basis for Chapter 3. The paper treats label algebras as parametric in line with the approach of attributed graph transformation. The main technical result is the lifting of the Church-Rosser Theorem from rule instances to rule schemata, in this more general setting.

[HP16b] Ivaylo Hristakiev and Detlef Plump. Towards critical pair analysis for the graph programming language GP 2, 2017. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques (WADT 2016), Revised Selected Papers*, volume 10644 of *Lecture Notes in Computer Science*, pages 153–169, Springer, 2017

- Presents our approach to critical pairs for GP 2 by giving a construction algorithm and also proving the important properties of completeness and finiteness of the set of critical pairs. Much of Chapter 4 is based on this.

[HP17a] Ivaylo Hristakiev and Detlef Plump. Checking graph programs for confluence. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations — STAF 2017 Collocated Workshops, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 92–108, Springer, 2017

- Extends critical pairs to consider joinability by means of symbolic rewriting, and presents our confluence analysis algorithm, on which Chapter 6 is largely based. Also shows the Shorest Distances case study, which is part of Chapter 7. The long version contains the full case study discussion.

Chapter 2

Graph Transformation, Graph Programming and Confluence

This chapter has three primary aims: first, to review and explain the intuition behind graph transformation, by only focusing on the essentials needed for the thesis; second, to review the graph programming language GP 2 that we are setting to analyse, discussing both its syntax and semantics, and illustrating its use with some example programs; third, to present the essence of confluence analysis as described in existing literature together with some variations.

2.1 Fundamentals of Graph Transformation

Graph transformation in its basic form is concerned with how to change one graph into another. These changes are called *transformations* (or *derivations*) and are guided by *rules*. Rules specify changes to a small substructure of the graph such as adding an edge between two nodes or removing a loop edge. Sometimes graph transformation is referred to as *rule-based manipulation* for this reason.

There exist several approaches to how exactly the rules operate. The algebraic approach, on which this chapter is based, uses pushout constructions from category theory which model the gluing of graphs. The two variants, the single-pushout (SPO) and the double-pushout (DPO) approaches, handle some details differently [EHK$^+$97]. For a comprehensive review of the different approaches in graph transformation, we direct the interested reader to [Roz97], and, containing a more recent state-of-the-art exposition, the handbook [EEGH15].

2.1.1 Graphs and Morphisms

We begin by defining graphs, and how graphs relate to each other by using structure-preserving mappings called morphisms. In our setting, edges are directed, nodes and edges have labels (unless specified), and parallel edges can exist. In the case when the labels play no role we will omit them, formally represented by a $\Box$ label which is not drawn. Unlabelled nodes are required later for technical reasons, when rules will be allowed to relabel nodes or edges.

Definition 2.1 (Label alphabet)**.** A *label alphabet* $\mathcal{L}$ is a set of labels used for annotating nodes and edges with labels.

Definition 2.2 (Partially labelled graph)**.** A *partially labelled graph* over a label alphabet $\mathcal{L}$ is a tuple $G = \langle V, E, s, t, l \rangle$ where

- V and E are finite sets of *nodes* and *edges*
- $s, t : E \to V$ are the *source* and *target* functions for edges
- $l : V + E \to \mathcal{L}$ is a partial node/edge labelling function

Given a node v, $l(v) = \perp$ expresses that $l(v)$ is undefined [1]. We say that node v is incident to edge e and vice versa if $s(e) = v$ or $t(e) = v$. Graph G is *totally labelled* if l_G is a total function. The classes of partially graphs and totally labelled graphs over a fixed label set $\mathcal{L}$ are written as $\mathcal{G}_\perp(\mathcal{L})$ and $\mathcal{G}(\mathcal{L})$. For convenience, we add indexes to graph elements to denote their graph, i.e. the labelling function of graph G is written as l_G.

Most of the graphs in this thesis will be totally labelled.

Example 1 (A graph). Consider the graphs in Figure 2.1 over the label alphabet $\mathcal{L} = \{\text{a,b}, \Box\}$. The graph on the left is the pictorial representation of

$$\begin{aligned} G_1 = \langle \{1,2\},\ \{e_1, e_2\},\ s : (e_1 \mapsto 1, e_2 \mapsto 1),\ t : (e_1 \mapsto 2, e_2 \mapsto 2), \\ l_G : (e_1 \mapsto \Box, e_2 \mapsto \Box) \rangle \end{aligned}$$

and the right one is a representation of

$$\begin{aligned} G_2 = \langle \{3,4\},\ \{e_3, e_4, e_5\},\ s : (e_3 \mapsto 3, e_4 \mapsto 3, e_5 \mapsto 3),\ t : (e_3 \mapsto 4, e_4 \mapsto 4, e_5 \mapsto 3), \\ l_G : (e_3 \mapsto \Box, e_4 \mapsto \Box, e_5 \mapsto \Box) \rangle \end{aligned}$$

Both graphs are totally labelled.

We follow the convention that nodes are drawn as circles and edges as arrows. Node labels are drawn inside the circle and node *identifiers* are drawn next to the nodes. Edge labels are drawn next to the arrows. Edge identifiers are not usually drawn, but when they are relevant we add them in a *id:label* syntax next to the arrows. If the label is $\Box$ then we omit from drawing it and just include the identifier in *italics*. Note that the box label is different from $\perp$ in that a node labelled with it has that distinct label whereas $\perp$ denotes the labelling function is undefined on that node. Later when we present graphs in the language of GP, we will make the same distinction between the empty list and $\perp$.

Graph morphisms allow us to relate graphs in a formal way. They are structure-preserving mappings for nodes and edges from one graph to another graph and allow us to properly define rule applications.

We start with the notion of a premorphism which is useful because it only talks about the *structure* of graphs rather than also their labels.

[1]We do not distinguish between nodes and edges in statements that hold for either.

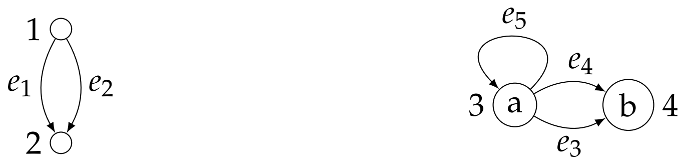


Figure 2.1: Pictorial representation of graphs.

Definition 2.3 (Premorphism)**.** A *premorphism* $g : G \to H$ between graphs G, H in $\mathcal{G}_\perp(\mathcal{L})$ consists of two functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ that preserve sources and targets : $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$.

Definition 2.4 (Graph morphism; undefinedness preservation)**.** A *graph morphism* is a premorphism g that also preserves labels of labelled nodes and edges, i.e. $(l_H \circ g_E)(x) = l_G(x)$ and $(l_H \circ g_V)(x) = l_G(x)$ for all $x \in G$ such that $l_G(x) \neq \perp$.

If a graph morphism also preserves the lack of labels of unlabelled nodes, then it is also *undefinedness-preserving*: $l_H \circ g = l_G$ for all nodes and edges.

Example 2. (A premorphism and a graph morphism). From Example 1, we can find several ways to map G_1 to G_2:

$$f_1 = \langle 1 \mapsto 3, 2 \mapsto 4, \{e_1 \mapsto e_3, e_2 \mapsto e_4\}\rangle$$

$$f_2 = \langle 1 \mapsto 3, 2 \mapsto 4, \{e_1 \mapsto e_4, e_2 \mapsto e_3\}\rangle$$

$$f_3 = \langle 1 \mapsto 3, 2 \mapsto 3, \{e_1 \mapsto e_5, e_2 \mapsto e_5\}\rangle$$

We have that f_1 and f_2 are injective graph morphisms and f_3 is non-injective. All three preserve node and edge labels because l_{G_1} is undefined for all nodes ($l_{G_1}(1) =\perp$ and $l_{G_1}(2) =\perp$) and edge labels are always $\square$. The mapping $h : G_2 \to G_1 = \langle 3 \mapsto 1, 4 \mapsto 2, \{e_3 \mapsto e_1, e_4 \mapsto e_2\}\rangle$ is not a proper graph morphism because both nodes 3 and 4 end up unlabelled in G_1.

Graph morphisms can be composed: given morphisms $f : G \to H$ and $g : H \to T$ the composition $g \circ f$ is the morphism $\langle g_V \circ f_V,\ g_E \circ f_E\rangle$ where $\circ$ denotes standard function composition.

A graph morphism is *injective* (*surjective*) if g_V and g_E are both injective (surjective). It is an *isomorphism*, denoted by $G \cong H$, if it is injective, surjective, and preserves undefinedness [2]. A graph morphism $g : G \to H$ is an inclusion if $g(x) = x$ for all items of G. Note inclusions need not preserve undefinedness, and that inclusions are premorphisms.

[2] It is clear why this is the case: if labels are ignored, then unlabelled graphs are isomorphic, meaning each item of G has a unique corresponding item in H; when labels are considered, for each item in G, its corresponding item in H has the same label using the undefinedness-preserving property.

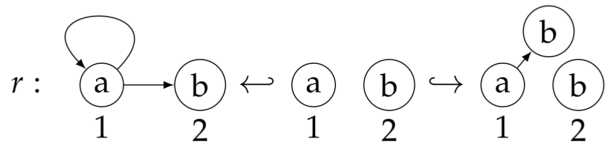


Figure 2.2: A rule.

2.1.2 Rules, Matches and Direct Derivations

A graph transformation rule allows us to change the graph to which the rule is applied. In the setting of the *double-pushout(DPO)* approach, rule applications are *local* in the sense that all changes to a given graph are as prescribed by the rule.

Definition 2.5 (Graph Transformation Rule)**.** A *graph transformation rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ over $\mathcal{L}$ (or *rule* for short) consists of two inclusions $K \rightarrow L$ and $K \rightarrow R$ such that L, K, R are graphs in $\mathcal{G}(\mathcal{L})$.

The left graph L can be thought as the pattern that needs to be matched, and the right graph R is what the pattern is replaced with. The interface K is used to define how the right graph glues into the context of the rule application. We will later allow K to be partially labelled, which facilitates relabelling of nodes/edges.

Example 3 (Example Rule). Figure 2.2 shows an example rule. It represents the replacement of a graph of two connected nodes with a loop on the first by a graph that is computed from the left graph by deleting both edges, and creating a new node labelled with 'b' that is connected to node 1. Note that nodes 1 and 2 are kept the same.

Sometimes it will be easier to write rules in the shorthand $\langle L \Rightarrow R \rangle$ instead of $\langle L \leftarrow K \rightarrow R \rangle$. In that case we assume that K consists of all and only nodes that have the same labels as their images in L and R.

Informally, an application of a rule r to a graph will remove the items in $L - K$, preserve K, and add the items in $R - K$. Such an application (in the DPO approach) is possible only when the match for the rule's left-hand graph L satisfies the following condition.

Definition 2.6 (Dangling condition)**.** Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule and G a graph in $\mathcal{G}(\mathcal{L})$. A graph morphism $g : L \rightarrow G$ satisfies the *dangling condition* if no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$

The morphism g is called the *match* of r in G. The dangling condition guarantees that the graph D obtained by removing nodes in $L - K$ is a proper graph, i.e. has no edges without source or target.

Definition 2.7 (Direct Derivation)**.** Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule, G a graph in $\mathcal{G}(\mathcal{L})$, and $g : L \rightarrow G$ an injective graph morphism satisfying the dangling condition. The direct derivation of G to H using r, denoted by $G \Rightarrow_{r,g} H$ or $G \Rightarrow_r H$, is a *direct derivation* if $H \cong H'$ where H' is constructed as follows:

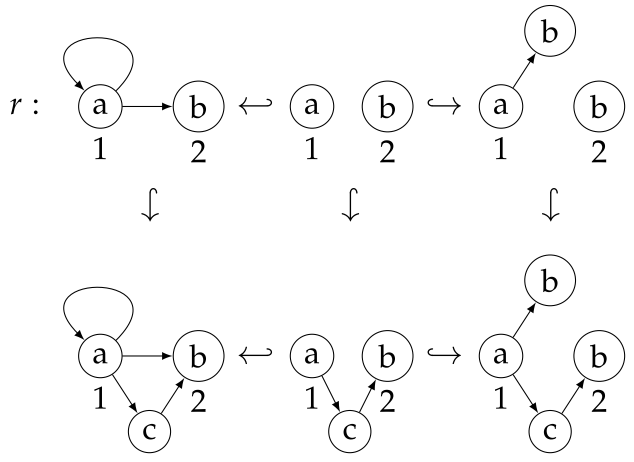


Figure 2.3: A direct derivation.

1. Remove all nodes and edges in $g(L) - g(K)$ from G, obtaining graph D
2. Add disjointly to D all nodes and edges from $R - K$, keeping their labels, obtaining graph H'. For each edge e in $R - K$, define source like this:

$$s_{H'}(e) = \begin{cases} s_R(e) & \text{if } s_R(e) \in V_R - V_K \\ g_V(s_R(e)) & \text{otherwise} \end{cases}$$

3. Define target analogously as source.
4. For labels, the graph items of H' that originate from G keep the same labels as in G. For those graph items that originate from $R - K$, their labels are as in R.

If $\mathcal{R}$ is a set of rules, then $G \Rightarrow_{\mathcal{R}} H$ means that there is some $r \in \mathcal{R}$ such that $G \Rightarrow_r H$. A sequence of direct derivations

$$G \Rightarrow_{\mathcal{R}} H_1 \Rightarrow_{\mathcal{R}} H_2 \Rightarrow_{\mathcal{R}} \ldots \Rightarrow_{\mathcal{R}} H_n$$

can be written as $G \overset{*}{\Rightarrow}_{\mathcal{R}} H_n$ (or $G \overset{*}{\Rightarrow} H_n$ if $\mathcal{R}$ is obvious from the context), expressing that H is derived from G in 0 or more rule applications. A direct derivation of length 0 is the isomorphism relation $G \cong H$.

2.1.3 Pushouts and Relabelling

The above definition of rule application (Definition 2.7) is enough for operational purposes, but makes it difficult to reason about graph transformation at an abstract level. In the DPO approach, the definition can be restructured using the notion of a *pushout* from category theory. This allows us to reason about properties of graph transformation in a more abstract setting.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & B \arrow[d] \\
C \arrow[r] & D
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & B \arrow[d] \arrow[ddr, bend left] & \\
C \arrow[r] \arrow[drr, bend right] & D \arrow[dr] \arrow[r, phantom, "="] \arrow[d, phantom, "="] & \\
& & D'
\end{tikzcd}$$

Figure 2.3: Commutative square and the Universal property of pushouts.

Definition 2.8 (Pushout)**.** A *pushout* over graphs ($B \leftarrow A \rightarrow C$) is a graph D in Figure 2.3 together with morphisms $B \rightarrow D$ and $C \rightarrow D$ such that

- *Commutativity*: the morphisms commute, i.e. $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$
- *Universal property*: for every pair of morphisms $B \rightarrow D'$ and $C \rightarrow D'$ such that the outer diagram commutes ($A \rightarrow B \rightarrow D' = A \rightarrow C \rightarrow D'$), there exists a *unique* morphism $D \rightarrow D'$ such that the triangles commute $B \rightarrow D' = B \rightarrow D \rightarrow D'$ and $C \rightarrow D' = C \rightarrow D \rightarrow D'$.

A pushout is the formal way of *gluing* two objects B and C along a common object A. The graph D is the disjoint union of B and C except on elements that are common with A, which are merged. This is sometimes denoted as $D = B +_A C$. Note we do not consider pushouts over non-label-preserving pre-morphisms.

Pushouts have several important properties. First, every node in D has a pre-image in either B or C (if in both, then it is also in A). Second, if $A \rightarrow C$ is injective (surjective), then $B \rightarrow D$ is also injective (surjective). Third, D is unique up to isomorphism — if another graph D' was a pushout of ($B \leftarrow A \rightarrow C$), then by the uniqueness property there would be morphisms from D to D' and from D' to D, which define an isomorphism $D \cong D'$ by the equations $D \rightarrow D' \rightarrow D = id_D$ and $D' \rightarrow D \rightarrow D' = id_{D'}$ (shown using only the pushout properties of the definition).

A pushout complement is the last definition we need in order to formally re-define direct derivations. It is similar to that of pushouts:

Definition 2.9 (Pushout complement)**.** A *pushout complement* of graphs $(A \rightarrow B \rightarrow D)$ is a graph C together with morphisms $A \rightarrow B$ and $B \rightarrow D$ such that $(C \rightarrow D \leftarrow B)$ is a pushout of $(B \leftarrow A \rightarrow B)$

Specific constructions of pushouts and pushout complements in the category of (totally labelled) graphs can be found in [EEPT06].

A direct derivation can be viewed as a pushout complement construction followed by a pushout construction. It can be shown that this is equivalent to Definition 2.7 (see [EPS73, EEPT06]), so we can redefine it:

$$\begin{tikzcd}
L \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
G \arrow[u, phantom, "(1)", shift left=8ex] & D \arrow[l] \arrow[r] \arrow[u, phantom, "(2)", shift left=8ex] & H
\end{tikzcd}$$

Figure 2.4: Direct derivation in the double-pushout (DPO) approach.

Definition 2.10 (Direct Derivation)**.** Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule, G a graph in $\mathcal{G}(\mathcal{L})$, and $g : L \rightarrow G$ an injective graph morphism satisfying the dangling condition. The transformation of G to H using r is given by the double-pushout diagram in Figure 2.4 with pushouts (1) and (2) .

Even though the DPO approach allows us to further reason about graph transformation at an abstract level by using results about (de)composition of pushouts and pushout complements, it has the shortcoming of not handling relabelling very well. For example, if we wanted to relabel an edge, we can do so by deleting it and recreating it with the desired label. However, to relabel a node, we have to delete it and then recreate it. This is only possible if the node is isolated due to the dangling condition.

In [HP02], the authors solve this problem by allowing rule interfaces to be partially labelled but requiring that pushouts (1) and (2) are also *pullbacks*. The paper also keeps *uniqueness* of direct derivations (up to isomorphism). Furthermore, the approach relaxes the requirement that all graphs are totally labelled, but places further restrictions on such unlabelled nodes/edges. As we shall see later, in GP 2 the graphs L and R are totally labelled and K is unlabelled on the nodes that we wish to relabel.

Definition 2.11 (Pullback)**.** A *pullback* over graphs $(C \rightarrow D \leftarrow B)$ is a graph A (as in Figure 2.5) together with morphisms $A \rightarrow B$ and $A \rightarrow C$ such that

- *Commutativity*: the morphisms commute, i.e. $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$
- *Universal property*: for every pair of morphisms $A' \rightarrow B$ and $A' \rightarrow C$ such that the outer diagram commutes ($A' \rightarrow B \rightarrow D = A' \rightarrow C \rightarrow D$), there exists a *unique* morphism $A' \rightarrow A$ such that the triangles commute $A' \rightarrow B = A' \rightarrow A \rightarrow B$ and $A' \rightarrow C = A' \rightarrow A \rightarrow C$.

Pullbacks are the dual construction of pushouts in category theory. Similarly, they represent an intersection of two objects over a common object. The results for pullbacks are similar to those of pushouts, e.g. injective/surjective morphisms are closed under pullbacks, and that pullbacks are unique up to isomorphism. Note we do not consider pullbacks over non-label-preserving pre-morphisms.

Definition 2.12 (Natural Pushout)**.** A natural pushout is a pushout that is also a pullback.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & B \arrow[d] \\
C \arrow[r] & D
\end{tikzcd}
\qquad
\begin{tikzcd}
A' \arrow[dr] \arrow[drr, bend left] \arrow[ddr, bend right] \arrow[rr, phantom, "=" near end] \arrow[dd, phantom, "=" near end] & & \\
& A \arrow[r] \arrow[d] \arrow[dr, phantom, "="] & C \arrow[d] \\
& B \arrow[r] & D
\end{tikzcd}$$

Figure 2.5: A commutative square and the Universal property of pullbacks.

What is special about the class of partially labelled graphs is that pushouts need not always exist, and not all pushouts along injective morphisms are natural.

Definition 2.13 (Direct Derivation with relabelling)**.** Let $r = \langle L \leftarrow K \rightarrow R \rangle$ be a rule, $L, R \in \mathcal{G}(\mathcal{L}), K \in \mathcal{G}(\mathcal{L}_\perp)$, G a graph in $\mathcal{G}(\mathcal{L})$, and $g : L \rightarrow G$ an injective graph morphism satisfying the dangling condition. A direct derivation with relabelling is a direct derivation in the sense of Definition 2.7 and Figure 2.4, but with the extra step:

4. For each unlabelled node v in K, $l_H(g_V(v))$ becomes $l_R(v)$

The change to Figure 2.4 is that pushouts (1) and (2) have to be natural. This requirement guarantees that double pushout diagrams are unique. For an example direct derivation with relabelling, see Figure 2.6.

The paper [HP02] gives a characterization of natural pushouts in the sense of the following lemma.

Lemma 1. *The pushout in Definition 2.8 is natural if and only if for all elements $x \in A$, $l_A(x) = \perp$ implies $l_B(b(x)) = \perp$ or $l_C(c(x)) = \perp$.*

This restriction means that for every node we want to relabel, first we omit a label in the interface K, and second, we omit a label in the gluing graph D. As a result, the graph H becomes unique up to isomorphism, and is totally labelled iff G is totally labelled.

2.2 Graph programming with GP

Graph transformation has been extended in several ways. One of them has been to allow the explicit control of rule applications. The paper [HP01] showed that when equipped with sequential composition and as-long-as-possible iteration constructs, graph transformation becomes computationally complete in the sense of Turing. This led to the design of GP, a minimal complete language for graph transformation that allows high-level problem solving on graphs [Bak16, Plu12, Plu09], on which we concentrate for the rest of this thesis.

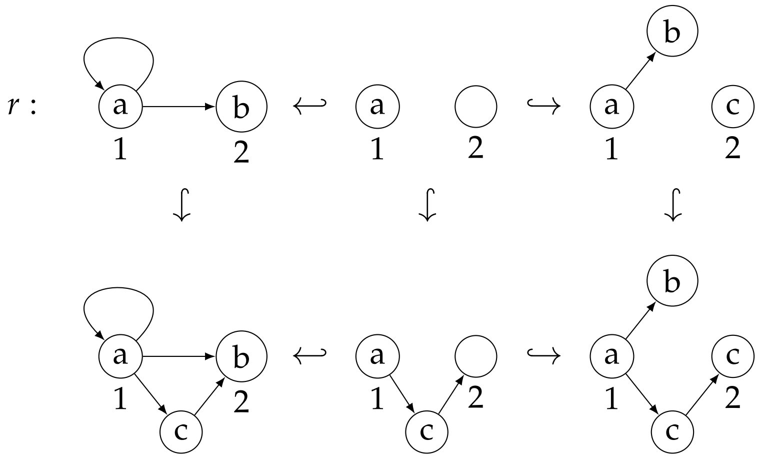


Figure 2.6: A direct derivation with relabelling.

A graph program comprises of two components: (1) a set of rules, based on the (DPO) rules with relabelling (but more general); and (2) program text expressing how the rules are combined into programs, e.g. using sequential composition, as-long-as-possible iteration and non-deterministic choice. The rules are actually *rule schemata* that play the role of rule blueprints that are instantiated at run-time with concrete data. The language is equipped with with an operational semantics, intended to facilitate formal reasoning and verification.

Remark 1 (GP and GP 2). The language's second version (called GP 2) was formally described in [Plu12], and later implemented in [Bak16]. We will interchange the terms "graph programs" or "GP" or "GP 2", but it can be understood that we refer to the most recent description of the language in [Bak16]. The language's first version was originally described in [Plu09].

The rest of the section is organized as follows — first we define GP's building blocks (conditional) rule schemata and what graphs they operate on. Then, we give the abstract syntax of graph programs and give example GP programs. Finally, we briefly outline the operational semantics of the core and derived commands.

2.2.1 Graphs and Rule Schemata

In GP, there is a difference between the labels in program rules and labels of graphs on which the rules operate. The first are labelled over integer, string and list *expressions* and may contain variables of those types. The latter are labelled over integers, strings and lists of integers and strings and are provided as input to programs. Both kinds of labels can also be *marked*.

Let $\mathbb{Z}$ be the set of integers, Char be a finite set of characters and $\mathbb{M}$ be the finite set of marks $\{\texttt{red}, \texttt{green}, \texttt{blue}, \texttt{grey}, \texttt{dashed}\}$.

Definition 2.14 (Host graph label alphabet)**.** The label alphabet $\mathcal{L} = (\mathbb{Z} \cup \text{Char}^*)^* \times$

$$\begin{tikzcd}[row sep=small, column sep=small]
 & \texttt{list} & \\
 & \texttt{atom} \arrow[u, phantom, sloped, "\subseteq"] & \\
\texttt{int} \arrow[ur, phantom, sloped, "\subseteq"] & & \texttt{string} \arrow[ul, phantom, sloped, "\subseteq"] \\
 & & \texttt{char} \arrow[u, phantom, sloped, "\subseteq"]
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=small, column sep=small]
 & (\mathbb{Z} \cup \mathrm{Char}^*)^* & \\
 & \mathbb{Z} \cup \mathrm{Char}^* \arrow[u, phantom, sloped, "\subseteq"] & \\
\mathbb{Z} \arrow[ur, phantom, sloped, "\subseteq"] & & \mathrm{Char}^* \arrow[ul, phantom, sloped, "\subseteq"] \\
 & & \mathrm{Char} \arrow[u, phantom, sloped, "\subseteq"]
\end{tikzcd}$$

Figure 2.7: Subtype hierarchy for GP expressions.

RSLabel	::=	List [Mark]
List	::=	`empty` | Atom | LVar | List ':' List
Mark	::=	`red` | `green` | `blue` | `dashed` | `any`
Atom	::=	Integer | String | AVar
String	::=	'"' {Char} '"' | SVar | String '.' String
Integer	::=	Digit {Digit} | IVar | '-' Integer
		| Integer ArithOp Integer | (`indeg` | `outdeg`) '(' Node ')'
		| `length` '(' (LVar|AVar|SVar) ')'
ArithOp	::=	'+' | '-' | '*' | '/'
String	::=	'"' {Character} '"' | SVar | CVar | String '.' String

Figure 2.8: Abstract syntax of rule schema labels.

$\mathbb{M}$ is called the label alphabet for *host graphs*, where each label consists of a list of *atoms* (integers or strings) and an optional mark ($\in \mathcal{M}$).

Sometimes the mark component is omitted, e.g. writing $l_G(v) = 25$ means that label of node v in graph G has list component 25 with an unspecified mark component.

Each list expression is built up from constant symbols in $\mathbb{Z}$ and Char, variables, and function symbols. Variables are *typed*, each drawn from a disjoint set of variables: IVar, SVar, AVar and LVar respectively for denoting integer, string, atom and list variables. Their union is denoted as Var. The subtype hierarchy for expressions is presented in Figure 2.7.

The grammar presented in Figure 2.8 defines how expressions are built using basic components. Node is the set of node identifiers occurring in a GP rule schema, which must be the same for the left and the right graph. The intended meaning of symbols is '+' for addition, '.' for string concatenation and ':' for list concatenation. The length operator returns the length of its variable argument according to the variable's assignment obtained during graph matching.

Definition 2.15 (Symbolic graphs)**.** Let RS denote the label alphabet of all expressions generated by the syntactic class RSLabel of the grammar in Figure 2.8, then $\mathcal{G}(\mathrm{RS})$ is the set of all *symbolic graphs* over RS and $\mathcal{G}_\perp(\mathrm{RS})$ is the set of all partially labelled symbolic graphs over RS.

Rule schemata are rules in the sense of Definition 2.5, but with L and R being labelled with expressions and K consisting of unlabelled nodes only. They represent possibly infinite sets of rules over $\mathcal{G}(\mathcal{L})$, obtained by assigning values to variables and evaluating expressions. Because instantiation takes place at execution time and is determined by graph matching, it is required that expressions in the left graph of a rule schema must have a simple shape.

Definition 2.16 (Simple list)**.** An expression $e \in$ List is *simple* if

1. e contains no arithmetic operators
2. e contains at most one occurrence of a list variable
3. each occurrence of a string expression in e contains at most one occurrence of a string variable

For example, if $\mathtt{x}, \mathtt{y} \in \mathtt{LVar}$, $\mathtt{a} \in \mathtt{AVar}$, $\mathtt{s}, \mathtt{t} \in \mathtt{SVar}$ are variables, then the expressions a:x and "no".s:y:t are simple whereas x:y and $s.t$ are not simple.

Definition 2.17 (Rule schema)**.** A *rule schema* $\langle L \leftarrow K \rightarrow R\rangle$ consists of two inclusions $K \rightarrow L$ and $K \rightarrow R$ such that L, R are graphs in $\mathcal{G}(\mathrm{RS})$ and K is a graph in $\mathcal{G}(\mathrm{RS}_\perp)$ consisting of unlabelled nodes only. All list expressions in L must be simple and that all variables occurring in R must also occur in L.

The requirements on labels and variables ensure that, for a given match, applying a rule produces a unique graph (up to isomorphism). See [PS04, Section 5] for a formal proof.

Rule schemata are instantiated by evaluating their labels according to some assignment $\alpha : \mathtt{Var} \rightarrow \mathcal{L}$ [3].

Definition 2.18 (Assignment)**.** An assignment is a family of mappings

$$\alpha = (\alpha_X)_{X \in \{I,S,A,L\}}$$

where

- $\alpha_I : \mathtt{IVar} \rightarrow \mathbb{Z}$
- $\alpha_S : \mathtt{SVar} \rightarrow \mathrm{Char}^*$
- $\alpha_A : \mathtt{AVar} \rightarrow \mathbb{Z} \cup \mathrm{Char}^*$
- $\alpha_L : \mathtt{LVar} \rightarrow (\mathbb{Z} \cup \mathrm{Char}^*)^*$

We call $r^{g,\alpha} = \langle L^{g,\alpha} \leftarrow K \rightarrow R^{g,\alpha}\rangle$ the *instance* of r with respect to g and α, where $L^{g,\alpha}$ and $R^{g,\alpha}$ are obtained from L and R by replacing each label l with $l^{g,\alpha}$ which involves evaluation of expressions. See Figure 2.11 for a complete example.

We can restrict the application of a rule schema by a textual application condition. This extension is inspired by application conditions in graph transformation

[3] $\mathtt{Var} = \mathtt{LVar} \cup \mathtt{AVar} \cup \mathtt{SVar} \cup \mathtt{IVar}$

Condition	::=	Type ′(′ List ′)′ | List (′=′ | ′!=′) List
		| Integer IntRel Integer
		| `edge` ′(′ Node ′,′ Node [′,′ Label] ′)′
		| `not` Condition | Condition (`and`|`or`) Condition
Type	::=	`int` | `string` | `atom`
IntRel	::=	′<′ | ′≤′ | ′>′ | ′≥′

Figure 2.9: Abstract syntax of rule schema conditions.

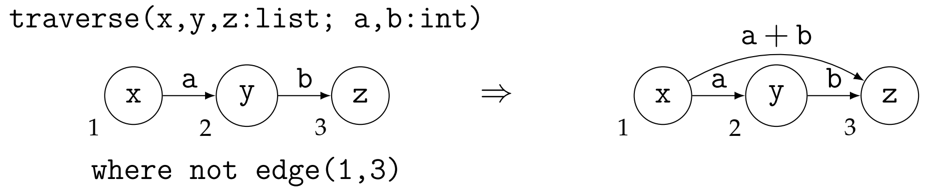


Figure 2.10: A conditional rule schema.

that allow for the specification of a structure that must or must not exist for the rule to be applied. Application conditions have been subject to extensive research as their existence has implications for static properties of graph transformation like confluence ([EGH$^+$12, GLEO12, EHL$^+$10]).

In GP, rule schemata can be equipped with conditions as specified by the grammar in Figure 2.9. Here Node denotes a set of node identifiers occurring in the rule schema.

The Type predicate expresses that a list expression must evaluate to a certain concrete type. The List comparison allows for comparing the values of two list expressions, and similarly for integer comparison. The `edge` predicate gives a structural restriction on existence of an edge between two nodes in the matched graph (the third optional parameter specifies the label of that edge). This is more useful when negated, forbidding the rule application in contexts where edges (with a particular label) exist outside of the match.

Definition 2.19 (Conditional rule schema)**.** A *conditional rule schema* is a pair $\langle r, ac \rangle$ where r is a rule schema and ac is a condition generated by the Condition grammar in Figure 2.9 such that all variables occuring in ac also occur in the left-hand graph L of r.

Figure 2.10 shows a conditional rule schema `traverse` that links two indirectly connected nodes with a direct edge. The rule schema identifier is written above the two graphs along with any variable declarations. Multiple variables can have the same type, separated by a semi-colon (:). The condition is written below the two graphs following the keyword `where`. This condition ensures that in the context of the match there isn't already an edge connecting the nodes 1 and 2 and hence the rule cannot be matched forever (i.e. not leading to non-termination).

From now on we will refer to conditional rule schemata as rule schemata and be explicit in the cases when the condition matters.

Definition 2.20 (Rule schema application)**.** The application of a rule schema $r = \langle L \leftarrow K \rightarrow R, ac\rangle$ to a host graph $G \in \mathcal{G}(\mathcal{L})$, denoted by $G \Rightarrow_{r,g} H$ (or just $G \Rightarrow_r H$) informally is done as follows:

1. Match L with a subgraph of G, ignoring labels, by means of a premorphism $g : L \rightarrow G$
2. Check whether there is an assignment α of variables to values such that after evaluating the simple expressions in L, g is label-preserving
3. Check whether the rule schema condition ac evaluates to `true` under the given premorphism g and assignment α
4. Apply the rule $r^{g,\alpha}$, obtained from r by evaluating all expressions in L and R, to G accroding to Definition 2.7 to obtain graph H .

We write $G \Rightarrow_{r^{g,\alpha},g} H$ to denote the application of $r^{g,\alpha}$ to G with match g. Given a set $\mathcal{R}$ of rule schemata, we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some rule schema r in $\mathcal{R}$.

Example 4 (Example direct derivation). Figure 2.11 shows the rule schema `traverse` being applied to a labelled graph G. The application starts by finding an injective premorphism $g : L \rightarrow G$ — there are two of them:

$$g_1 = \langle 1 \mapsto 5, 2 \mapsto 6, 3 \mapsto 7\rangle$$

$$g_2 = \langle 1 \mapsto 5, 2 \mapsto 8, 3 \mapsto 7\rangle$$

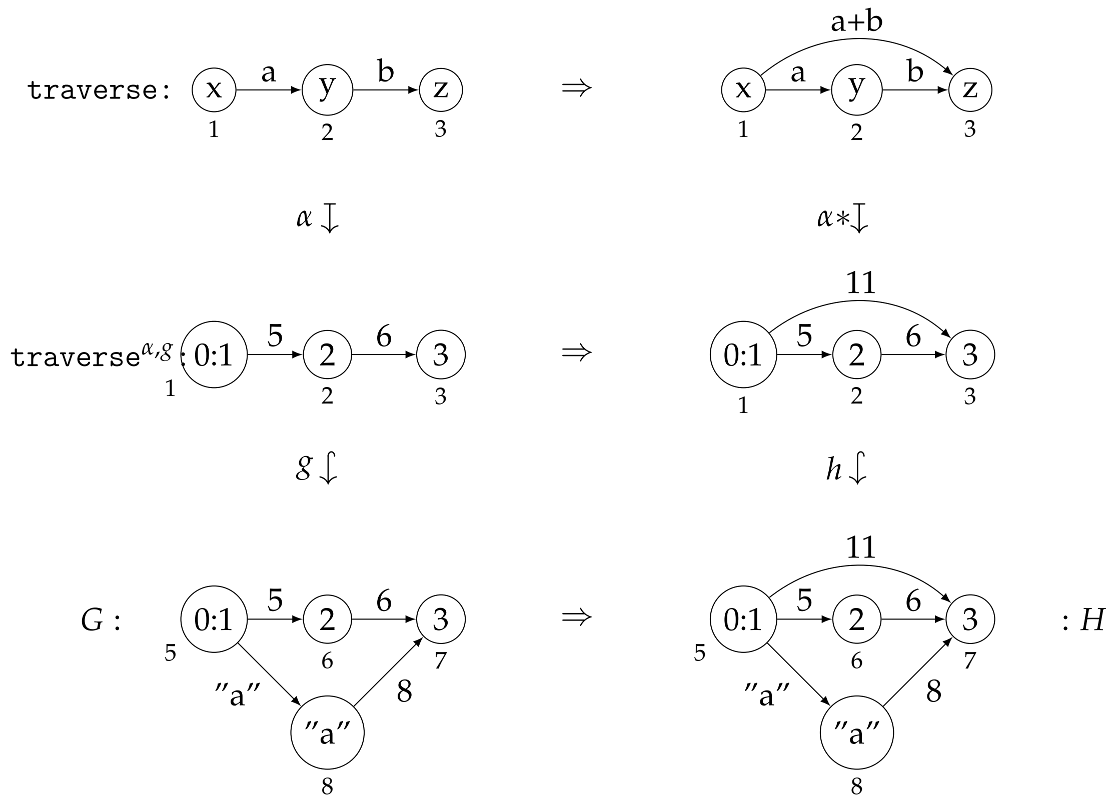
If g_2 is chosen as a match, then the next step fails because there is no assignment of the integer variable a to the string "a". Therefore, the only match is g_1 and the corresponding assignment is

$$\alpha = \langle \mathrm{x} \mapsto 0{:}1, \mathrm{y} \mapsto 2, \mathrm{z} \mapsto 3, \mathrm{a} \mapsto 5, \mathrm{b} \mapsto 6\rangle$$

The rule schema condition `"not edge(1,3)"` is checked next — it evaluates to `true` because $g(1)$ and $g(3)$ in G (respectively nodes 5 and 7) do `not` have an edge between them.

The derivation process can proceed by *instantiating* `traverse` with α and g which is shown in the middle of the figure. Note that $R^{\alpha,g}$ contains *evaluated* labels instead of expressions, i.e. the value 11 instead of 5+6 for node 3.

Once `traverse`$^{\alpha,g}$ has been obtained, the process can proceed as a DPO direct derivation with relabelling (Definition 2.13). The resulting graph H is shown in the bottom right. None of the nodes change their labels, hence the rule instance interface is totally labelled and essentially an unlabelled copy of L (not explicitly shown).



$g_V = \langle 1 \mapsto 5, 2 \mapsto 6, 3 \mapsto 7 \rangle$
$\alpha = \langle \mathrm{x} \mapsto 0{:}1, \mathrm{y} \mapsto 2, \mathrm{z} \mapsto 3, \mathrm{a} \mapsto 5, \mathrm{b} \mapsto 6 \rangle$

Figure 2.11: A rule schema instance and application.

2.2.2 Graph Programs

A graph program is a set of rule schemata declarations together with some program text. As mentioned at the start of this section, the control constructs of sequential composition and as-long-as-possible iteration are enough for computational completeness. However, GP offers a few extra operatros for usability clarity and convenience. The syntax of programs is in Figure 2.12.

A program is a list of declarations, which are either a Rule Schema declaration, a macro declaration, or a `main` program declaration. The `main` program signifies the program text to be executed and contains several Command clauses (generalized in Command Sequences). A command sequence is the sequential composition of commands:

- RuleSetCall — either a single rule schema or a non-deterministic choice of rule schemata. RuleID represents the set of rule identifiers, unique for each rule schema.
- `if-then-else` — a conditional branching construct; executes the `then` clause

```
Prog         ::=  Decl {Decl}
Decl         ::=  RuleDecl | MacroDecl | MainDecl
MainDecl     ::=  main '=' ComSq
MacroDecl    ::=  MacroID '=' ComSq
ComSq        ::=  Com {';' Com}
Com          ::=  RuleSetCall | MacroCall
                  | if ComSq then ComSq [else ComSq]
                  | try ComSq [then ComSq] [else ComSq]
                  | ComSq '!'
                  | '('ComSq ')'
                  | ComSq 'or' ComSq
                  | skip | fail
RuleSetCall  ::=  RuleID | '{' [ RuleID {',' RuleID } ] '}'
MacroCall    ::=  MacroID
```

Figure 2.12: Abstract syntax of graph programs.

or the `else` clause depending on the result of the condition *execution* on a *copy* of the input graph.

- `try-then-else` — another conditional branching construct; the condition program is executed on the input graph rather than on a copy
- `ComSq!` — the looping construct; the body is executed as many times as possible
- `ComSq or ComSq` — non-deterministic choice of two programs
- `skip` — do nothing; equivalent to executing the empty rule which is always applicable
- `fail` — manually trigger program termination. Equivalent to executing the empty set of rule schemata $\{\}$

The `skip`,`fail` and `or` commands can be simulated by the others.

Example 5 (Transitive closure). Consider the example GP program in Figure 2.13 that computes the transitive closure of an integer labelled graph [Plu09, Example 1]. It uses the rule schema from Figure 2.10.

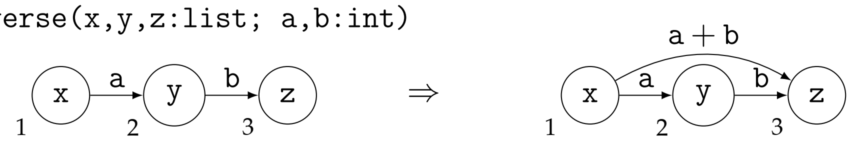
The rule schema is applied as long as possible. At each step, two nodes that are indirectly connected are joined with a direct edge. The program can be shown to terminate and to be correct, see [Plu16, Section 3].

2.2.3 Example Programs

In this section, we present some example programs in order to give some intuition into graph programming. These programs will be useful later in Chapter 7

```
main = traverse!


traverse(x,y,z:list; a,b:int)
```



```
where not edge(1,3)
```

Figure 2.13: The program `traverse`, taken from [Plu09, Example 1] and allowing node labels to be lists.

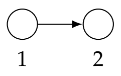
when we will study whether they are confluent or not by applying the techniques developed in this thesis.

The first program recognizes a specific class of graphs called *series-parallel graphs;* the second program computes the shortest path in a graph from a given source node to all other nodes; the third manipulates the labels of nodes to compute the 2-colouring of a graph (if it exists); the last program computes any graph colouring (rather than 2-colouring).

2.2.3.1 Recognition of Series-Parallel graphs

This program is concerned with the recognition of a class of graphs called 'series-parallel' graphs (SPGs). These graphs have been introduced as models of electrical networks [Duf65], and have been of interest in computational complexity theory since many graph problems, some of which NP-complete, are solvable in linear time for these graphs (e.g. maximum matching, maximum independent set, Hamiltonian completion).

SPG definition. Series-parallel graphs can be defined inductively by two simple composition operations (Figure 2.14):

- The graph $G =$  is series-parallel; define v_1 to be the *source* of G, and v_2 to be the *sink* of G
- The *parallel* composition of two SPGs G and H is created from the disjoint union $G + H$ by merging the sources of G and H and merging the sinks of G and H
- The *series* composition of two SPGs G and H is created from the disjoint union $G + H$ by merging the sink of G with the source of H

As the class of graphs is defined inductively, checking whether a graph is SPG is difficult using the definition directly. There exists an alternative approach where

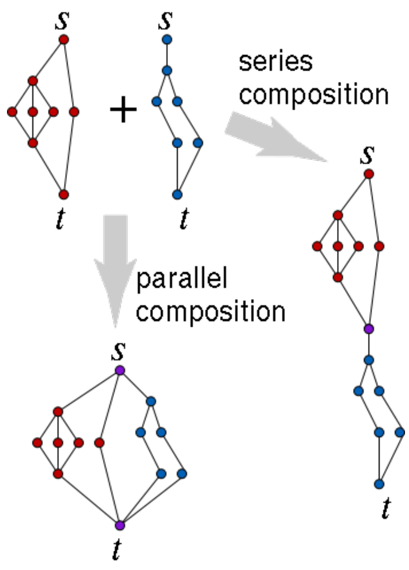


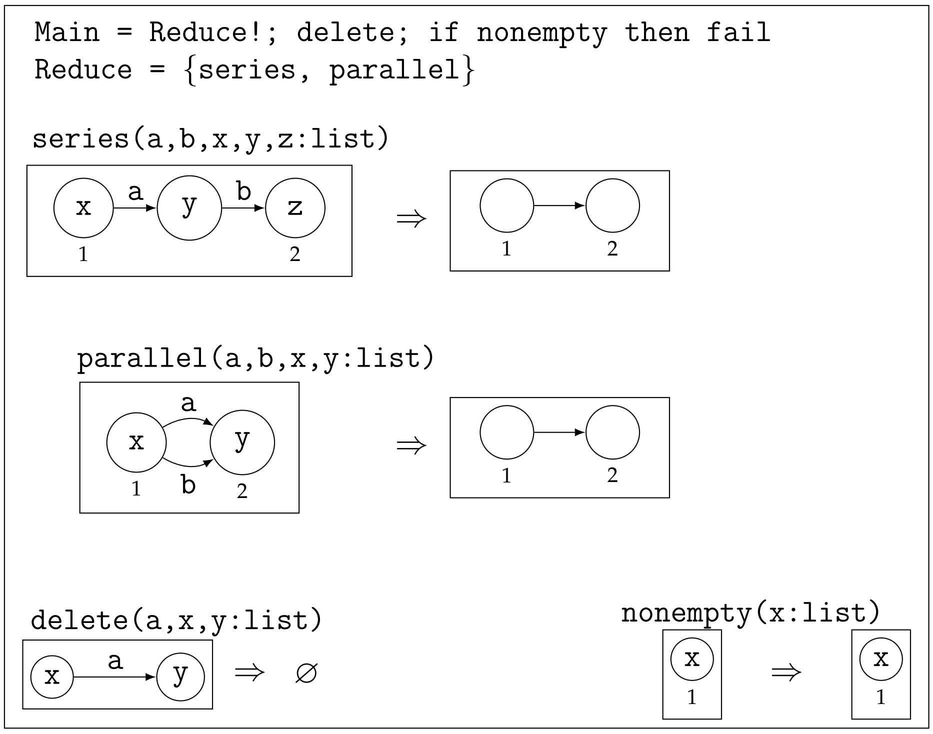
Figure 2.14: Series-parallel graphs and their compositions. *Source: Wikipedia*

one applies a set of rules to a graph and reduce the original problem to an isomorphism check.

Recognition of SPGs as a program. The recognition of SPGs is done by means of *graph reduction*: for an input graph G, apply a set of reduction rules as long as possible, obtaining graph H. Check for H having a specific property X to decide whether the original graph G has the property Y. In the case of series-parallel checking, the reduction algorithm is instantiated as follows:

- For an input graph G, apply the reduction rules `Reduce = {series, parallel}` as long as possible, obtaining a graph H
- Check whether the graph H is isomorphic to the graph $\bigcirc\!\!\longrightarrow\!\!\bigcirc$
  - If yes, output "G is a series-parallel graph"
  - Otherwise, output "G is not a series-parallel graph"

That algorithm is implemented by the program in Figure 2.15, which expects an input host graph G that is assumed to not contain node or edge marks. (Node marks are irrelevant to the problem of series-parallel recognition.) The program first applies the set of reduction rules as long as possible, resulting in some graph H. To determine whether H has the correct shape, the program first deletes the predefined shape (see above), then checks whether the result is the empty graph. If either the deletion or the non-empty check fails then the program fails. In this context, termination of the program with a proper graph means the input graph G is series-parallel, and failure means G is *not* series-parallel.



```
Main = Reduce!; delete; if nonempty then fail
Reduce = {series, parallel}
```

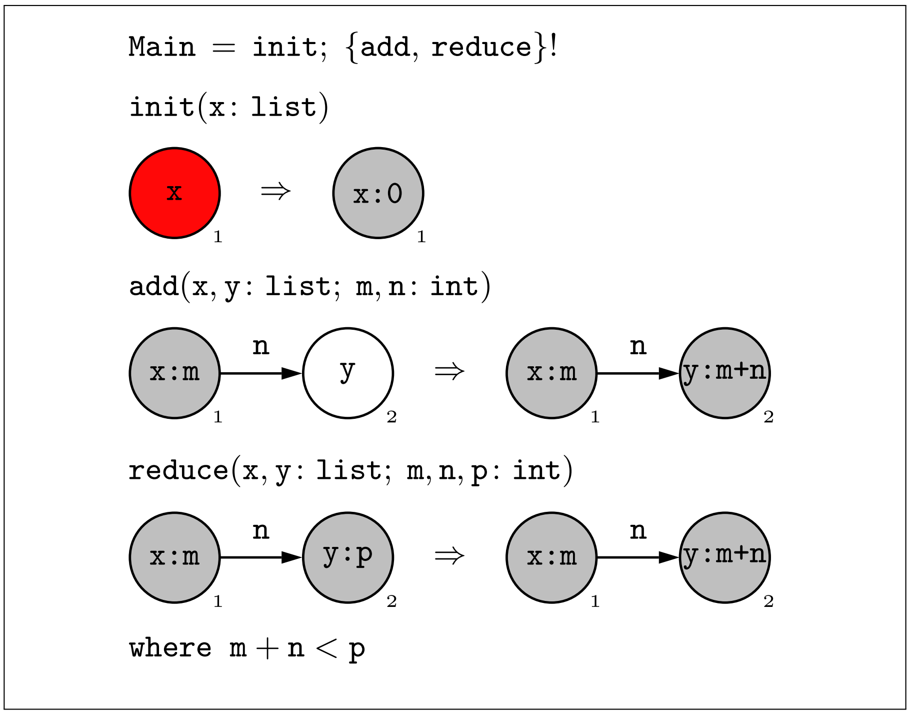
Figure 2.15: A program that checks whether a graph is series-parallel or not [Plu16, Section 6].

A GP version of the program was first shown in [Plu09, Example 6], and here we have presented it using GP 2's syntax, the only difference being the semantics of success / failure as opposed to wrapping the program in an `if-then-else` that executes pre-specified sub-programs depending on the SPG check. A more recent version is of [Plu16, Section 6] from which Figure 2.15 originates.

However, if the non-deterministic reduction results in a graph different than $\bigcirc\!\!\rightarrow\!\!\bigcirc$, there might be some other execution sequence resulting in that graph. Therefore, the *correctness* of the algorithm depends on the *confluence* of the reduction rules. Here correctness means the program succeeds for (unmarked) SPGs and fails otherwise.

2.2.3.2 Computing Shortest Distances

The shortest distances problem is about calculating the minimal distances between a given node (the *source* node) and all other connected nodes in a graph, where the distance between two nodes is defined as the sum of edge weights on any path connecting the two nodes. The Bellman–Ford algorithm [BG02] is an algorithm that solves that problem. It is based on *relaxation* where the current distance to a node is gradually replaced by more accurate values until eventually reaching the optimal / minimal solution. An assumption made is that there is no *negative cycle* (a cycle whose edge weights sum to a negative value) that is connected with the



```
Main = init; {add, reduce}!
init(x: list)
add(x, y: list; m, n: int)
reduce(x, y: list; m, n, p: int)
where m + n < p
```

Figure 2.16: Shortest Distances program.

source, in which case there is no shortest path. Furthermore, the input is assumed to be unmarked except for the single red node as the program itself uses marks to track information; this does not restrict the set of possible inputs as a marked graph can have its mark translated as part of a pre-processing step.

A GP 2 program that implements the Bellman–Ford algorithm is shown in Figure 2.16. Distances from the source node are recorded by concatenating the distance value to each node's label. Nodes *marks* are used: the source node is red, visited nodes are gray, and unvisited nodes are unmarked. Given an input graph G with a unique source node and no negative cycle, the program initializes the distance of the source node to 0. The `add` rule explores the unvisited neighbours of any visited nodes, assigns them a tentative distance and marks them as visited to avoid non-termination. The `reduce` rule finds occurrences of visited nodes whose current distance is higher than alternative distances, i.e. only when the application condition ($\mathtt{m} + \mathtt{n} < \mathtt{p}$) is satisfied by the schema instantiation. The program terminates when neither `add` or `reduce` rules can be further applied. Note that for simplicity the program only operates on input graphs that are (initially) unmarked.

A GP version of the program was first shown in [Plu09], and here we have presented it using GP 2's syntax, the only difference being the assumption of a unique red node as opposed to a node with a specific unique label.

However, since rule application is non-deterministic, different graphs may result from program execution. The algorithm is correct only if the loop {`add,reduce`}!

is confluent. In the absence of a full program verification, a programmer may want to check that this loop indeed returns unique results. Here correctness means the program, when given an input satisfying the described assumptions, terminates with a graph such that each node's label is augmented with the shortest distance to the source node.

2.2.3.3 Computing a 2-colouring

This program is concerned with computing a 2-colouring of an input graph if such a colouring is possible. To colour the nodes of a graph means to assign the values 0 or 1 to each node, representing node colours, such that no two adjacent nodes have the same integer/colour. (A graph is 2-colourable iff its underlying loop-free, undirected graph does not contain a cycle of odd length). The nature of the colouring problem depends on the number of colours but not on how they are represented. In this problem, we assume colours are represented by concatenating the colour value to the node's pre-existing label. If the input graph has existing node labels, then those should be preserved, and the assigned colours should be appended to the existing labels. An assumption of this algorithm is that the input graph is *connected* and is unmarked. A more complicated algorithm is needed to deal with disconnected components.

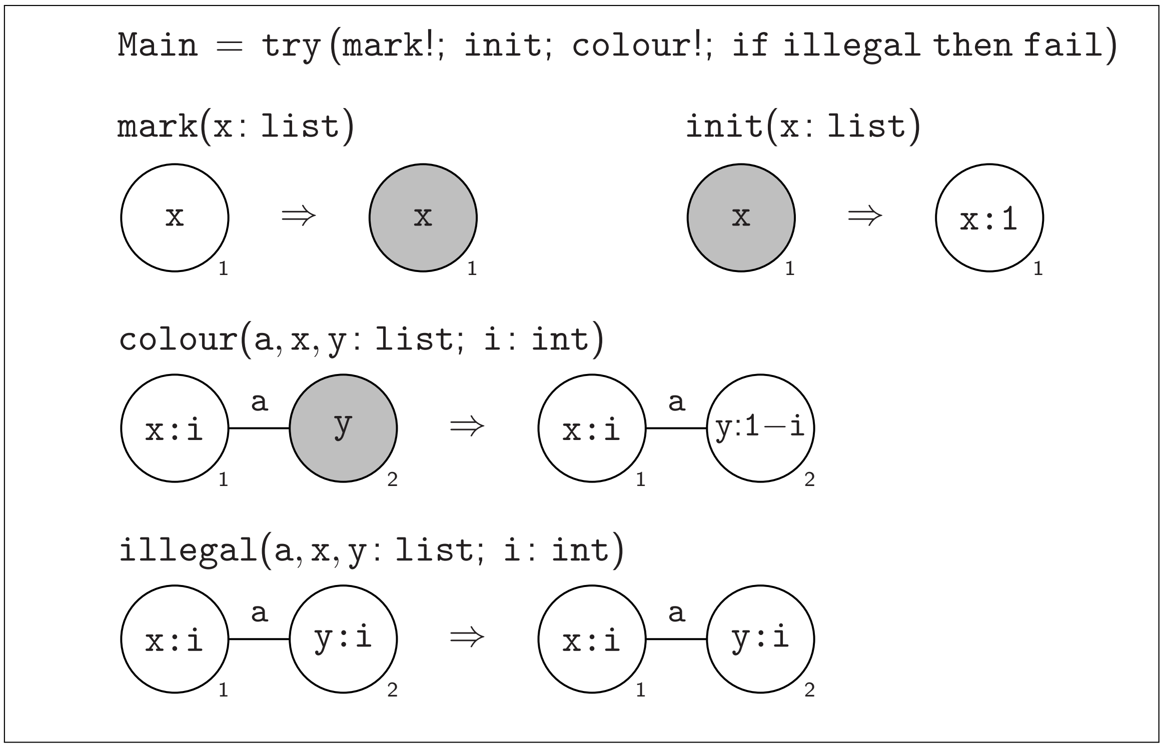
A program that computes such a 2-colouring (Figure 2.17) works as follows. Then, apply the `colour` rule as long as possible. The `colour` rule pick an unmarked node and explore its marked neighbours, assigning them alternative colours (0 or 1) and unmarking them. This process is guaranteed to terminate. If the resulting graph has an invalid colouring, signal failure. Otherwise, output the graph as it was initially. The latter is achieved by wrapping the computation in a `try` block.

The program uses so-called bidirectional edges which are essentially syntactic sugar for matching edges in either direction. To avoid complications due to this feature, we can consider `colour` to be a rule set where the given edge is directed from node 1 to node 2 (rule `colour1`) or from node 2 to node 1 (rule `colour2`).

A GP version of this program was first shown in [Plu09], and here we have presented it using GP 2's syntax, the only differences being (1) the use of explicit failure as opposed to undoing the illegal colouring, and (2) the changed input assumption that there must be no marks as opposed to nodes having integer-only labels [4]. First, initialize all nodes as unvisited by marking them gray and nondeterministically choose one marked (grey) node to append the initial colour 1 and unmark it. A different version of the solution where colours are represented as GP 2 node marks (red and blue) is given in [Bak16, Section 6.3].

The application of `colour` is non-deterministic — give an input graph, there may be several different assignments of colours for its nodes. The 'illegal colouring' check relies on the correctness (and, more specifically, the confluence) of applying `colour` as long as possible — otherwise, there might be some other derivation sequence ending in a correctly coloured graph. Correctness of colouring means that, given a connected unmarked input graph, after colouring is done, all

[4]GP did not have marks.



Main = try (mark!; init; colour!; if illegal then fail)
mark(x: list)
init(x: list)
colour(a, x, y: list; i: int)
illegal(a, x, y: list; i: int)
x
x:1
x:i
y
y:1−i
y:i
a
⇒
1
2

Figure 2.17: 2-colouring program [Plu16, Section 4].

nodes have been coloured and that no two adjacent nodes have the same colour. Therefore, it is interesting to study the confluence of `colour` in the absence of full program verification.

2.2.3.4 Computing a Graph Colouring

A *vertex colouring* (or *colouring* for short) is an assignment of colours to nodes such that each non-loop edge has end points with distinct colours. Computing such a colouring is always possible for loop-free finite graphs by just assigning unique colours to nodes, and computing a *minimal* colouring is NP-complete. A program that computes such a (non-minimal) colouring is as follows (copied from [Plu16, Section 4]), and is very similar to the 2-colouring program given in the previous section:

$$\texttt{Main} = \texttt{mark!;\ init!;\ inc!}$$

where the `mark` and `init` rules are as previously defined, and the `inc` (given in Figure 2.18) finds adjacent nodes assigned the same colour and changes one of the nodes' colour assignment. No loops or marks are allowed.

The first part of the program (`mark!; init!`) is deterministic: it will append an initial colour (1) to every node. The second part of the program is the loop `inc!` which is terminating but highly non-deterministic. What is interesting about this

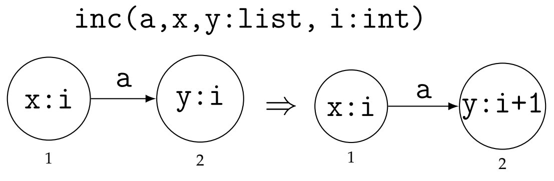


Figure 2.18: The `inc` rule.

program is that we can compute a counter example to confluence at the symbolic (expression) level.

2.2.4 Operational Semantics

The language GP has formal semantics ([Plu12, Bak16]) in the style of Plotkin's structural operational semantics [Plo04]. Each programming construct has several related inference rules that induce a transition relation $\to$ on *configurations*. A configuration represents the current state of computation and is either a graph (a proper result) or a command sequence and a graph (unfinished computation) or a special element `fail` representing program failure.

Definition 2.21 (Transition relation)**.** The *small-step transition relation*

$$\to \;\subseteq\; \big(\mathrm{ComSq} \times \mathcal{G}(\mathcal{L})\big) \times \big((\mathrm{ComSq} \times \mathcal{G}(\mathcal{L})) \cup \mathcal{G}(\mathcal{L}) \cup \{\mathrm{fail}\}\big)$$

on configurations defines the single-step computation on graphs. The transitive and reflexive-transitive closures are written as $\to^+$ and $\to^*$ respectively.

A configuration λ is said to be *terminal* if there is no configuration δ such that $\lambda \to \delta$.

The inference rules for GP's core and derived commands ("derived" because they can be defined by core commands) are given in Figure 2.19 and Figure 2.20 respectively. An inference rule has a *premise* and a *conclusion*. The rules contain (implicitly) universally quantified meta-variables for command sequences and graphs — $\mathcal{R}$ stands for a call in RuleSetCall, C, P, P', Q stand for command sequences in ComSq and G, H stand for graphs in $\mathcal{G}(\mathcal{L})$.

Definition 2.22 (Semantic inference rules for core commands)**.** Figure 2.19 defines the *inference rules for the core GP commands*. The notation $G \not\Rightarrow_{\mathcal{R}} H$ expresses that for a host graph $G \in \mathcal{G}$, there is no graph H such that $G \Rightarrow_{\mathcal{R}} H$.

Consider the rule [call$_1$] — intuitively it reads "for all sets of conditional rule schemata $\mathcal{R}$ and host graphs G, H, $G \Rightarrow_{\mathcal{R}} H$ implies $\langle \mathcal{R}, G\rangle \to H$".

Note the definition of the `if-then-else` — the alternative Q is executed if the conditional program C fails. This is the concept of *negation as failure* that comes from logic programming. Also, the programs P and Q are executed on the *original graph* G, which allows for the hiding of "destructive" tests. The alternative

$$[\mathrm{call}_1]\ \frac{G \Rightarrow_{\mathcal{R}} H}{\langle \mathcal{R}, G\rangle \to H} \qquad [\mathrm{call}_2]\ \frac{G \not\Rightarrow_{\mathcal{R}}}{\langle \mathcal{R}, G\rangle \to \texttt{fail}}$$

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

$$[\mathrm{seq}_3]\ \frac{\langle P, G\rangle \to \texttt{fail}}{\langle P; Q, G\rangle \to \texttt{fail}}$$

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

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

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

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

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

Figure 2.19: Inference rules for the core commands.

`try-then-else` executes the consequent P on the graph resulting from executing C.

Failing executions of `if-then-else`, `try-then-else` and `as-long-as-possible` are notable from the point of view of nondeterminism — if the guard C (or loop body P respectively) may result in both a proper result graph and a fail, then the choice of which alternative to take (whether to continue looping or to pass back control to the outer program) is nondeterministic. To illustrate this subtlety, consider a looping sequence `(r1; r2)!` (similar to [Bak16, p. 45]). If the first rule succeeds and the second fails, i.e. $G \overset{r_1}{\Rightarrow} H$ and $H \not\Rightarrow^{r_2}$, then the loop body fails and the program returns the original input G. However, if the result of the first rule H is such a graph that the second rule succeeds, then the loop body succeeds and the execution continues with trying the loop body again.

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

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

$$[\text{if}_3]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to \langle P, G\rangle} \qquad [\text{if}_4]\ \frac{\langle C, G\rangle \to^+ \texttt{fail}}{\langle \texttt{if}\ C\ \texttt{then}\ P, G\rangle \to G}$$

$$[\text{try}_3]\ \frac{\langle C, G\rangle \to^+ H}{\langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \to \langle P, H\rangle} \qquad [\text{try}_4]\ \frac{\langle C, G\rangle \to^+ \texttt{fail}}{\langle \texttt{try}\ C\ \texttt{then}\ P, G\rangle \to G}$$

Figure 2.20: Inference rules for derived commands.

Figure 2.20 defines the *inference rules* for the derived commands of GP. Similarly as above, the `or` command is nondeterministic in the sense that the evaluation may choose which of the two programs to execute.

The meaning of the GP commands can be defined by the semantic function $[\![\_]\!]$ which assigns to each program P the function $[\![P]\!]$ that maps an input graph G to the set of all possible results of executing P on G. The application of $[\![P]\!]$ to G is denoted by $[\![P]\!]G$. This set may also contain the special symbols `fail` and $\perp$. The first indicates that the program can end in failure, and the second means that the program may diverge (not terminate) or "get stuck".

Definition 2.23 (Divergence)**.** A program P *diverges from* graph G if there is an infinite sequence

$$\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$$

Definition 2.24 (Getting stuck)**.** A program P *gets stuck* from graph G if there is terminal configuration $\langle Q, H\rangle$ such that $\langle P, Q\rangle \to^* \langle Q, H\rangle$

A program can get stuck in two situations:

- it contains a `if-then-else` (or `try-then-else`) such that the condition C can diverge on some graph G and can neither produce a proper result graph nor fail
- it contains a loop $P!$ such that the body P has the previous property.

The evaluation of such programs gets stuck because none of the inference rules for the respective constructs are applicable.

The intention of the semantic function is to assign to a program sequence a set of possible outputs for any input. This set of results depends on the program and the input graph.

Definition 2.25 (Semantic function)**.** The semantic function $[\![\_]\!] : \text{ComSq} \to \big(\mathcal{G}(\mathcal{L}) \to 2^{\mathcal{G}(\mathcal{L}) \cup \{\texttt{fail}, \bot\}}\big)$ is defined by

$$\begin{aligned} [\![P]\!]G = \; & \{\, \mathrm{X} \in (\mathcal{G}(\mathcal{L}) \cup \{\texttt{fail}\}) \mid \langle P, G\rangle \to^{+} X\} \cup \\ & \{\, \bot \mid P \text{ can diverge or get stuck from } G\} \end{aligned}$$

Definition 2.26 (Semantic equivalence)**.** Two programs P and Q are *semantically equivalent*, denoted by $P \equiv Q$, if $[\![P]\!] = [\![Q]\!]$.

For example, the following equivalences can be proven:

- `skip` $\equiv$ `null`, where `null` is the empty rule schema $\emptyset \Rightarrow \emptyset$ that is always applicable
- `fail` $\equiv \{\}$, where $\{\}$ is the empty set of rule schemata
- `if` C `then` $P \equiv$ `if` C `then` P `else null`, for all programs C and P
- `try` C `then` $P \equiv$ `try` C `then` P `else null`, for all programs C and P
- `try` C `else` $P \equiv$ `try` C `then null else` P, for all programs C and P

2.3 Confluence

As already mentioned, the computational model of graph transformation is non-deterministic. However, due to the ways that rules interact, it may be the case that there exists exactly one result for each input graph.

A graph transformation system is a label alphabet $\mathcal{L}$ together with a finite set of rules $\mathcal{R}$ over $\mathcal{L}$. Given a graph G, transforming the graph under $\mathcal{R}$ means to apply the rules as long as possible, obtaining a result graph H. The graph H is called a *normal form* of G. If each graph has a unique normal form, then the *order* of transformations does not matter and $\mathcal{R}$ is called *confluent*.

The notion of confluence originally comes from (abstract) rewriting systems [KB83, Hue80]. A set of transformation rules $\mathcal{R}$ is confluent if for every pair of diverging transformations $H_1 \overset{*}{\Leftarrow} G$ and $G \overset{*}{\Rightarrow} H_2$, there is a common graph H such that $H_1 \overset{*}{\Rightarrow} H \overset{*}{\Leftarrow} H_2$. Confluence ensures there exists at most one normal form.

Confluence is an important property in several aspects. First, together with termination, it guarantees the global determinism (*functional behaviour*) of transformations. This is important when transformations are used to implement functions or translations from one domain into another. Second, on a more practical level, it alleviates the costs associated with non-determinism while keeping semantic equivalence of the implementation. On a similar note, in the presence of *failures*, if a confluent sequence of derivations fails, then any possible sequence from the same start graph will also fail, thus removing the need for retrying a failed computation.

Counterexamples to confluence, i.e. pairs of non-isomorphic results for a given input, can also be useful. An implementation may choose to first apply independent derivation steps as long as possible and then start creating decision points for backtracking, thus reducing memory footprint. Furthermore, the proof of non-confluence allows for the classic Knuth-Bendix completion procedure that adds extra rules until confluence of the expanded set of rules is established.

Confluence turns out to be a very useful property for several application areas of graph transformation. One such application is model transformation (see [RLP$^+$14] for a comprehensive comparison between existing model transformation systems), one of the criteria for its correctness is the existence of confluence analysis. The compared model transformation tools in [RLP$^+$14] all lack confluence analysis: only the tool Kermeta comes close, achieving confluence in principle but providing no tool support for a proof.

Other tools are also available: the tool AGG [RET11] implements confluence analysis in the setting of typed attributed graph transformation, while Henshin [ABJ$^+$10, SBG$^+$17] offers confluence support for a purpose-built model transformation language used for model refactoring, pattern introduction, and model evolution.

It is well known that confluence is an undecidable property [Plu93]. However, a sufficient criterion for confluence exists which is concerned with minimal conflicting situations called *critical pairs*. Critical pairs can be statically computed and are representative of all conflicts that may arise during computation. Critical pairs can be used to argue that a rewriting system is confluent or to provide a counter-example to its confluence.

The rest of the chapter is organized as follows — first we properly study what it means for two transformations to be in conflict; next, we present critical pairs and the basic algorithm for their computation; last, we present flavours of critical pairs that handle attribution, typing and application conditions.

2.3.1 Structural Confluence

We start by investigating the formal definitions of confluence and its flavour local confluence.

Definition 2.27 (Confluence)**.** A pair of derivations $G \overset{*}{\Rightarrow}_{\mathcal{R}} H_1$ and $G \overset{*}{\Rightarrow}_{\mathcal{R}} H_2$ is *confluent* if there exist derivations $H_1 \overset{*}{\Rightarrow}_{\mathcal{R}} G'$ and $H_2 \overset{*}{\Rightarrow}_{\mathcal{R}} G'$. A graph transformation system $\mathcal{R}$ (a set of rules) is *confluent* if for all graphs G, all pairs of derivations from G are confluent.

Informally, this means that two diverging derivations are confluent if they can be further rewritten to a common graph. See Figure 2.21.

Remark 2. For the rest of this section we will only consider *unlabelled* graphs (graphs in $\mathcal{G}(\mathcal{L}_\perp)$), i.e. will only be interested in the structural aspect of confluence. Later on we will discuss how labels/attributes affect confluence.

A similar version of confluence is called local confluence, which is concerned with the pair of derivations in the definition to *direct* derivations (derivations of length 1). According to a general result for rewriting systems, it is sufficient to consider local confluence, given that the system is terminating.

$$\begin{tikzcd}
 & G \arrow[dl, Rightarrow, "*"'] \arrow[dr, Rightarrow, "*"] & \\
H_1 \arrow[dr, Rightarrow, "*"'] & & H_2 \arrow[dl, Rightarrow, "*"] \\
 & G' &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & G \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
H_1 \arrow[dr, Rightarrow, "*"'] & & H_2 \arrow[dl, Rightarrow, "*"] \\
 & G' &
\end{tikzcd}$$

Figure 2.21: Confluence and Local Confluence.

Definition 2.28 (Local Confluence)**.** A pair of direct derivations $G \Rightarrow_{\mathcal{R}} H_1$ and $G \Rightarrow_{\mathcal{R}} H_2$ is *locally confluent* if there exist derivations and $H_1 \overset{*}{\Rightarrow}_{\mathcal{R}} G'$ and $H_2 \overset{*}{\Rightarrow}_{\mathcal{R}} G'$. A graph transformation system $\mathcal{R}$ is *locally confluent* if for all graphs G, all pairs of direct derivations from G are locally confluent.

The difference between confluence and local confluence is illustrated in Figure 2.21.

Local confluence is strictly weaker because it does *not* imply confluence, as shown in the following example.

Example 6 (Local confluence does not imply confluence). Consider the locally confluent system with four different direct transformations.

$$\begin{tikzcd}
A & B \arrow[l, Leftarrow] \arrow[r, Rightarrow, bend left] & C \arrow[l, Rightarrow, bend left] \arrow[r, Rightarrow] & D
\end{tikzcd}$$

The pairs $A \Leftarrow B \Rightarrow C$ and $B \Leftarrow C \Rightarrow D$ are locally confluent because they can be joined by $C \Rightarrow B \Rightarrow A$ and $B \Rightarrow C \Rightarrow D$ respectively.

However, we can see that A and D are two unique normal forms of B (cannot be transformed further) — the pair $A \Leftarrow B \overset{*}{\Rightarrow} D$ cannot be joined. Similarly, the pair $A \overset{*}{\Leftarrow} C \Rightarrow D$ cannot be joined. Therefore, the system is not confluent.

Definition 2.29 (Termination)**.** A system of rules $\mathcal{R}$ is *terminating* if there is no infinite sequence of transformations

$$G_0 \Rightarrow_{\mathcal{R}} G_1 \Rightarrow_{\mathcal{R}} G_2 \Rightarrow_{\mathcal{R}} \dots$$

Termination ensures the existence of at least one normal form. However, termination is also an undecidable property [Plu98]. Some termination criteria for graph transformation systems have been shown in [EEdL$^+$05].

Lemma 2 (Newman's Lemma)**.** *Every terminating and locally confluent graph transformation system is also confluent.*

The previous example is non-terminating because there exists an infinite sequence

$$B \Rightarrow C \Rightarrow B \Rightarrow \dots$$

As a consequence, the local confluence of the system does not imply confluence. However, one might suspect that the cycle between B and C is responsible which turns out to not be the case — even for acyclic relations, local confluence does not imply confluence [BN98, Section 2.7].

2.3.2 Independence and Conflicts

In order to reason about local confluence, we will distinguish two types of pairs of direct derivations — those that are *independent* and those that are dependent (in *conflict*). The reason for this is that two independent derivations are always interchangeable and can be applied in any order, whereas pairs that are dependent may be a source of non-determinism. In this section we only show *parallel* independence, and avoid talking about *sequential* independence which is its dual (e.g. see [EEPT06] for further clarification, and Definition 3.4).

Definition 2.30 (Independence)**.** Two direct transformations $G \Rightarrow_{r_1, g_1} H_1$ and $G \Rightarrow_{r_2, g_2} H_2$ are *independent* iff the second derivation can be applied to H_1 and vice versa, i.e. there are morphisms (matches) $L_1 \to D_2$ and $L_2 \to D_1$ such that $L_1 \to G = L_1 \to D_2 \to G$ and $L_2 \to G = L_2 \to D_1 \to G$

$$\begin{tikzcd}
R_1 \arrow[r, dash] \arrow[d, dash] & K_1 \arrow[r, dash] \arrow[d, dash] & L_1 \arrow[dr, hook, "g_1"'] \arrow[drrr, dashed, "h_{12}"] & & L_2 \arrow[r, dash] \arrow[dl, hook', "g_2"] \arrow[dlll, dashed, "h_{21}"'] & K_2 \arrow[r, dash] \arrow[d, dash] & R_2 \arrow[d, dash] \\
H_1 \arrow[r, dash] & D_1 \arrow[r, dash] & f_1 \arrow[r, dash] & G \arrow[r, dash] & f_2 \arrow[r, dash] & D_2 \arrow[r, dash] & H_2
\end{tikzcd}$$

A pair of transformations is in conflict if it is not independent. An equivalent definition [EEPT06, Fact 3.18] is to say the pair is independent iff all common items are preserved, given by the following set-theoretic condition:

$$g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$$

where for convenience we have assumed that $K_i \to L_i$ are inclusions (and hence the match g_i is defined over the subgraph $K_i \subseteq L_i$).

It has been shown that two independent transformations can be applied in any order. This is the classic Local Church-Rosser Theorem — rule r_2 can be applied to the result of the first transformation H_1 using the same match g_2 (and vice versa).

Theorem 2.1 (Local Church-Rosser Theorem [Roz97, EEPT06])**.** *Given two independent transformations $G \overset{r_1, g_1}{\Longrightarrow} H_1$ and $G \overset{r_2, g_2}{\Longrightarrow} H_2$, there is a graph G' together with transformations $H_1 \overset{r_2, g_2'}{\Longrightarrow} G'$ and $H_1 \overset{r_1, g_1'}{\Longrightarrow} G'$ such that the following diagram commutes.*

$$\begin{tikzcd}
& G \arrow[dl, Rightarrow, "{r_1, g_1}"'] \arrow[dr, Rightarrow, "{r_2, g_2}"] & \\
H_1 \arrow[dr, Rightarrow, "{r_2, g_2'}"'] & & H_2 \arrow[dl, Rightarrow, "{r_1, g_1'}"] \\
& G' &
\end{tikzcd}$$

Conflicts can be *characterized* depending on the setting — when dealing with "plain" (unlabelled) graphs and rules, conflicts are called *delete-use* because one of the rules deletes a node/edge which the other matches. In the presence of attribution/labels and application conditions, there can be more types of conflicts such as *forbid-produce* and *delete-require*.

2.3.3 Critical Pairs

Proving a transformation system is confluent may be a very difficult task just by looking at the definition — joinability must hold for *all* graphs G and *all* derivable graphs H_1 and H_2 from G. The presence of conflicts makes the task even more difficult as the Church-Rosser Theorem cannot be applied, and the resulting system of rules may not be confluent.

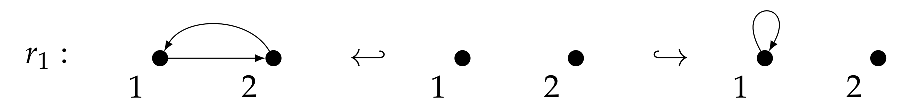
In order to check confluence in finite time, the potentially infinite set of conflicting pairs must be reduced to a finite set of representatives. This is the aim of constructing *critical pairs*, pairs of dependent transformations in minimal context.

Definition 2.31 (Critical pair)**.** A pair of direct derivations $T_1 \Leftarrow_{r_1,g_1} S \;{}_{r_2,g_2}\!\Rightarrow T_2$ is a *critical pair* if:

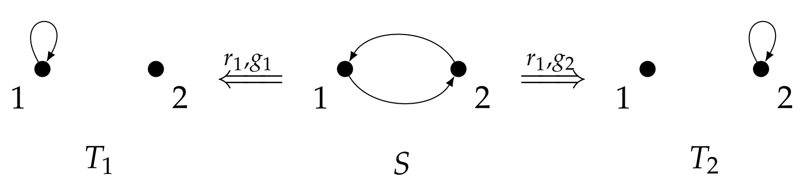
1. The graph S is an overlap of the left-hand sides of the given rules, i.e. the matches g_1 and g_2 are jointly surjective and that each item in S has a preimage in L_1 or L_2
$$S = g_1(L_2) \cup g_2(L_2)$$
2. The derivations are in conflict:
$$g_1(L_1) \cap g_2(L_2) \not\subseteq g_1(K_1) \cap g_2(K_2)$$
3. If $r_1 = r_2$, then $g_1 \neq g_2$ must hold

The minimality condition means that S does not contain unnecessary context. The dependency condition ensures that the derivations are not applicable to each others' result graphs H_1 or H_2, i.e. the Church-Rosser Theorem does not apply. The extra condition of different matches ensures that we are not dealing with the same derivation.

Example 7 (Example Critical Pair). Consider the rule r_1:



The following diagram represents a critical pair of the rule with itself:



with matches $g_1 = L \to S : \langle 1 \to 1, 2 \to 2\rangle$ and $g_2 = L \to S : \langle 1 \to 2, 2 \to 1\rangle$ respectively. (Here L is the left-hand graph of r_1.)

The (critical pair) graph S is minimal because it contains no extra context that is unused by the matches, i.e. g_1 and g_2 are jointly surjective. The pair of derivations is a conflict because both delete the proper edges of S and the rules cannot be applied to the results T_1 and T_2 at all. We also have that $g_1 \neq g_2$, which is needed since the derivations involve the same rule.

It can be shown that each pair of dependent transformations is an extension of a critical pair.

Lemma 3 (Completeness of critical pairs)**.** *For each pair of conflicting direct derivations $H_1 \Leftarrow_{r_1,g_1'} G \;_{r_2,g_2'}\!\Rightarrow H_2$, there is a critical pair $T_1 \Leftarrow_{r_1,g_1} S \;_{r_2,g_2}\!\Rightarrow T_2$ such that S is a subgraph of G.*

$$\begin{tikzcd}
T_1 \arrow[d] & S \arrow[l, Leftarrow, "r_1,g_1"'] \arrow[r, Rightarrow, "r_2,g_2"] \arrow[d, "m"] & T_2 \arrow[d] \\
H_1 & G \arrow[l, Leftarrow, "r_1,g_1'"'] \arrow[r, Rightarrow, "r_2,g_2'"] & H_2
\end{tikzcd}$$

Both of these squares are extension diagrams in the sense of [EEPT06].

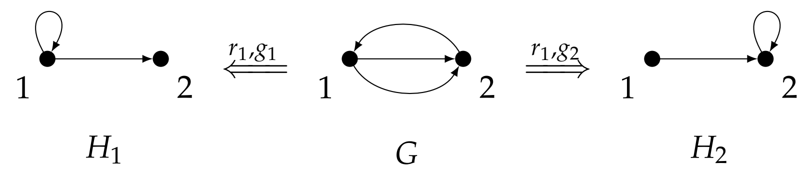
Apart from reasoning about conflicts, critical pairs have another important property — in rewrite systems where the rules consist of finite graphs (such as in graph transformation) the set of critical pairs is finite because the left-hand sides of two rules can be overlapped in only finitely many ways. This is in contrast to the number of pairs of conflicting derivations which is infinite in general.

If the set of critical pairs is empty, then the above lemma already implies local confluence of the graph transformation system because all pairs of derivations will be independent by the Church-Rosser Theorem. Otherwise, in order to show local confluence, it is sufficient to examine all critical pairs and determine if they are *strongly joinable*. Strong joinability means that the critical pair is

- joinable — for some graph S', there exist derivations $T_1 \overset{*}{\Rightarrow} S'$ and $T_2 \overset{*}{\Rightarrow} S'$
- strictness — the largest subgraph N of S that is preserved by *both* steps in the critical pair is preserved by the joining derivations $T_1 \overset{*}{\Rightarrow} S'$ and $T_2 \overset{*}{\Rightarrow} S'$.

Joinability without the strictness condition has been shown not to be enough for verifying local confluence (see [Plu93]) because the isomorphism between the two resulting graphs from the joining derivations may be destroyed when the critical pair is embedded into context.

In Example 7, the critical pair is joinable because $T_1 \cong T_2$ with isomorphism $iso = T_1 \to T_2 : \langle 1 \to 2, 2 \to 1 \rangle$. However, it is not strongly joinable because the isomorphism does not map nodes correctly: either of the nodes of G, when tracked through the joining derivations, end up in different nodes. It is easy to see a counter example to confluence by adding an extra edge, i.e. context:



Both H_1 and H_2 are non-isomorphic normal forms of G. Therefore, the rule r_1 does not represent a confluent graph transformation system.

Strong joinability allows us to state the so-called Critical Pair Lemma.

Theorem 2.2 (Local Confluence Theorem, Critical Pair Lemma)**.** *A graph transformation system is locally confluent if all its critical pairs are strongly joinable.*

2.3.4 Critical Pair Construction Algorithm

Having given an intuition of critical pairs and why they are useful for analysing sets of rules for confluence, we now proceed with showing practical ways of computing all critical pairs induced by a set of rules.

The basic algorithm for computing the set of all critical pairs can be derived from the definition [LE06]. First, we need to compute all jointly surjective matches for the two rules, i.e. suitable gluings of the left-hand sides. Second, we need to analyse if the transformations given by the two rules and computed matches are indeed parallel dependent. If that is the case, then that is indeed a critical pair.

```
input : Two rules $r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle$ and $r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle$
output: A set of critical pairs $CP$
1  $CP \leftarrow \varnothing$
2  Compute all jointly surjective morphisms ($g_1 : L_1 \rightarrow S$, $g_2 : L_2 \rightarrow S$) of $r_1$ and $r_2$
3  foreach $(g_1, g_2)$ do
4      if $r_1 = r_2$ and $g_1 = g_2$ then skip;
5      Compute $t_1 = (S \Rightarrow_{r_1,g_1} T_1)$ and $t_2 = (S \Rightarrow_{r_2,g_2} T_2)$
6      if $t_1$ and $t_2$ are not parallel independent then
7          $CP \leftarrow CP \cup \{(t_1, t_2)\}$
8      end
9  end
10 return $CP$
```

Algorithm 1: Basic algorithm for computing all critical pairs.

The main part of the algorithm is the computation of the jointly-surjective matches — this is computationally expensive since their number will grow very large if the left-hand sides of the two rules are large [Wel14, Section 7.2].

The number of overlaps (partitions) is bounded by the n-th Bell number

$$v(n) = B_n < \left(\frac{0.792n}{ln(n+1)} \right)^n \text{ , } n \in \mathbb{N}_+$$

where $v(n)$ is the total number of possible overlaps (injective and non-injective) of n elements (in our case $n = |L_1| + |L_2|$). Note that the algorithm can be easily modified to consider only injective matches.

Related Work. [LE06] present two possible optimizations to algorithm 1. First, if both rules r_1 and r_2 are non-deleting, then they will always be parallely independent. A rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is non-deleting if the morphism $K \rightarrow L$ is an isomorphism. There are different ways of checking this, and the authors prefer the construction of the *context* graph C of $K \to L$ (see [EEPT06] for further details).

The second optimization involves the situation between a deleting and a non-deleting rule. Instead of computing all possible overlaps and later filtering them, it is possible to compute only those ones which lead to a critical pair. This is achieved by requiring that an element that is deleted by one rule to be matched by the non-deleting rule. The authors define certain conditions under which the pair of rules are guaranteed to be in conflict. In practice, this involves constructing a pullback object of the context of the deleting rule and the left-hand side of the non-deleting rule (and hence reusing the computation of the first optimization).

In a further paper [LEO08], the authors define *essential* critical pairs, which are a subset of all critical pairs, and are complete in the sense that all critical pairs are strongly joinable if and only if all essential critical pairs are strongly joinable. However, the set of essential critical pairs is not significantly smaller than the set of all critical pairs.

2.3.5 Confluence Extensions

In this section, we present several extensions to confluence that handle different more complex approaches to graph transformation. The extensions we show handle attributes of nodes/edges, and also application conditions.

Attributes. Confluence in its basic form is *structural* in the sense that the source of non-determinism are delete-use conflicts, in the setting of unlabelled graphs (as described in the previous part). However, in many cases where graph transformation is used as a modelling technique, the domain is *attributed* (*labelled*) graphs that represent diagrams with textual, numerical, semantic annotations.

In [HKT02], the authors develop the theory of critical pairs to the setting of attribution. The paper introduces typed, vertex-attributed graph transformation systems [5] and proves the Local Confluence Theorem in that setting. The paper's running example considers a translation between simple UML state-charts into CSP in the context of automated verification using a CSP model checker, and this process needs to be functional (confluent and terminating).

In this setting, graph attributes are represented by means of special *data* nodes and linked to ordinary graph nodes/edges by attribution edges. This gives rise to infinite graphs as, for example, all natural numbers will exist as separate data nodes. Attributed rules contain a data node for each term in the term algebra. The attributes of the critical pairs are obtained by computing the most general unifier of the overlapped terms. However, this construction has been shown to be incomplete in [EEPT06, p.198]. The problem is avoided by requiring the severe restriction that attributes are variables or variable-free.

[5]Typed graphs can be seen as a generalization of labelled graphs [EEPT06, Fact 2.9].

Application Conditions. Graph conditions allow us to restrict the application of rules by equipping them with an extra graph that must or must not exist in the match of a rule. They emerged as a general concept of restriction [EH86], and were extended to *negative* application conditions [HHT96] and *nested* application conditions [HP09]. The dangling condition in the DPO approach can be expressed as a negative application condition.

Application conditions add to the expressive power of graph transformation. However, their use poses additional problems for constructing critical pairs — a pair of direct derivations may be independent in the sense of Definition 2.30, but one of the rules creates a structure that is forbidden by the other rule's application condition. A more elaborate version of independence and confluence was developed in [Roz97, LEO06, Lam09].

The paper [EGH$^+$12] considers confluence of rules equipped with arbitrary nested application conditions. The constructions rely on *shifting* conditions over morphisms and rules. However, since application conditions are arbitrarily nested, constructing a set of critical pairs in this setting turns out to be an undecidable problem due to nesting being equivalent to first-order graph formulas. Showing strong joinability turns out to also be undecidable.

Close to these ideas is the filtering of critical pairs who violate the normal constraints of the domain they ultimately model, represented using graph constraints [HW95]. Graph constraints are special cases of application conditions. What is important is that such critical pairs can be discarded without violating the soundness of the approach. For example, this approach is taken in the tool SyGrAV [Dec17, DKL$^+$16, KDL$^+$15]. We employ the same idea in Section 7.2.

2.3.6 Confluence Applications

Confluence has been used in various fields of computer science to reason about properties of computation.

Reduction systems are a powerful way of finitely representing an infinite set of graphs with some special property [ACPS93, BPR04]. They consist of a set of rules and a finite number of accepting graphs. A reduction system defines a language of graphs such that every normal form of a graph in the language is an accepting graph. Classical examples of graph reduction consists trees, series-parallel graphs, flowcharts. However, recognition of such languages can be a costly process since *all* normal forms are to be computed. In the case when the reduction rules are confluent, this becomes much more efficient as confluence implies uniqueness of normal forms. We will see an example of this when we study the confluence of recognizing series-parallel graphs in Chapter 7.

In [EEPT06, Ch. 14], the authors use graph transformation to implement the model transformation between UML Statecharts and Petri Nets. One of the main properties of model transformation is its functional behaviour, so the underlying graph transformation system has to be proven locally confluent and terminating. The tool AGG is used, and the graph transformation is in the context of typed attributed graphs. The source and target languages are given as typed graphs. The tool AGG is then used to construct the critical pairs of the translation rules, and thus reason about local confluence and functional behaviour of the transformation.

Conflicts between requirements of different stakeholders may be difficult and expensive to resolve during software development. [HHT02] propose a method for detecting such conflicts as early as possible by means of graph transformation. The authors use representations of UML use case, activity and collaboration diagrams as typed attributed graphs and transformation rules between such graphs. Critical pairs analysis can be used to annotate use cases and activity diagrams with extra information regarding conflicts and dependencies and to trigger a re-iteration of the requirements model to eliminate the undesired effects. Note that confluence is not strictly required, but only the detection of potential conflicts between use cases. Furthermore, not all such conflicts represent an error — if a requirements analyst decides that two use cases or activities are meant to happen alternatively, the conflict simply reflects this requirement at a semantic level.

Software refactoring is a common technique for improving the structure of object-oriented code while preserving external behaviour [MT04]. Existing tools only offer help with the automated application of such refactorings. The developer has to choose interactively which refactorings he would like to apply, and use a refactoring tool to apply these refactorings. [MTR07] propose a method for the automated detection of implicit conflicts and dependencies between refactorings. Their method is based on critical pair analysis which would allow a tool to suggest refactorings that are more appropriate in a given context and offer explanation why. The authors model refactorings as rules in the setting of typed attributed graphs, and use AGG to compute all possible conflicts between them. Then, based on the number of dependencies, some rules are suggested before others.

Visual languages are a programming paradigm where users create programs that have spatial relationships between elements rather than being textual. Their parsing can be described as a graph transformation system. [BTS00] propose an efficient parsing algorithm using critical pair analysis to delay decision between conflicting rule applications as much as possible. This means applying non-conflicting rules first and reducing the graph as much as possible before creating decision points for the backtracking.

2.4 Summary

In this chapter we have:

- given some prelimary definitions: label alphabets, graphs, graph morphisms;
- reviewed the DPO approach to applying rules in graph transformation, giving both the concrete steps and the algebraic constructions formed from two pushouts;
- reviewed the DPO approach with relabelling;

- presented the language GP 2, facilitating the high-level specification of graph programs;
- defined rule schemata, the building blocks of graph programs, allowing expressions in labels;
- explained the control constructs of graph programs together with their semantics, and given a number of example programs;
- presented the notion of *confluence*, a property of a set of rules that ensures every input has a unique result from a computation;
- studied conflicts and critical pairs, as basic elements for checking confluence, and how they are constructed;
- discussed how confluence is extended in more complicated frameworks in the general area of graph transformation, specifically involving attribution and application conditions.

Chapter 3

Independence of GP 2 Rule Schemata

In this chapter we introduce the notions of independence and conflict for rule schemata, the building blocks of graph programs, which follows the approach initiated by [EK76] for graph transformation and later extended by [HP12] for graph transformation with relabelling. *We lift the notions of independence and conflict to rule schemata*. Our main technical result, the Local Church-Rosser Theorem, establishes that independent derivations are commutative and thus lead to the same result regardless of application order. These contributions are important for two main reasons — (1) the idea of conflict is heavily used in critical pair analysis, and (2) without proving the commutativity of independent derivations, the confluence analysis based on critical pairs would be unsound.

This chapter is based on [HP16a] where we lift independence and conflicts from rule instances to rule schemata. However, that paper does not fix the underlying label alphabet, whereas we assume GP 2's fixed algebra of list expressions. Schemata are assumed to be *unconditional*, as conditions interfere with which pairs of derivations are considered in conflict.

3.1 Reasoning about conflicts

In graph transformation, the notions of independence and conflict have been widely studied and are of significant practical importance [Cor16]. Independence in its essence is a syntactic condition that ensures a pair of derivations on the same graph, i.e. executed in parallel, are applicable to each other's results. In such situations, executing the derivations in any order is guaranteed to lead to the same result, i.e. they are commutative (also known as the *diamond* property or the *Local Church-Rosser* property). Since we talk about independence in the context of derivations in parallel, we simply refer to parallel independence as independence. This notion has been of particular importance in several fields, e.g. when graphs are used to model software artefacts and rules to model changes to such artefacts [MTR07, EET11].

However, in order to reason about confluence of a graph program, one needs to lift the notion of conflict to derivations with rule schemata. And this is quite natural — confluence checking is ultimately about restricting the infinite domain of all graphs and all derivations that need to be considered. It is also not surprising as the study of independence and conflicts pre-dates critical pairs and confluence, at least in the case of graph transformation. The idea of lifting independence to the schema derivation level is to allow confluence reasoning just from the syntax of a program.

Fundamentally, the problem of (parallel) independence can be stated as this: Find a condition, called (parallel) independence, such that two rule schema direct derivations $H_1 \Leftarrow_{r_1} G \Rightarrow_{r_2} H_2$ are (parallel) independent if there are direct schema derivations $H_1 \Rightarrow_{r_2} X$ and $H_2 \Rightarrow_{r_2} X$ such that the composed derivations $G \Rightarrow_{r_1} H_1 \Rightarrow_{r_2} X$ and $G \Rightarrow_{r_2} H_2 \Rightarrow_{r_1} X$ are equivalent.

The above problem is stated rather informally, and thus opens several lines of questioning. First, we haven't stated the *matches* used by the derivations, but the idea is that the same matches are used in the joining derivations rather than computing new matches. Second, the notion of equivalent derivations means the derivations are isomorphic under the standard notion of *sequential independence*. We explicitly refer to sequential independence when necessary, and also note that sequential and parallel independence are related concepts. Third, we want the condition to be universally quantified, i.e. commutativity to hold *for all* independent derivations rather than for a particular subset. Last but not least, we want the condition to be in some sense *syntactic* meaning it can be checked just by looking at the components of the pair of derivations, as opposed to being a *semantic* condition which would be actually checking the commutativity property directly.

In this chapter, we give our answer to the above problem. In particular, we lift the notion of independence developed for derivations with relabelling in [HP12] to schema derivations, and show our main technical result that the corresponding property of commutativity holds, i.e. the correctness of the condition. We give a more intuitive characterization of independence stating that two derivations are in conflict if either deletes or relabels a common item. Furthermore, we study *generalized* rule schemata which are semantically equivalent to GP 2 rule schemata but induce less conflicts. This is achieved by means of a 'maximal' interface.

It should be noted that the notion of conflict is intrinsic to the study of critical pairs, the idea being that, since independent derivations commute and thus have the (local) confluence property, one need only look at conflicting derivations as potential sources of non-confluence. Critical pairs are a subset of all conflicts that have special properties. We cover critical pairs for GP 2 in Chapter 4. Furthermore, the commutativity property is one of the central results used in the Local Confluence Theorem (Chapter 6), and the definition of independence plays a central role in proving that critical pairs are complete, i.e. represent all possible conflicts.

3.2 Lifting Independence to Rule Schema Derivations

In this section, we explain our approach to independence and conflicts of rewriting with rule schemata. The main idea is to lift the notion of independence of deriva-

$$\begin{tikzcd}
R_1 \arrow[d, mapsto] & K_1 \arrow[l] \arrow[r] \arrow[d, mapsto] & L_1 \arrow[d, mapsto] & & L_2 \arrow[d, mapsto] & K_2 \arrow[l] \arrow[r] \arrow[d, mapsto] & R_2 \arrow[d, mapsto] \\
R_1^\alpha \arrow[d] & K_1^\alpha \arrow[l] \arrow[r] \arrow[d] & L_1^\alpha \arrow[dr, "m_1"] \arrow[drrr, dotted, "i"] & & L_2^\beta \arrow[dl, "m_2"'] \arrow[dlll, dotted, "j"'] & K_2^\beta \arrow[l] \arrow[r] \arrow[d] & R_2^\beta \arrow[d] \\
H_1 & D_1 \arrow[l] \arrow[rr, "f_1"] & & G & & D_2 \arrow[ll, "f_2"'] \arrow[r] & H_2
\end{tikzcd}$$

Figure 3.1: Independent schema derivations.

tions with relabelling. These notions have been obtained in [HP12] as a corollary of the fact that derivations with relabelling fall into the category of so-called $\mathcal{M}, \mathcal{N}$-adhesiveness and thus inherit many 'classical' results, the (algebraic) definition of independence and the Local Church-Rosser Theorem being part of them.

Informally, two derivations with relabelling are independent if every common item of the two matches is an interface item and no such common item is relabelled by either derivation. This intuition can be formally stated as an "existence-of-morphisms" statement (Chapter 2). To lift this, we say that two rule schema direct derivations are independent if their components, which are derivations with relabelling, are parallel independent.

Definition 3.1 (Independence of schema derivations)**.** Two rule schema direct derivations $G \Rightarrow_{r_1,m_1,\alpha} H_1$ and $G \Rightarrow_{r_2,m_2,\beta} H_2$ are *independent* as in Figure 3.1 if their component derivations with relabelling $G \Rightarrow_{r_1^\alpha,m_1} H_1$ and $G \Rightarrow_{r_2^\beta,m_2} H_2$ are parallel independent, meaning the following:

Two direct derivations with relabelling $H_1 \Leftarrow_{r_1^\alpha,m_1} G \Rightarrow_{r_2^\alpha,m_2} H_2$ are *independent* if there exist morphisms $i : L_1^\alpha \to D_2$ and $j : L_2^\alpha \to D_1$ such that $f_2 \circ i = m_1$ and $f_1 \circ j = m_2$ where $f_1 : D_1 \to G, f_2 : D_2 \to G$ are inclusions.

□

Definition 3.2 (Conflict between schema derivations)**.** Two rule schema direct derivations $G \Rightarrow_{r_1,m_1,\alpha} H_1$ and $G \Rightarrow_{r_2,m_2,\beta} H_2$ are in *conflict* if they are not independent. □

The reason for introducing independence using morphisms and not in a set-theoretic way is pragmatical – it is easier to use in the proof of the Local Church-Rosser Theorem for derivations with relabelling, heavily based on diagrammatic constructions, but also in the proof that our critical pairs are complete (Chapter 4), also based on diagrammatic constructions. Even so, it is useful to provide an equivalent characterization of independence, and we give such a characterization in Lemma 4.

Several observations about independence can be noted. Firstly, the right-hand sides of rules do not play a role when determining independence, i.e. whether the rules create items is irrelevant. Secondly, the mappings between the left-hand

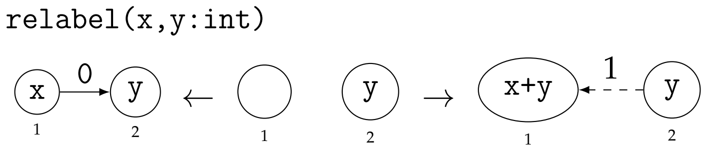


Figure 3.2: The `relabel` rule.

graphs and the opposing intermediate graphs (D_1 and D_2) should preserve labels. Thirdly, the interfaces of schemata play an important role. The conventions that left-hand graphs are totally labelled and interface graphs consist only of unlabelled nodes means too many derivations would be considered dependent regardless of the semantics of the rule schema.

The first and second of the above points lead us to define *conflict types*, a classification of conflicts based on whether they involve deleting a common matched item or relabel such an item. The third observation leads us to *generalize* the interface of a schema in order to reduce the number of potential conflicts.

In the following example(s), we consider the schema `relabel` over integer variables `x,y` , given as an expanded version showing its interface graph, given in Figure 3.2. The schema is performing integer addition over the labels: it relabels node 1, and deletes/recreates the 0 edge (in the opposite direction). Note that we have allowed node 2 to keep its label. We will see in Section 3.4 that this is not problematic, as the schema where node 2 is unlabelled in the interface (and thus conforms to the language's interface convention) can be generalized to the above version without compromising the semantics.

Example 8 (Independent derivations). Figure 3.3 gives an example of two independent derivations. The derivations involve the schema `relabel` of Figure 3.2, instantiated in different ways. There are no common items, and the required mappings are the original matches. To avoid repeating the rule schema as in Figure 3.1, we only show the instantiations with the assignments α and β that assign `x` to 1 and `y` to 2 and 3, respectively.

□

Example 9 (Conflicting derivations). Figure 3.4 shows two direct derivations $H_1 \Leftarrow G \Rightarrow H_2$ that are derivations of the `relabel` rule schema. Unlike the derivations of Figure 3.3, here the matches have a common node (node 1) which gets relabelled by both derivations. The derivations are *not* parallel independent, i.e. are in conflict: there are no morphisms $L_1^\alpha \to D_2$ and $L_2^\beta \to D_1$ with the desired properties. The problem is the premorphisms (equal to the matches) are not label-preserving, and hence not graph morphisms.

□

Characterization. Now we are able to state our characterization of independence in a set-theoretic style, which simply means that we directly work with the common items of the matches rather than talk about morphisms between the left-hand

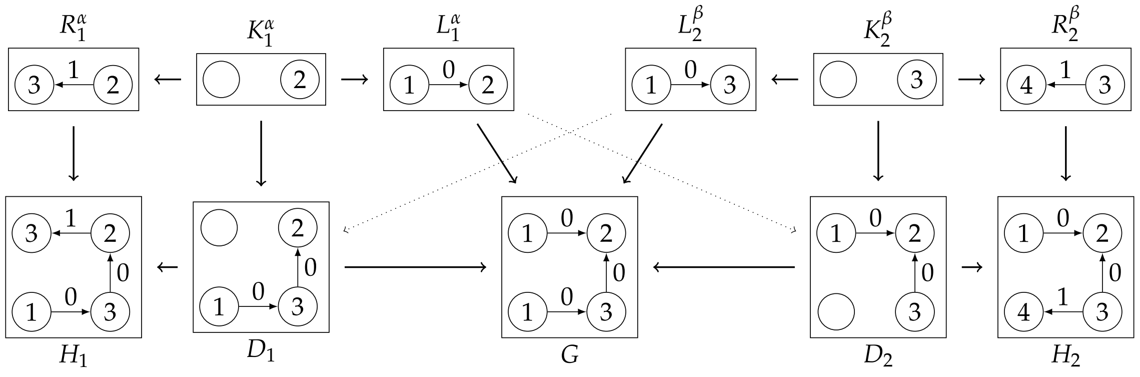


Figure 3.3: Independent derivations involving `relabel`.

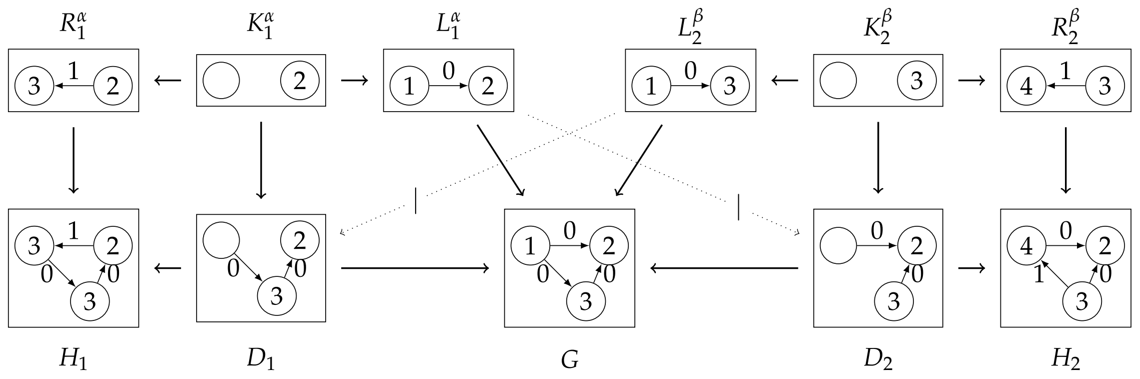


Figure 3.4: Conflicting derivations involving `relabel`.

graphs and opposite intermediate graphs.

Lemma 4 (Characterization of independence). *Two direct derivations $H_1 \Leftarrow_{r_1,m_1} G \Rightarrow_{r_2,m_2} H_2$ are parallel independent if and only if for all items $x_1 \in L_1^\alpha$ and $x_2 \in L_2^\alpha$ such that $m_1(x_1) = m_2(x_2)$, we have the following:*

1. *$x_1 \in K_1^\alpha$ and $x_2 \in K_2^\alpha$, and*
2. *$l_{K_1^\alpha}(x_1) \neq \bot$ and $l_{K_2^\alpha}(x_2) \neq \bot$.*

□

Rephrasing the condition, we are saying that each common item of G is (1) preserved by both derivations, i.e. the pre-images of that common item in the rules are interface items, and (2) the pre-images of the common items in the interface have labels, i.e. are not relabelled. Note that here we are using the injectivity and totality of the matches, meaning that each common item has a unique pre-image along each match.

The proof of the characterization is relatively short - the first condition is the original set-theoretic condition of [EK76] that requires the intersection of the matches to be a subset of the intersection of the interfaces; the second condition correlates directly with the requirement that the induced morphisms are label-preserving and that left-hand graphs are totally labelled.

Conflict type. Depending on which part of the independence condition does not hold, we characterize the associated conflict by giving it a conflict *type*. Here we exploit the fact that graph morphisms are premorphisms that preserve labels, i.e. the existence-of-premorphisms condition means the structure (nodes/edges) is preserved, while the condition of label preservation means the such nodes and edges are labelled in the interface and intermediate graph.

Definition 3.3 (Conflict type)**.** Two derivations in conflict are classified as one of the following *conflict types*:

- (delete-use) if at least one of the derivations deletes a common item, i.e. there does not exist a pair of premorphisms $i : L_1^\alpha \to D_2$ and $j : L_2^\alpha \to D_1$ such that $f_2 \circ i = m_1$ and $f_1 \circ j = m_2$
- (relabel) if at least one of the derivations relabels a common item, i.e. the above premorphisms exist but at least one of them is *not* a graph morphism

□

Without loss of generalization, we have skipped talking about the symmetric versions, e.g. use-delete and delete-delete. Furthermore, it is useful to note that a conflicting derivation might fall into both conflict types when a common item is deleted by the first and relabelled by the second derivation (delete-relabel). For example, the conflict in Figure 3.7 is a relabelling conflict (type) as the common nodes are preserved but relabelled by the derivations.

3.3 Local Church-Rosser Theorem

In this section we show the commutativity property of independent schema derivations. Recall that a pair of derivations is independent if neither deletes or relabels common graph items. We now prove that, using this formulation, independent derivations can be applied in any order, thus leading to the same result. Again, we lift the results of [HP12] to schema derivations. More specifically, we use Theorem 2 of [HP12] that shows the commutativity of independent derivations in any so-called $\mathcal{M},\mathcal{N}$-adhesive category and Theorem 3 of the same paper that shows that rewriting using rules with relabelling is $\mathcal{M},\mathcal{N}$-adhesive. However, before we do so, we quickly formally define *sequential* independence, which is a formality that is necessary to describe the equivalence of the resulting derivations.

$$\begin{tikzcd}
&&&& L_1 \arrow[ddr, mapsto] &&&& L_2 \arrow[ddl, mapsto] &&&& \\
&& K_1 \arrow[urr] \arrow[dll] \arrow[ddr, mapsto] &&&&&&&& K_2 \arrow[ull] \arrow[drr] \arrow[ddl, mapsto] && \\
R_1 \arrow[ddr, mapsto] &&&&& L_1^\alpha \arrow[ddr, hook, "m_1"'] && L_2^\beta \arrow[ddl, hook, "m_2"] &&&&& R_2 \arrow[ddl, mapsto] \\
&&& K_1^\alpha \arrow[urr] \arrow[dll] \arrow[ddr] &&&&&& K_2^\beta \arrow[ull] \arrow[drr] \arrow[ddl] &&& \\
& R_1^\alpha \arrow[ddr] &&&&& G &&&&& R_2^\beta \arrow[ddl] & \\
&&&& D_1 \arrow[urr] \arrow[dll] &&&& D_2 \arrow[ull] \arrow[drr] &&&& \\
&& H_1 &&&&&&&& H_2 && \\
L_2^\beta \arrow[urr, "m_2'"] &&&& \overline{D_2} \arrow[ull] \arrow[drr] &&&& \overline{D_1} \arrow[urr] \arrow[dll] &&&& L_1^\alpha \arrow[ull, "m_1'"'] \\
&& K_2^\beta \arrow[ull] \arrow[urr] \arrow[drr] &&&& X &&&& K_1^\alpha \arrow[ull] \arrow[urr] \arrow[dll] && \\
&&&& R_2^\beta \arrow[urr] &&&& R_1^\alpha \arrow[ull] &&&&
\end{tikzcd}$$

Figure 3.5: Church-Rosser theorem for rule schemata.

Definition 3.4 (Sequential Independence)**.** Two schema direct derivations $G \Rightarrow_{r_1,m_1,\alpha} H_1 \Rightarrow_{r_2,m_2',\beta} H_2$ are *sequentially independent* if their component derivations with relabelling $G \Rightarrow_{r_1^\alpha,m_1} H_1 \Rightarrow_{r_2^\beta,m_2'} H_2$ are sequentially independent, i.e. there exist morphisms $i : R_1^\alpha \to D_2$ and $j : L_2^\alpha \to D_1$ such that $f_2 \circ i = m_1$ and $f_1 \circ j = m_2$ where $f_1 : D_1 \to G, f_2 : D_2 \to G$ are inclusions. □

Lemma 5 ([HP12])**.** *Given independent derivations $G \Rightarrow_{r_1^\alpha,m_1} H_1$ and $G \Rightarrow_{r_2^\beta,m_2} H_2$, there is a graph X and direct derivations $H_1 \Rightarrow_{r_2^\beta,m_2'} X$ and $H_2 \Rightarrow_{r_1^\alpha,m_1'} X$ that are sequentially independent.*

Proof. By combining Theorem 2 and Theorem 3 of [HP12]. □

Here r_1^α, r_2^β are rule instances of any two schemata; the matches m_1', m_2' are derived from m_1, m_2 using the morphisms $L_1^\alpha \to D_2, L_2^\beta \to D_1$ whose existence is guaranteed by the definition of independence.

With this lemma, we are now able to state our main result, namely that independent derivations commute.

Theorem 3.1 (Local Church-Rosser Theorem)**.** *Given two independent rule schema direct derivations $G \Rightarrow_{r_1,m_1,\alpha} H_1$ and $G \Rightarrow_{r_2,m_2,\beta} H_2$, there is a graph X and rule schema direct derivations $H_1 \Rightarrow_{r_2,m_2',\beta} X$ and $H_2 \Rightarrow_{r_1,m_1',\alpha} X$. Moreover $G \Rightarrow_{r_1,m_1,\alpha} H_1 \Rightarrow_{r_2,m_2',\beta} X$ as well as $G \Rightarrow_{r_2,m_2,\beta} H_2 \Rightarrow_{r_1,m_1',\alpha} X$ are sequentially independent.* □

Proof. From Lemma 5 and the definition of independence, we know that independence of the derivations with relabelling $G \Rightarrow_{r_1^\alpha,m_1} H_1$ and $G \Rightarrow_{r_2^\beta,m_2} H_2$ implies the existence of a graph X and direct derivations $H_1 \Rightarrow_{r_2^\beta,m_2'} X$ and $H_2 \Rightarrow_{r_1^\alpha,m_1'} X$. This is illustrated in Figure 3.5. Here $m'_1 : L_1^\alpha \to H_2$ is the composition $L_1^\alpha \to D_2 \to H_2$ where the first morphism is guaranteed by the definition of independence. Similarly for m'_2.

The direct derivations $G \Rightarrow_{r_1^\alpha, m_1} H_1$ and $G \Rightarrow_{r_2^\beta, m_2} H_2$ involve rule instances of the schemata r_1 and r_2. When used together with the morphisms, we get that there are rule schema direct derivations $H_1 \Rightarrow_{r_2, m'_2, \beta} X$ and $H_2 \Rightarrow_{r_1, m'_1, \alpha} X$. Furthermore, it also follows that both $G \Rightarrow_{r_1, m'_1, \alpha} H_1 \Rightarrow_{r_2, m'_2, \beta} X$ and $G \Rightarrow_{r_2, m'_2, \beta} H_2 \Rightarrow_{r_1, m'_1, \alpha} X$ are sequentially independent. □

Example 10 (Commutating independent derivations). Figure 3.6 shows the commutativity of the independent derivations of Example 8 involving instances of the schema `relabel`. The top derivations are the original $G \Rightarrow H_1$ (top left) and $G \Rightarrow H_2$ (top right), and the bottom derivations are the joining derivations $H_1 \Rightarrow X$ (bottom left) and $H_2 \Rightarrow X$ (bottom right).

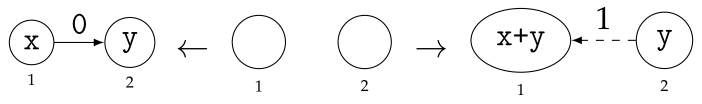
3.4 Generalized rule schemata

In the previous sections we discussed the independence condition and associated commutativity property without much attention to the practical aspects of the condition. That is, from the point of view of a confluence checker, it would be desirable to analyse as few conflicts as necessary to establish/disprove confluence. As already observed, the interface of schemata plays a crucial role in determining whether derivations are independent.

The main problem we explore in this section is this: how to relax the restriction of a rule schema interface such that pairs of derivations which have been previously in conflict are now considered independent, but without changing the semantics of schema application nor the established notion of independence.

To illustrate the problem, consider the following GP 2 schema `relabel2`, a variant of `relabel` given in Figure 3.2, that adheres to the convention that the interface consists of unlabelled nodes only:

`relabel2(x,y:int)`



Operationally, the schema relabels node 2 without needing to (it receives the same label in the right-hand graph). This means any pair of derivations where node 2 is a common node would be in conflict, but the derivations would have the Church-Rosser property given no other items are reason for a conflict.

Our solution is to lift the restriction on GP 2 schema interfaces, and propose so-called *generalized* rule schemata which can be obtained algorithmically from existing schemata by inserting additional information in a schema's interface in

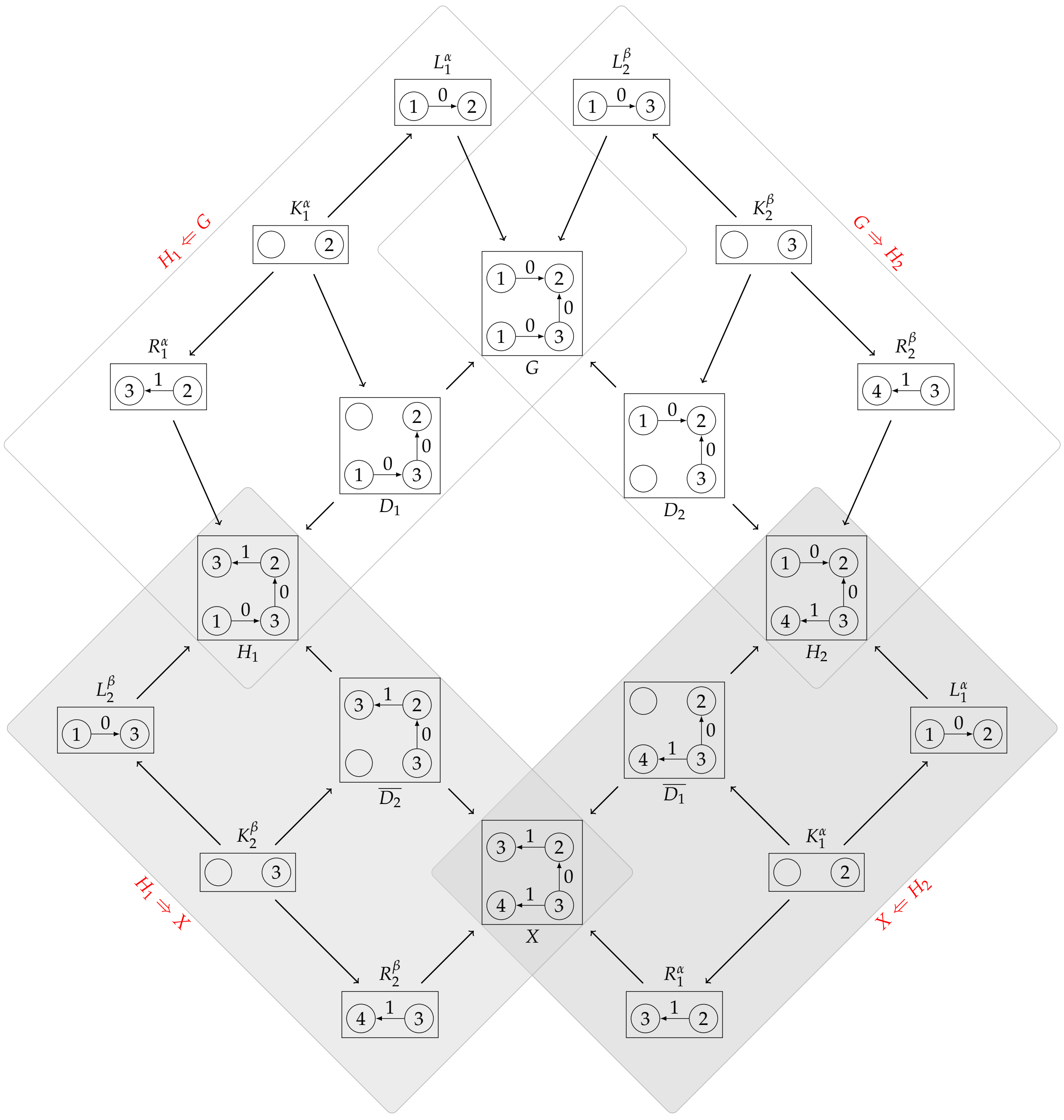


Figure 3.6: Commutativity diagram of independent derivations.

the form of labels and edges. This approach can be viewed as *rule transformation* where the objects under transformation are rules rather than graphs.

Formally, our algorithm takes a schema and returns an equivalent generalized schema with the same left- and right-hand graphs but a 'maximal' interface graph. Our algorithm does not change the semantics of the transformed schema, and thus we can simply equate schemata with their generalized versions, and there is always a unique generalized schema. There are three types of changes to the interface: (1) nodes which are not relabelled by the original schema become labelled in the interface, using the same label; and (2) edges which get reinserted in the same position become preserved as members of the interface; and (3) the edges inserted in the previous step are labelled given any label attached to their images are also labelled.

Definition 3.5 (Generalized rule schema)**.** Given a rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, its *generalized* rule schema $r' = \langle L \leftarrow K' \rightarrow R \rangle$ contains the same left- and right-hand graphs as r but the interface K' obtained from K as follows:

1. for each node $n \in K$, check if it has the same labels in L and R – if $l_L(n) = l_R(n)$ then $l_{K'}(n) = l_L(n)$
2. for each deleted edge $e \in L - K$ between preserved nodes, check if there exists a corresponding created edge $e' \in R$ with the same source and target ($s(e') = s(e)$ and $t(e) = t(e')$), if yes then insert e in K'
3. For each edge added in the previous step, check if the corresponding edge has the same label – if $l_R(e') = l_L(e)$ then $l_{K'}(e) = l_L(e)$

□

This definition specifies how to obtain the generalized rule schema. For example, the schema `relabel2` has its generalized version as `relabel`: node 2 is preserved and has the same label in L and R and step 1 of generalization gives it the label `y` in K'. Note the edge between nodes 1 and 2 is *not* part of the interface as there is not a corresponding edge in R in the same direction.

The above process does not insert nodes in the interface K'. This is because nodes have special status due to the dangling condition — in GP 2, the programmer has explicit control over which nodes are part of the interface, and thus can control whether the rule is applicable or not. Changing the interface w.r.t. addition of nodes would change the situations in which the rule is applicable. For example, the rule $\langle \bigcirc \leftarrow \varnothing \rightarrow \bigcirc \rangle$ is only applicable to isolated empty-labelled nodes (nodes not connected to any other nodes) whereas the rule $\langle \bigcirc \leftarrow \bigcirc \rightarrow \bigcirc \rangle$ is applicable to all empty-labelled nodes.

Example 11 (Reducing conflicts). In Figure 3.7 is a pair of conflicts of the schema `relabel2`. The derivations are in conflict because the common node 2 gets relabelled by both derivations, albeit to the same value. In the generalized version of the schema (which is just the schema `relabel` shown previously), node 2 preserves its label, and thus the same situation is actually a non-conflict. This is depicted in Figure 3.8 using the same graph G, same schema instances, and same matches as above. The only changes are to the interfaces and the intermediate graphs D_1 and D_2. This time the derivations are independent.

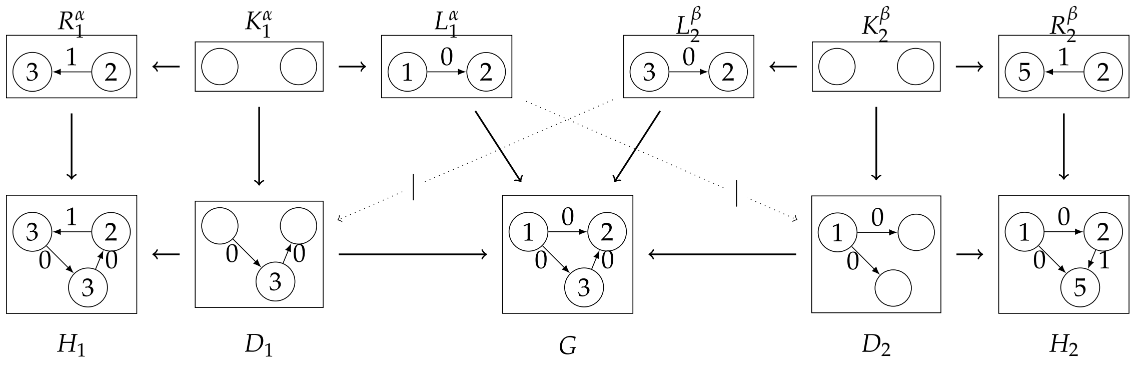


Figure 3.7: Conflicting derivations involving `relabel2`.

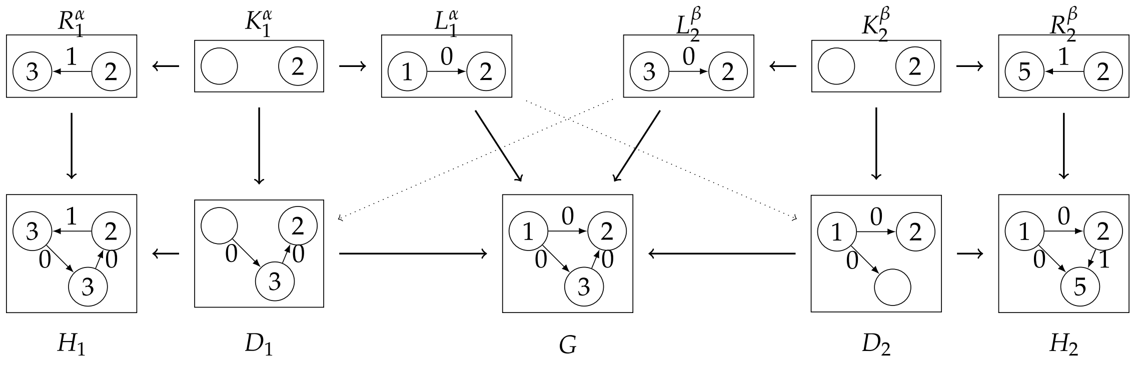


Figure 3.8: Independent derivations involving the generalized `relabel2`.

Generalized schemata can be seen as a conservative extension of GP 2 schemata for the following reasons. First, only the interface of a schema is changed, and the underlying theoretical framework of rules with relabelling already allows for such an extension. Second, the change of the interface is such that edges that are deleted/created are rather preserved, and labels that are reinstated to the same value are rather not changed at all.

These extensions need not constitute a change to the semantics nor implementation of GP 2. The purpose of these is to reduce the number of conflicts during confluence analysis rather than during program execution, meaning the existing compiler would not need changes in its internal workings to handle generalized schemata. Furthermore, nor would the language specification need changing because these extensions are in the context of confluence checking. However, changes to the way the interface of a schema is specified would be welcome as it would allow the programmer more control over the meaning of a schema.

3.5 Related work

The earliest formalization of independence in double-pushout graph transformation with injective rules was given by [EK76] that expressed independence as a set-theoretic condition (the intersection of matches is a subset of the intersection of interfaces) and proved the associated Local Church-Rosser Theorem for independent derivations. For term rewriting, steps are independent if they have a non-critical overlap [BS01]. The algebraic formulation (as existence-of-morphisms) has been used when extending the notion of independence to categories other than graphs such as High Level Replacement systems [EHKP91], DPO transformations in Adhesive categories [LS05], graph transformation with negative application conditions [LEO06], and many others. These richer notions of rewriting also introduce more involved conflict types, e.g. when negative application conditions are present, derivations can be in *produce-forbid* conflict where a rule creates a structure forbidden by the other rule [Lam09].

[KDL$^+$15] extend the notion of independence by giving a separate condition called 'direct confluence' that checks the Church-Rosser property for steps that are dependent but produce isomorphic graphs when attribute operations are taken into account. If this is the case, then the steps are not conflicting, i.e. the relationship between independence and conflicts is refined.

The proof of equivalence between the set-theoretic formulation and the existence-of-morphisms condition has been proven for graphs and injective rules e.g. in [EEPT06, Fact 3.18] by exploiting specific properties of pushouts in the category of graphs.

3.6 Summary

In this chapter, we have:

- defined what it means for pairs of schema derivations to be *independent* and in *conflict*, by lifting the existing notions for derivations with relabelling
- shown a characterization of independence that states no common nodes are deleted or relabelled by a pair of independence derivations;
- proven that independent derivations commute (Local Church-Rosser Theorem), by lifting the corresponding result about derivations with relabelling;
- proposed *generalized* rule schemata which are a conservative extension of GP 2 schemata but have a 'maximal' interface, and thus fewerf conflicts.

Chapter 4

Symbolic Critical Pairs for GP 2

In Chapter 3, we developed the notions of independence and conflict of rule schema derivations. The reason for this distinction is that independent pairs of direct derivations commute, which has been called the Church-Rosser property [BN98, CR36], whereas conflicting pairs of derivations may not be confluent at all.

The next step we take in this thesis is to study pairs of direct derivations which are in conflict. In this chapter we develop critical pair analysis for sets of rule schemata, which follows the approach initiated by [KB70] for term rewriting and later by [Plu93] for graph transformation. We begin by introducing symbolic critical pairs, which are pairs of derivations at the level of schemata, i.e. labelled with expressions, that are minimal and in conflict. We work at the level of graphs labelled with expressions to avoid an infinite number of such pairs. Then we give an algorithm for their construction, and show the set of critical pairs is complete (that is, every pair of conflicting derivations is represented by a critical pair) and finite under suitable restrictions. Finally, we briefly review related work on critical pairs for graph transformation.

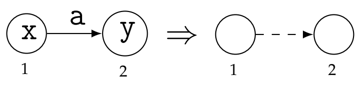
This chapter is based on [HP16b]. Schemata are assumed to be unconditional.

4.1 Confluence Analysis with Critical Pairs

Critical pair analysis is a technique for rigorously reasoning about the confluence of a set of rewriting rules. Central to the approach is the notion of a *critical pair* - a pair of direct derivations $H_1 \Leftarrow_{r_1} G \Rightarrow_{r_2} H_2$ with suitable properties. Roughly speaking, the pair has to be (1) *minimal* meaning the graph G is an overlap of the left-hand graphs of the rules r_1 and r_2, i.e. all items in G originate from the rules' left-hand graphs; and (2) *conflicting*, i.e. either derivation modifies an item matched by the other. The idea of critical pair analysis is to restrict the state space of a confluence checker: instead of checking all pairs of conflicting derivations for confluence, check only a finite set of representatives. (Confluence of critical pairs is often referred to as *joinability*.) If all critical pairs are joinable, then the set of rules is locally confluent.

The minimality property of critical pairs gives us an intuition of how to construct them, namely by considering the rule schemata of a GP 2 program. Take, for example, the following schema:

`unlabel(a,x,y:list)`



and consider an overlap of the left-hand graph of `unlabel` with itself such that the two resulting matches are different. An obvious critical pair would be

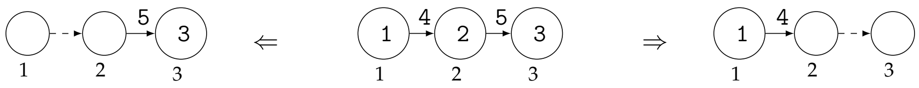
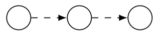
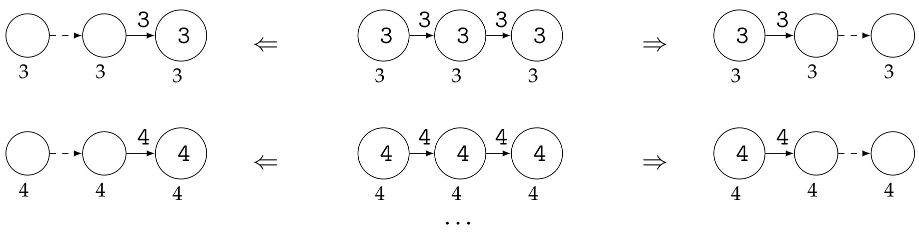


Figure 4.1: A conflict of `unlabel` with itself.

where the middle graph is obtained by overlapping the left-hand graph of `unlabel`'s instance with a copy of itself, and the graphs on either side are the results of applying the schema in conflicting ways. The pair is in conflict because both derivations relabel a common node (2) to the empty list. The critical pair represents all pairs of derivations where `unlabel` is in conflict with itself by overlapping on a node. The critical pair is joinable since `unlabel` can be applied to either result graph but with a different assignment for its variables, resulting in the common graph 
.

Constructing and using critical pairs is appealing because they are constructed by overlapping the rule schemata of a program rather than depending on the input graphs. (An analysis that accounts for properties of the input graph is more complicated, and will be discussed in our Shortest Distances Case Study of Section 7.2.) In a sense, critical pairs capture the *conflict reasons* of the rule set whilst avoiding the need to look at all possible conflicts [LEO08].

To be useful, the set of critical pairs needs to have several properties. First, it should be *finite* and *computable*. That is, there should exist a terminating algorithm that, given a set of rules, returns exactly all of its associated critical pairs. Typically, proving the correctness of such a construction involves appealing to the definition of critical pairs. The above example overlap however illustrates that there might be an infinite set of critical pairs due to GP's infinite label algebra – different assignments for the variables of `unlabel` result in a different critical pairs:



All we did was change the assignments of the schema to different concrete values. To avoid this, we instead introduce critical pairs labelled with expressions.

Another desirable property of the set of critical pairs is *completeness*. That is, each possible conflict is represented by a critical pair (according to some notion of representation). Unfortunately, given an infinite label set, it is not always possible to achieve completeness depending on how critical pairs are defined, as already shown in Chapter 2 for the case of attributed graph transformation. Instead, one needs to impose certain syntactic restriction on rules, and we use a so-called *left-linearity* condition that forbids repetition of label variables (see Remark 3).

Last but not least, joinability of the critical pairs should imply the confluence of the conflicts that the critical pairs represent. That is, if one can show that each critical pair is joinable, then all conflicts are confluent. (For the time being, for a terminating set of rules we assume joinability is a decidable property.) However, it turns out joinability of all critical pairs need not always imply confluence of the rewrite rules, in contrast to the situation in term rewriting as discussed in Chapter 2. The reason is the way critical pairs are embedded into larger context, as observed in [Plu93]. Instead, a stronger notion of joinability (*strong joinability*) needs to be considered. It turns out that strong joinability of all critical pairs is sufficient for showing local confluence, but is not a necessary condition. From now on we refer to strong joinability simply as joinability, unless explicitly distinguished. We will explore in detail joinability and confluence implication in Chapter 6.

4.2 Symbolic Critical Pairs

Having given an intuition for what critical pairs are, we now proceed to give a proper definition. This definition relies on the definition of the system of equations induced by overlapping two left-hand graphs.

Informally, a pair of derivations $T_1 \overset{r_1,m_1,\sigma}{\Longleftarrow} S \overset{r_2,m_2,\sigma}{\Longrightarrow} T_2$ between graphs labelled with expressions is a critical pair if it is in conflict and minimal. Minimality means the pair of matches (m_1, m_2) is jointly surjective – the graph S can be considered as a suitable overlap of L_1^σ and L_2^σ. Computing overlaps is the first step of our critical pair construction algorithm; we ignore the labels of overlaps for the time being.

Definition 4.1 (Overlap; Induced System of Equations). Two graphs L_1 and L_2 labelled with expressions are *overlapped* if there exists a pair of jointly surjective premorphisms $m_1 : L_1 \to S, m_2 : L_2 \to S$ to a graph S. The graph S is the *overlap* of L_1 and L_2. An overlap induces a system of (label) equations EQ defined as:

$$\mathrm{EQ}(L_1 \overset{m_1}{\to} S \overset{m_2}{\leftarrow} L_2) = \{l_{L_1}(a) \overset{?}{=} l_{L_2}(b) \mid (a,b) \in L_1 \times L_2 \text{ with } m_1(a) = m_2(b)\}$$

□

Note the definition is symmetric in that it does not matter the way we write the surjective morphisms, i.e. we consider the systems $\mathrm{EQ}(L_1 \overset{m_1}{\to} S \overset{m_2}{\leftarrow} L_2)$ and $\mathrm{EQ}(L_2 \overset{m_2}{\to} S \overset{m_1}{\leftarrow} L_1)$ to be equivalent.

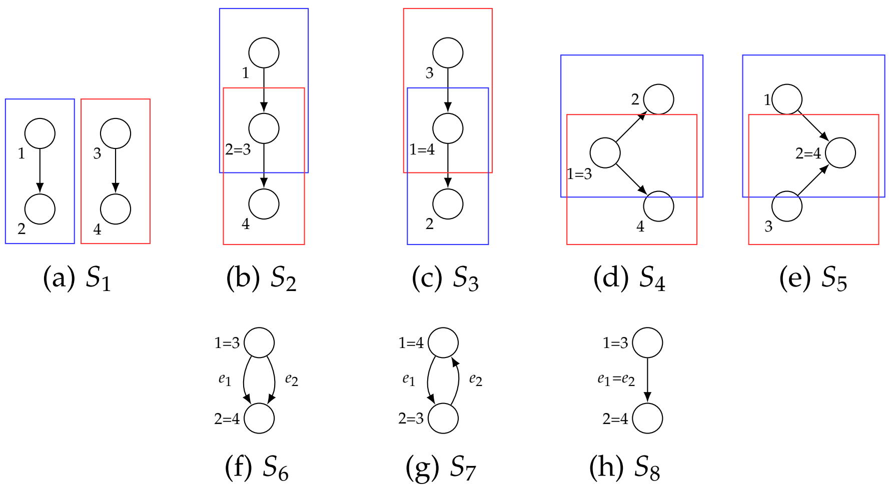
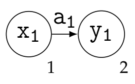
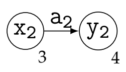


Figure 4.2: Overlaps of `unlabel` with itself.

Example 12 (Overlap; Induced System of Equations). Consider again the schema `unlabel`, more specifically its left-hand graph

$L_1 =$ 

and its copy

$L_2 =$ 

where in L_2 for clarity we have changed the nodes identifiers and indexed the variables. The overlaps of L_1 and L_2 (when ignoring labels) are given in Figure 4.2 where we have used coloured boxes to distinguish between the matches for L_1 (blue) and L_2 (red). Overlapped nodes/edges are given identifiers like (2=3) to denote which nodes/edges of L_1 and L_2 overlap.

In the graph S_1 no items overlap, i.e. the graph is the disjoint union of the unlabelled versions of L_1 and L_2, while in S_8 all items are overlapped. The other graphs ($S_2 \ldots S_7$) have a varying amount of overlapped items. Note that S_6 and S_7 are denoted without boxes for L_1 and L_2 since the matches overlap on both nodes. Instead, the edge identifiers are given, from which the matches become obvious. Note that S_2 is the overlap of the conflicting pair in Figure 4.1.

Each overlap induces a system of label equations, shown in Figure 4.3 The intuition is that these systems of equations need to be solved in order to decide the labels of each overlap.

Each set of equations is constructed by considering each pair of overlapped items. S_1 is the disjoint union (no common items) and hence the induced system of equations is empty, while for S_2 we have to require that the labels of node 2 (in L_1) and nodes 3 (in L_2) are made equal. □

A substitution σ is a mapping from variables to expressions. For critical pairs, σ is taken from a complete set of unifiers of the induced system of equations.

$$\begin{aligned}
\mathrm{EQ}(S_1) &= \{\} \\
\mathrm{EQ}(S_2) &= \{\mathtt{y}_1 \overset{?}{=} \mathtt{x}_2\} \\
\mathrm{EQ}(S_3) &= \{\mathtt{y}_2 \overset{?}{=} \mathtt{x}_1\} \\
\mathrm{EQ}(S_4) &= \{\mathtt{x}_1 \overset{?}{=} \mathtt{x}_2\} \\
\mathrm{EQ}(S_5) &= \{\mathtt{y}_1 \overset{?}{=} \mathtt{y}_2\} \\
\mathrm{EQ}(S_6) &= \{\mathtt{x}_1 \overset{?}{=} \mathtt{x}_2, \mathtt{y}_1 \overset{?}{=} \mathtt{y}_2\} \\
\mathrm{EQ}(S_7) &= \{\mathtt{x}_1 \overset{?}{=} \mathtt{y}_2, \mathtt{y}_1 \overset{?}{=} \mathtt{x}_2\} \\
\mathrm{EQ}(S_8) &= \{\mathtt{x}_1 \overset{?}{=} \mathtt{x}_2, \mathtt{y}_1 \overset{?}{=} \mathtt{y}_2, \mathtt{a}_1 \overset{?}{=} \mathtt{a}_2\}
\end{aligned}$$

Figure 4.3: Systems of equations induced by each of the overlaps $S_1, \ldots, S_8$.

Informally, given a equation $a \overset{?}{=} b$ between *simple* list expressions (i.e. label expressions occurring in left-hand graphs of rule schemata), a unifier for that equation is a substitution such that $\sigma(a) \approx \sigma(b)$ where $\approx$ denotes equivalence of GP 2 list expressions. The intuition extends to systems of equations where a unifier unifies each equation given. We give a proper definition of a unifier and complete set of unifiers in Chapter 5. For now, we assume there exists an algorithm UNIF to compute such a set of unifiers for a given system of equations. The purpose of substitutions is to construct the rule instances of the critical pair.

Formally, we define critical pairs as minimal conflicting derivations that are labelled with expressions (rather than with GP 2 host graph labels). Each symbolic critical pair represents a possibly infinite set of conflicting host graph derivations. What is special about our critical pairs is that they show the conflict in the most abstract way. To disambiguate between the critical pairs of our approach and conventional critical pairs found in literature (e.g. [Plu93]), we introduce them as *symbolic*.

Definition 4.2 (Symbolic Critical Pair)**.** A *symbolic critical pair* is a pair of direct derivations $T_1 \overset{r_1,m_1,\sigma}{\Longleftarrow} S \overset{r_2,m_2,\sigma}{\Longrightarrow} T_2$ on graphs labelled with expressions such that:

(1) σ is a substitution in $\mathrm{UNIF}(\mathrm{EQ}(L_1 \xrightarrow{m_1} S \xleftarrow{m_2} L_2))$ where L_1 and L_2 are the left-hand graphs of r_1 and r_2, m_1 and m_2 are injective premorphisms;

(2) the pair of derivations is in conflict;

(3) $S = m_1(L_1^\sigma) \cup m_2(L_2^\sigma)$, meaning that$S$ is minimal;

(4) $r_1^\sigma = r_2^\sigma$ implies $m_1 \neq m_2$. □

Remark 3 (Distinct variables; left-linearity). For the rest of this chapter, we assume the variables occurring in rule schemata are distinct, which can always be achieved by variable renaming. This is useful when we have to overlap left-hand graphs of schemata, as in the previous example – the graph L_2 is a copy of L_1 with its variables renamed. This assumption allows us to consider pairs of derivations using the same assignment or substitution for instantiation.

We also assume rule schemata are *left-linear* - in a left-hand graph of a schema, no list variable is shared between different node/edge labels. This will be necessary for our unification algorithm UNIF to give a finite set of unifiers for a system of equations (e.g. see EQ(S_7) in Example 12). □

Remark 4 (Symbolic derivations). The graphs S, T_1, and T_2 of a symbolic critical pair are labelled with expressions, and hence the derivations $S \Rightarrow T_1$ and $S \Rightarrow T_2$ are not ordinary GP 2 direct derivations as defined in Chapter 2 which only consider derivations on host graphs. For the duration of this chapter, we abuse notation while in Chapter 6 we properly define the rewrite relation $\Rrightarrow$ on graphs labelled with expressions. □

Example 13 (Symbolic critical pair of `unlabel`). An example symbolic critical pair of the `unlabel` rule is shown in Figure 4.4 where the middle graph is obtained by overlapping the left-hand graph of the `unlabel` schema with itself, and the graphs on either side are the results of applying the schema in conflicting ways. Note that this symbolic critical pair looks very similar to the pair of conflict-

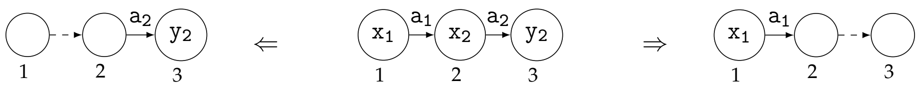


Figure 4.4: A symbolic critical pair of `unlabel` with itself.

ing derivations in Figure 4.1. In fact, they are related by the instantiation $\lambda = \{\mathtt{x}_1 \mapsto \mathtt{1}, \mathtt{y}_1 \mapsto \mathtt{2}, \mathtt{y}_2 \mapsto \mathtt{3}, \mathtt{a}_1 \mapsto \mathtt{4}, \mathtt{a}_2 \mapsto \mathtt{5}\}$.

There are 5 more symbolic critical pairs obtained by self-overlapping `unlabel`, induced by the overlaps $S_3 \ldots S_7$ of Figure 4.2. We will list them all in the next subsection. The overlap S_1 does not lead to a critical pair since the derivations would be independent, and the overlap S_8 violates the (distinction) condition (4) requiring the matches to be different given the same rule instances. □

4.3 Construction and Finiteness

Having defined critical pairs in our setting, in this subsection we give an algorithm for their construction. Informally, this requires computing graph overlaps and then solving the induced system of equations using our unification algorithm (Chapter 5), giving rise to a set of unifiers per overlap. The pair of derivations is checked for being in conflict, and if so, it is a symbolic critical pair.

```
Input : Left-linear rule schemata $r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle$ and
        $r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle$
output: Set of symbolic critical pairs over $r_1$ and $r_2$
1  Compute all overlaps of $L_1$ and $L_2$, giving rise to pairs of jointly surjective
     premorphisms $(L_1 \overset{m_1}{\rightarrow} S \overset{m_2}{\leftarrow} L_2)$ into an unlabelled graph $S$
2  foreach overlap $(L_1 \overset{m_1}{\rightarrow} S \overset{m_2}{\leftarrow} L_2)$ do
3      Check that $m_1$ and $m_2$ satisfy the dangling condition w.r.t. $r_1$ and $r_2$.
4      Check that the conflict condition is satisfied, i.e. at least one common
         item is deleted or relabelled.
5      foreach unifier $\sigma$ in UNIF(EQ($L_1 \overset{m_1}{\rightarrow} S \overset{m_2}{\leftarrow} L_2$)) do
6          Instantiate $r_1$ and $r_2$ using $\sigma$ to obtain the rules $r_1^\sigma$ and $r_2^\sigma$, and if
             $r_1^\sigma = r_2^\sigma$ check that $m_1 \neq m_2$
7          Define the labelling function of $S$ as
               $l_S(x) = \begin{cases} l_{L_1^\sigma}(x') & \text{if } \exists x' \in L_1^\sigma \text{ such that } m_1(x') = x \\ l_{L_2^\sigma}(x') & \text{if } \exists x' \in L_2^\sigma \text{ such that } m_2(x') = x \end{cases}$
8          Construct the (critical) pair of derivations $T_1 \overset{r_1,m_1,\sigma}{\Longleftarrow} S \overset{r_2,m_2,\sigma}{\Longrightarrow} T_2$.
9      end
10 end
```

Algorithm 2: Construction of Symbolic Critical Pairs.

The construction is given as the following theorem. We show that the construction produces exactly all critical pairs according to Definition 4.2.

Theorem 4.1 (Construction Correctness). *Given left-linear rule schemata $r_1 = \langle L_1 \leftarrow K_1 \rightarrow R_1 \rangle$ and $r_2 = \langle L_2 \leftarrow K_2 \rightarrow R_2 \rangle$, the construction of algorithm 2 computes all symbolic critical pairs of r_1 and r_2 up to isomorphism.*

Proof. The construction computes only symbolic critical pairs (according to Definition 4.2) - when line 8 is reached, the pair of derivation exists, is minimal and in conflict, and is labelled using one of the substitutions returned by UNIF. The construction computes all critical pairs since all overlaps and all unifiers per overlap are considered. □

The construction of symbolic critical pairs is similar to that of conventional critical pairs described in subsection 2.3.4: the left-hand graphs of the two schemata are overlapped while ignoring labels, the labels of overlapped items are checked for equality, and then certain syntactic properties are checked. The most important difference occurs when overlapping graph nodes or edges since unification needs to be considered. We use our unification algorithm UNIF which, given a system of label equations, terminates with a finite set of unifiers due to the sufficient label restriction of left-linearity, thus making the construction also terminate with a finite set of critical pairs. Recall that the number of conventional critical pairs is

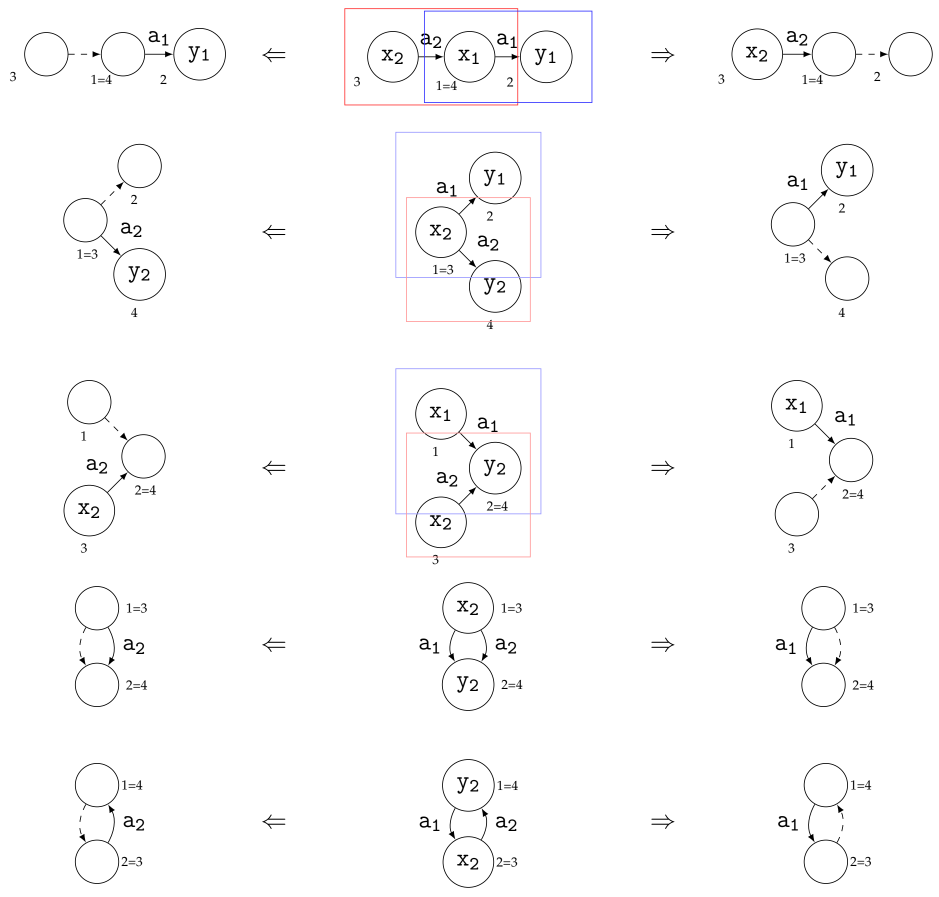


Figure 4.5: The critical pairs of `unlabel` with itself (except for S_2).

infinite in general due to the infinite number of rule instances that a schema represents. Consequently, the computation of symbolic critical pairs is more suitable for automation as part of a confluence checker.

Example 14 (Construction of `unlabel`'s critical pairs.). All overlaps of `unlabel` with itself (when ignoring labels) were shown in Example 12, which we would get after line 1 of the construction algorithm. Since the schema doesn't delete anything, the dangling condition check that follows passes for all overlaps. However, the check for conflicts rules out S_1 as a critical pair since there are no common items, i.e. the steps would be independent. The algorithm then computes the sets of unifiers for each induced system of equations. It is not difficult to see these unifiers only involve variable renaming, e.g. the labelling function l_{S_3} assigns the overlapped node $(1 = 3)$ the label $\mathtt{x}_1$, and the other nodes/edges with labels of their preimages in L_1 and L_2. The check on line 6 rules out the overlap S_8 since the matches are equal.

All resulting symbolic critical pairs are shown in Figure 4.5. The symbolic critical pair based on S_2 was shown in Figure 4.4.

□

The fact that the set of critical pairs is finite is a direct consequence of our construction algorithm. Formally, we state this as the following fact.

Corollary 1 (Finiteness of Symbolic Critical Pairs). *For each pair of left-linear rule schemata r_1 and r_2, the set of symbolic critical pairs induced by r_1 and r_2 is finite.*

Proof. Since the Theorem 4.1 construction computes exactly all critical pairs and terminates, then the set of symbolic critical pairs must be finite. □

4.4 Completeness of Symbolic Critical Pairs

In this section, we show that our critical pairs are complete, i.e. they represent all host graph conflicts. Intuitively, representation means that each conflict is an embedding of a critical pair *instance* into larger context — by means of an instantiation combined with an embedding morphism.

Our notion of representation is more involved than standard graph transformation (e.g. [EEPT06]), where it is enough to consider an embedding morphism between a host graph conflict and its critical pair. Instead, we consider the representation to consist of an instantiation and a graph morphism as components. This is very similar to the approach taken in the framework of attributed graph transformation where the match morphisms already contain a label instantiation and graph morphism as components.

The completeness proof we give is concerned with the properties of the class of partially labelled graphs $\mathcal{G}_\perp$ and the classes of horizontal and vertical morphisms in direct derivations, namely the class of injective label preserving morphisms $\mathcal{M}$ and the class of injective label and undefinedness preserving morphisms $\mathcal{N}$. Their basic properties have already been studied in [HP12] and [HP02]. We list these properties in Appendix A and refer to them in proofs as Fact A.

The proof also relies on a restriction lemma, the idea being that an arbitrary host graph conflict can be restricted to a minimal one, roughly corresponding to a conventional critical pair. The lemma is formulated only for direct derivations as subsequently we only require it for restricting conflicts which are pairs of direct derivations. Restriction is in some sense the inverse of extending a derivation to a larger context.

Lemma 6 (Restriction). *Given a direct derivation $G \overset{r,m,\alpha}{\Longrightarrow} H$, a morphism $e : P \to G \in \mathcal{N}$, and a match $m' : L^\alpha \to P \in \mathcal{N}$ such that $m = e \circ m'$, then there is a direct derivation $P \overset{r,m',\alpha}{\Longrightarrow} Q$ leading to the (extension) diagram below.*

$$\begin{tikzcd}
L^{\alpha} \arrow[d, "m'"] \arrow[dd, bend right, "m"'] \arrow[dr, phantom, "(2)"] & K^{\alpha} \arrow[l] \arrow[r] \arrow[d] \arrow[dd, bend left] \arrow[dr, phantom, "(3)"] & R^{\alpha} \arrow[d] \arrow[dd, bend left] \\
P \arrow[d, "e"] \arrow[dr, phantom, "(1)"] & N \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(4)"] & Q \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Proof. We start by constructing the pullback (1) of $(D \to G \leftarrow P)$, inducing a morphism $K^\alpha \to N$ such that the diagram (1+2) commutes (Fact A.1.2). Since $D \to G \in \mathcal{M}$ then (1) is an $\mathcal{M}$-pullback. From $P \to G \in \mathcal{N}$ we get that $N \to D \in \mathcal{N}$ due to stability of $\mathcal{N}$ along $\mathcal{M}$-pullbacks (Fact A.1.3). Due to commutativity of $(K^\alpha ND)$ and $K^\alpha \to D, N \to D \in \mathcal{N}$ we get $K^\alpha \to N \in \mathcal{N}$ by decomposition property of $\mathcal{N}$ (Fact A.1.1). By applying the $\mathcal{M},\mathcal{N}$-pushout-pullback decomposition property (Fact A.2.2), we get that squares (1) and (2) are NPOs. The argument that the property can be applied is as follows: (a) square (1+2) is NPO by definition, (b) square (1) is a pullback by construction, (c) $K^\alpha \to L^\alpha \in \mathcal{M}$ by definition, (d) $P \to G \in \mathcal{N}$ by construction, (e) $K^\alpha \to N \in \mathcal{N}$ was shown.

We then construct the pushout (3) over $K^\alpha \to R^\alpha \in \mathcal{M}$ and $K^\alpha \to N \in \mathcal{N}$. This construction is possible because all $\mathcal{M},\mathcal{N}$-pushouts exist (Fact A.1.2). We obtain the induced morphism $Q \to H$ such that the diagram (3+4) commutes. By pushout decomposition in any category, the square (4) is a pushout. We also show it is a pullback. The argument needed is that $\mathcal{M},\mathcal{N}$-pushouts are also pullbacks (Fact A.2.1). The morphism $N \to Q \in \mathcal{M}$ due to the following reasons - (3) is an $\mathcal{M},\mathcal{N}$-pushout by construction, $K^\alpha \to R^\alpha \in \mathcal{M}$ by definition and $\mathcal{M}$ is stable under $\mathcal{M},\mathcal{N}$-pushouts (Fact A.1.3). We have already shown that $N \to D \in \mathcal{N}$. Therefore (4) is an $\mathcal{M},\mathcal{N}$-pushout and NPO. □

The main theorem — that our critical pairs are complete — follows from this lemma, the results about morphisms in our category, and the completeness result of our unification algorithm (Chapter 5). The first part of the proof is concerned with proving that each pair of conflicting derivations can be restricted to a minimal conflicting pair of derivations on host graphs (e.g. see the proof of Lemma 6.22 in [EEPT06]). In the second part we show that each conflicting and minimal pair of host graph derivations is an instance of a symbolic critical pair. In this second part we use the fact that our unification algorithm is complete assuming left-linearity.

Theorem 4.2 (Completeness of Symbolic Critical Pairs). *For each pair of conflicting rule schema applications $H_1 \overset{r_1,m_1,\alpha}{\Longleftarrow} G \overset{r_2,m_2,\alpha}{\Longrightarrow} H_2$ between left-linear schemata r_1 and r_2 there exists a symbolic critical pair $T_1 \overset{r_1}{\Leftarrow} S \overset{r_2}{\Rightarrow} T_2$ with the (extension) diagrams between $H_1 \Leftarrow G \Rightarrow H_2$ and the critical pair's instance.*

$$\begin{tikzcd}
T_1 \arrow[d, mapsto] & S \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d, mapsto] & T_2 \arrow[d, mapsto] \\
Q_1 \arrow[d] & P \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d] & Q_2 \arrow[d] \\
H_1 & G \arrow[l, Rightarrow] \arrow[r, Rightarrow] & H_2
\end{tikzcd}$$

Proof. We start by decomposing the pair of matches $(m_1 : L_1^\alpha \to G, m_2 : L_2^\alpha \to G)$ (Figure 4.6) to obtain a graph $P = m_1(L_1^\alpha) \cup m_2(L_2^\alpha) = m_1'(L_1^\alpha) \cup m_2'(L_2^\alpha)$ together with jointly surjective matches $(m_1' : L_1^\alpha \to P, m_2' : L_2^\alpha \to P)$ and morphism $e : P \to G \in \mathcal{N}$ (Fact A.3.3) such that $m_1 = e \circ m_1'$, $m_2 = e \circ m_2'$, and $m_1', m_2' \in \mathcal{N}$ (Fact A.1.1).

$$\begin{tikzcd}
R_1^\alpha \arrow[d] \arrow[dd, bend right] \arrow[dr, phantom, "(3)"] & K_1^\alpha \arrow[l] \arrow[r] \arrow[d] \arrow[dd, bend left] \arrow[dr, phantom, "(2)"] & L_1^\alpha \arrow[dr, "m_1'"] \arrow[ddr, bend right, "m_1"'] & & L_2^\alpha \arrow[dl, "m_2'"'] \arrow[ddl, bend left, "m_2"] \arrow[dr, phantom, "(6)"] & K_2^\alpha \arrow[l] \arrow[r] \arrow[d] \arrow[dd, bend right] \arrow[dr, phantom, "(7)"] & R_2^\alpha \arrow[d] \arrow[dd, bend left] \\
Q_1 \arrow[d] \arrow[dr, phantom, "(4)"] & N_1 \arrow[l] \arrow[rr] \arrow[d] \arrow[drr, phantom, "(1)"] & & P \arrow[d, "e"] \arrow[drr, phantom, "(5)"] & & N_2 \arrow[ll] \arrow[r] \arrow[d] \arrow[dr, phantom, "(8)"] & Q_2 \arrow[d] \\
H_1 & D_1 \arrow[l] \arrow[rr] & & G & & D_2 \arrow[ll] \arrow[r] & H_2
\end{tikzcd}$$

Figure 4.6: Decomposed pushouts.

Next we apply Lemma 6 twice to obtain the restricted derivations $P \Rightarrow Q_1$ and $P \Rightarrow Q_2$. It is not difficult to show that $Q_1 \Leftarrow P \Rightarrow Q_2$ is minimal and in conflict using the commutativity of (1), the properties of (m_1', m_2'), and Definition 3.1. We know that (m_1', m_2') are jointly surjective by construction. Assume that the derivations are independent. This means there exist morphisms $L_1^\alpha \to N_2$, $L_2^\alpha \to N_1$ such that $L_1^\alpha \to N_2 \to P = L_1^\alpha \to P$ and $L_2^\alpha \to N_1 \to P = L_2^\alpha \to P$. We know that $m_1 = e \circ m_1'$ by construction. We get $L_2^\alpha \to G = L_2^\alpha \to P \to G = L_2^\alpha \to N_1 \to P \to G$. We have that square (1) commutes: $N_1 \to P \to G = N_1 \to D_1 \to G$. Therefore $L_2^\alpha \to G = L_2^\alpha \to N_1 \to D_1 \to G$, meaning there exists a morphism $L_2^\alpha \to D_1$ with the required independence property in Definition 3.1. Similarly we obtain morphism $L_1^\alpha \to D_2$. This means that $H_1 \Leftarrow G \Rightarrow H_2$ is independent, a contradiction. Therefore, $Q_1 \overset{r_1, m_1', \alpha}{\Longleftarrow} P \overset{r_2, m_2', \alpha}{\Longrightarrow} Q_2$ is a conflicting and minimal pair of derivations. This concludes the first part of the proof.

For the second part we will show that $Q_1 \Leftarrow P \Rightarrow Q_2$ is an instance of a symbolic critical pair $T_1 \Leftarrow S \Rightarrow T_2$. The graphs have the same node/edge sets as $Q_1 \Leftarrow P \Rightarrow Q_2$ but different labels. We use the fact that the assignment α is an AU-unifier for the system of equations $\mathrm{EQ}(L_1, L_2, m_1, m_2)$ and therefore, by the completeness of the generalized unification algorithm (Theorem 5.4) (r_1 and r_2 are left-linear), α is an instance of a unifier $\sigma \in \mathrm{UNIF}(\mathrm{EQ}(L_1, L_2, m_1', m_2'))$ such that $\alpha = \lambda \circ \sigma$ where λ is some assignment.

First, instantiate L_1 and L_2 using σ to obtain graphs L_1^σ and L_2^σ. Then define $S = m_1'(L_1^\sigma) \cup m_2'(L_2^\sigma)$. This definition is sound because σ is a unifier. It is easy to show that $P \cong S^\lambda$ using $\alpha = \lambda \circ \sigma$ — we have that $S^\lambda = m_1'(L_1^{\lambda \circ \sigma}) \cup m_2'(L_2^{\lambda \circ \sigma}) = m_1'(L_1^\alpha) \cup m_2'(L_2^\alpha)$, but also $P = m_1'(L_1^\alpha) \cup m_2'(L_2^\alpha)$ by construction.

We proceed by constructing the derivation $S \overset{r_1, m_1', \sigma}{\Longrightarrow} T_1$ - the double-pushout is (9+10) of Figure 4.7 together with the instantiation squares right above it. The morphism m_1' induces a morphism $L_1^\sigma \to S \in \mathcal{N}$. We have that $K^\sigma \to L_1^\sigma \in \mathcal{M}$. By uniqueness of $\mathcal{M}, \mathcal{N}$-pushout complements (Fact A.2.3), we get a graph O_1 such that square (9) is a $\mathcal{M}, \mathcal{N}$-pushout (and NPO) together with morphism $K_1^\sigma \to O_1 \in \mathcal{N}$. The complement exists because m_1' satisfies the dangling condition w.r.t. r_1^α and since the dangling condition is not concerned with labels, $L_1^\sigma \to S$

$$\begin{tikzcd}
R_1 \arrow[dd, mapsto, "\sigma"] & K_1 \arrow[l] \arrow[r] \arrow[dd, mapsto, "\sigma"] & L_1 \arrow[dr, mapsto, "\alpha"] \arrow[dd, mapsto, "\sigma"] & \\
 & & & L_1^{\alpha} \arrow[ddd, bend left, "m'_1"] \\
R_1^{\sigma} \arrow[d] \arrow[r, phantom, "(10)" below] & K_1^{\sigma} \arrow[l] \arrow[r] \arrow[d] \arrow[r, phantom, "(9)" below] & L_1^{\sigma} \arrow[ur, mapsto, "\lambda"] \arrow[dr] & \\
T_1 \arrow[d, mapsto, "\lambda"] & O_1 \arrow[l] \arrow[rr] \arrow[d, mapsto, "\lambda"] & & S \arrow[d, mapsto, "\lambda"] \\
Q_1 & N_1 \arrow[l] \arrow[rr] & & S^{\lambda} \cong P
\end{tikzcd}$$

Figure 4.7: Construction of $S \Rightarrow T_1$.

satisfies the dangling condition w.r.t. r_1^{σ}. Therefore we can also construct the $\mathcal{M},\mathcal{N}$-pushout (10) of $R_1^{\sigma} \leftarrow Q_1^{\sigma} \rightarrow O_1$ to obtain the direct derivation $S \overset{r_1,m'_1,\sigma}{\Longrightarrow} T_1$. The same construction is applied to obtain $S \Rightarrow T_2$.

Finally, we show that $T_1 \overset{r_1,m'_1,\sigma}{\Longleftarrow} S \overset{r_2,m'_2,\sigma}{\Longrightarrow} T_2$ is a symbolic critical pair by using Definition 4.2. We have that (m'_1, m'_2) are jointly surjective and σ is a unifier. It remains to be shown that $T_1 \Leftarrow S \Rightarrow T_2$ are in conflict. Assume they are not, i.e. there exist morphisms $L_1^{\sigma} \rightarrow O_2$ and $L_2^{\sigma} \rightarrow O_1$ such that $L_1^{\sigma} \rightarrow O_2 \rightarrow S = m'_1$ and $L_2^{\sigma} \rightarrow O_1 \rightarrow S = m'_2$. Since the graphs $L_1, L_1^{\alpha}, L_1^{\sigma}$ and $L_2, L_2^{\alpha}, L_2^{\sigma}$ have the same node/edge sets, the above morphisms induce morphisms $L_1^{\alpha} \rightarrow N_2$ and $L_2^{\alpha} \rightarrow N_1$ such that the independence conditions hold. This contradicts the fact that $Q_1 \Leftarrow P \Rightarrow Q_2$ is in conflict. Therefore $T_1 \overset{r_1,m'_1,\sigma}{\Longleftarrow} S \overset{r_2,m'_2,\sigma}{\Longrightarrow} T_2$ is a symbolic critical pair.

□

Example 15 (Completeness diagram for `unlabel`). Consider the pairs of derivations in Figure 4.1 and Figure 4.4. They form the layers of the diagram in Theorem 4.2. The instantiation $\lambda = \{\mathtt{x1} \rightarrow 1, \mathtt{y1} \rightarrow 2, \mathtt{y2} \rightarrow 3, \mathtt{a1} \rightarrow 4, \mathtt{a2} \rightarrow 5\}$ 'links' the symbolic critical pair to the concrete one.

4.5 Related Work

In this section we provide some pointers to related work on critical pairs. First, we provide some historical references for critical pairs and confluence. Following this we shall point to some recent parallel approaches extending the basic notions of confluence analysis.

Theory. The concept of critical pairs was first developed in the area of term rewriting [KB70], and later carried over to term graph rewriting [Plu94] and hypergraph rewriting [Plu93]. The approach has been extended to various flavours of graph transformation systems, including graph transformation with negative application conditions [LEO06], adhesive high-level replacement systems [EHPP04], typed attributed graph transformation [HKT02, EPT04, GLEO12], $\mathcal{M}$-adhesive transformation systems with nested application conditions [EGH$^+$12], SPO rewriting with negative application conditions [KMP05], layered graph transformation [BTS00].

In the context of graph transformation, the undecidability of confluence was first shown for hyper-graph rewriting in [Plu93] and later strengthened for DPO graph transformation in [Plu05]. Confluence becomes decidable when all critical pairs satisfy a special condition called *coverability* [Plu10], where a cover is a special structure that essentially plays the role of non-deletable context attached to persistent nodes. More recently, it has been shown that (non-ground) confluence of DPO graph transformation with interfaces is decidable [BGK$^+$17], where rather than rewriting graphs one rewrites graph morphisms.

The paper [LE06] gives a more elaborate construction algorithm for the plain graph transformation case. The idea is to statically analyse the rules to determine whether (1) they can cause a conflict at all, e.g. a pair of non-modifying rules [1] cannot have conflicts; and (2) their overlap contains at least one modified item. This is in contrast to constructing all possible overlaps of graphs and then checking whether the derivations exist/conflict with each other. It remains an open question whether this more involved construction leads to a significant optimisation. The paper [LEO06] extends this critical pair construction algorithm to handle negative application conditions (NACs).

Practice. Based on the rich theory of graph transformation, many tools have been developed over recent years. However, only a few of them provide support for conflict analysis.

The tool AGG [RET11] implements critical pair construction for the case of typed attributed graph transformation. It supports several features like negative application conditions, re-attribution, type graphs, layered transformations. In this setting, graph attributes are represented by means of special *data* nodes and linked to ordinary graph nodes/edges by attribution edges. This gives rise to infinite graphs as, for example, all natural numbers will exist as separate data nodes. Attributed rules contain a data node for each term in the term algebra $T(X)$. The critical pair construction however is restricted to rules whose attributes are variables or variable-free. An earlier version of the construction was based on computing a most general unifier of overlapped attributes [HKT02], which renders the critical pairs incomplete (shown in [EEPT06, p. 198]). Attribute algebras are treated as parameters.

Henshin [ABJ$^+$10, SBG$^+$17] is a tool for model transformation that has useful support for several application areas such as model refactoring, pattern introduction, model evolution. The tool is based on concepts from attributed graph transformation, offers a transformation language with formal semantics, and has rich

[1]A rule $\langle L \leftarrow K \rightarrow R \rangle$ is non-modifying if the inclusion $L \leftarrow K$ is an isomorphism.

support for attribute manipulation and application conditions. Henshin supports DPO rules with negative / positive application conditions, attribution using either literals or JavaScript expressions, injective / non-injective matching, rule orchestration, rule parameters (schemata). The critical pair module [BAHT15] (implementation available in Henshin v1.4) enables the detection of all potential conflicts and dependencies of a set of Henshin rules. The implementation itself relies entirely on AGG. Note the paper [BAHT15] does not say how the above rich rules are translated and/or restricted to account for e.g. undecidability.

SyGrAV [Dec17, DKL+16, KDL+15] is a tool prototype aimed at formal verification of symbolic graph transformations. The framework handles node-attributed symbolic graphs combined with first-order logic formulas, with the intention that graphs are attributed by variables only and whose values are constrained by the formula. The tool is based on the theory of projective graph transformation systems over a fixed (but unspecified) data algebra. The critical pair component performs construction of critical pairs for symbolic rules and also sub-commutativity analysis (0-1 step joinability). The tool supports attributes that are integers, reals, bitvectors (with finite domains) using the SMT-solver Z3 [dMB08], but without support for strings in labels nor (non-nested) application conditions nor control constructs. Since symbolic graph matching is undecidable in general, Z3 may be unable to decide the validity of a formula during analysis. In those cases, the conflict is not discarded, ensuring soundness of the approach. The tool also uses a pattern matching component called Democles.

The approach of [Dec17] has several similarities and some aspects are handled differently. For example, graphs conditions are handled by the theoretical framework (by using logic conjunction) whereas we do not handle conditions formally. The actual restrictions on graph conditions that can be handled come from the limited capabilities of Z3. Another difference are the kind of rules allowed — we place restrictions in order to avoid an infinite number of critical pairs, which actually stem from the nature of the problem, whereas [Dec17] handles attributes as a parameter (in the theoretical framework) and by relying again on Z3 to handle undecidability at run-time. Application conditions during confluence analysis are not supported. Similarly, control structures over rules are not formally treated, but also the practical example (a campus management system) has no such control flow and only a set of rules. A refreshing idea is the filtering of critical pairs who violate the normal constraints of the system, as such critical pairs can be discarded without violating the soundness of the approach. We exploit the same notion in Section 7.2 (with the critical pair named SD6).

4.6 Summary

In this chapter we have:

- extended the conventional notion of critical pairs to the setting of GP 2 to allow for reasoning about the confluence of sets of unconditional rule schemata;

- defined symbolic critical pairs that are minimal conflicts at the level of schemata;
- given a construction algorithm that involves a special-purpose *E*-unification algorithm for GP 2 list expressions;
- shown that the set of such critical pairs is finite and complete under the restriction of *left-linearity*;
- compared our approach with existing confluence analysis techniques in other approaches.

Chapter 5

Unification of GP 2 labels

In Chapter 4, we introduced critical pairs for confluence analysis of sets of schemata, and gave an algorithm for their construction by relying on a terminating algorithm for computing their labels. We also showed that the set of critical pairs to be finite, and that it represents all possible host graph conflicts. These properties rely on the properties of the said algorithm.

The next step we take in this thesis is to introduce our rule-based unification algorithm for solving systems of label equations. We show that the algorithm terminates, and is sound meaning that each generated substitution is a unifier of the input system. Moreover, the algorithm is complete meaning that every unifier of the input system of equations is an instance of some unifier in the computed set of solutions. Finally, we review related work on unification in critical pair construction.

This chapter is based on our work in [HP15] which gives the unification algorithm and proves termination and soundness. The completeness proof only appeared in the long version of that paper [HP17b].

5.1 Solving Label Equations with Unification

Unification is an algorithmic process for solving equations between expressions, and has a long tradition in automated reasoning [BS01]. The central idea of unification is that of finding a *unifier*: given an equation between two expressions s and t, find a unifier σ (a substitution) such that $\sigma(s) \approx \sigma(t)$. The choice of equivalence $\approx$ is important [1]: under syntactic equality $=$, the process is called *syntactic* unification; if one is given a set of identities E and thus considers equivalence *modulo* E ($=_E$), then the process is called *equational* unification (E-unification for short). (Logically, E is a set of identities that are universally quantified, whereas the given equation(s) is existentially quantified.)

To be more concrete, consider the unification problem $P = \langle \mathtt{x:1} =^? \mathtt{y:1} \rangle$ where $\mathtt{x,y}$ are list variables. To solve P, one has to replace the variables $\mathtt{x,y}$ by expressions such that the resulting expressions are equivalent. An example unifier

[1]Some would even say crucial.

is $\sigma = \{\mathtt{x} \to \mathtt{1}, \mathtt{y} \to \mathtt{1}\}$, and its application to the expressions is $\sigma(\mathtt{x} : \mathtt{1}) = \mathtt{1} : \mathtt{1} = \sigma(\mathtt{y} : \mathtt{1})$. Actually, the problem has an infinite number of such unifiers (in the domain of integers). In syntactic unification however, it is sufficient to consider the *most general* unifier, i.e. a unifier such that every other unifier can be obtained by instantiation. In the given example, this is $\sigma = \{\mathtt{x} \to \mathtt{y}\}$ since for all expressions $\mathtt{e}$ we have $\{\mathtt{x} \to \mathtt{e}, \mathtt{y} \to \mathtt{e}\} = \sigma\{\mathtt{y} \to \mathtt{e}\}$. Computation of a most general unifier (if it exists) is nearly linear in the size of the expressions.

Even though syntactic unification has been widely studied and applied in many areas of reasoning, it is inadequate for the computation of critical pairs. As already noted in Chapter 2, using only the most general unifier construction for label computation leads to an *incomplete* set of critical pairs, and thus an unsound analysis. Instead, one has to take into account the axioms valid in the label algebra. In our setting, we consider solving equations in the theory of list concatenation ':' that is associative and has a unit element which is referred to as *word unification* in the literature. Here the notion of a most general unifier can be extended to a complete set of unifiers. Solving unification modulo associativity is decidable, albeit in PSPACE [Pla99], and the set of solutions is infinite in general.

A related problem is to also generate the set of *all* unifiers for a given problem. This problem becomes more complex when the generated set must be *complete*, i.e. every solution is an instance of a generated solution, and *minimal* meaning that every generated solution is unique. In the above example, the substitution $\{\mathtt{x} \to \mathtt{y}\}$ forms a minimal complete set of unifiers. However, for applications of unification, one does not need the set of all unifiers, but rather a complete set from which all unifiers can be generated.

Constructing and using sets of unifiers is necessary in our setting because to construct critical pairs we have to overlap graphs labelled with expressions. In a sense, the process of unification determines the labels of critical pairs in such a way that we can prove the resulting critical pairs are suitable to be used in a confluence checker. This is in contrast to e.g. placing severe restrictions on the syntax of labels that appear in rule schemata which avoids the need for unification altogether, as in the approach to attributed graph transformation [EPT04].

Constructing and using sets of unifiers is also appealing because it allows us to solve systems of label equations that arise during construction of complete sets of critical pairs. The label unification approach facilitates separation of concerns when it comes to computing critical pairs and allows for the isolation of difficult aspects of the associated proofs, whereas a uniform treatment of label computation within the algebraic theory of graph transformation is cumbersome and makes proofs difficult to establish.

We present our rule-based unification algorithm that solves systems of label equations that copes with the equational axioms of GP 2 list concatenation. The algorithm is essentially a tree generation process where nodes are labelled with unification problems and edges represent applications of unification rules. The input label equation is the root of the tree, and its solutions are leaves of the tree. The algorithm produces a complete set of unifiers, but it is not minimal. Furthermore, the algorithm deals with GP 2's type system.

To be useful, the unification algorithm needs to have several properties. First, it should be *terminating* and *sound*. That is, given a system of equations, it returns a finite set of unifiers. Proving the correctness of such an algorithm is typically done by induction on the unification rules, and its termination by defining a lexicographic ordering that reduces during each application step, thus proving the unification tree has bounded depth. However, to avoid non-termination due to the nature of unification in our setting, we place a restriction of *left-linearity* which allows us to consider individual equations and combine their solutions rather than having to consider systems of interdependent equations.

Another necessary property is *completeness*. As explained above, the computed set of unifiers should represent all possible unifiers. The proof of this property is rather complicated. It involves a separate algorithm that, given a unification problem *and a unifier*, produces a branch in the unification tree associated with a more general unifier. The intuition here is that the computation of such branches proves a correspondence between the leaves of the unification tree and the desired complete set of unifiers.

An interesting property is that of *minimality*. Minimality essentially means that each pair of computed unifiers are incomparable, i.e. not instances of each other. For unification problems where the complete sets of unifiers are finite, it is not necessary to always return a minimal set of solutions, since dependent unifiers can be effectively compared and removed. In this case, computational efficiency becomes an important factor. However, minimality is in general much more difficult to achieve than correctness and completeness. We give a sufficient restriction that ensures finiteness of the complete set of unifiers for a given system of equations, and thus minimality concerns are avoided.

5.2 Preliminaries

We begin the exposition by giving some preliminary background on unification. This is necessary to understand our unification algorithm and the formal proofs of its properties. For general introduction and further reading on (E-)unification, see [BS01] [BS94].

Informally, substitutions allow us to map variables to expressions. The notion is required in the definition of critical pairs (labelled with expressions), and in the construction of a complete set of unifiers. Substitutions can be seen as generalization of assignments. The notion of application can thus remain the same — in label expressions, simply perform in-place replacement of variables with their corresponding expressions. However, for the purposes of unification, we will also need the ability to *compose* substitutions. This is due to the fact that our unification algorithm is rule-based and thus needs to record partial solutions.

Definition 5.1 (Substitution; Application; Composition)**.** A *substitution* is a family of mappings $\sigma = (\sigma_X)_{X \in \{I,S,A,L\}}$ where $\sigma_I \colon \mathrm{IVar} \to \mathrm{Integer}$, $\sigma_S \colon \mathrm{SVar} \to \mathrm{String}$, $\sigma_A \colon \mathrm{AVar} \to \mathrm{Atom}$, $\sigma_L \colon \mathrm{LVar} \to \mathrm{List}$. Here Integer, String, Atom and List are the sets of expressions defined by the GP label grammar of Figure 2.8. By $\mathrm{Dom}(\sigma)$ we denote the set $\{x \in \mathrm{Var} \mid \sigma(x) \neq x\}$ and by $\mathrm{VRan}(\sigma)$ the set of variables occurring in the expressions $\{\sigma(x) \mid x \in \mathrm{Var}\}$.

Applying a substitution σ to an expression t, denoted by $\sigma(t)$ (sometimes $t\sigma$), means to replace every variable x in t by $\sigma(x)$ simultaneously. Composition of substitutions λ and σ is written as $\lambda \circ \sigma$ (λ *after* σ). Given substitutions $\sigma_1, \ldots, \sigma_n$ with pairwise disjoint domains, their composition $\sigma_1 \circ \ldots \circ \sigma_n$ is commutative. □

It is useful to note that substitutions, like assignments, are *well-typed* in that variables can only be mapped to expressions of less or equal type. Furthermore, substitutions can be extended to mappings between terms in the usual way. To simplify notation, we usually do not distinguish between a substitution and its extension.

Example 16 (Substitution; Application; Composition). Consider the variables $\mathtt{x} \in$ LVar, $\mathtt{n} \in$ IVar and $\mathtt{s} \in$ SVar. We write $\sigma = \{\mathtt{n} \mapsto \mathtt{n}+1, \mathtt{x} \mapsto \mathtt{s} : -\mathtt{n} : \mathtt{s}\}$ for the substitution that maps $\mathtt{n}$ to $\mathtt{n}+1$, $\mathtt{x}$ to $\mathtt{s} : -\mathtt{n} : \mathtt{s}$ and every other variable to itself. The application of the substitution to the expression $\mathtt{x} : -\mathtt{n}$ is $\sigma(\mathtt{x} : -\mathtt{n}) = \mathtt{s} : -\mathtt{n} : \mathtt{s} : -(\mathtt{n}+1)$.

Composing σ with $\sigma' = \{\mathtt{s} \rightarrow \text{“}a\text{”}\}$ leads to the substitution $\sigma \circ \sigma' = \{\mathtt{n} \mapsto \mathtt{n}+1,\ \mathtt{x} \mapsto \text{“a”} : -\mathtt{n} : \text{“a”},\ \mathtt{s} \mapsto \text{“}a\text{”}\}$. Note that during composition we have applied σ' to the existing expressions of σ. □

Definition 5.2 (Unification problem)**.** A *unification problem* is a pair of an equation and a substitution

$$P = \langle s =^? t, \theta \rangle$$

where s and t are simple list expressions without common variables. □

The symbol $=^?$ signifies that the equation must be *solved* rather than having to hold for all values of its variables. The purpose of θ is for the unification algorithm to record a partial solution (Section 5.3). An illustration of this concept will be seen in Figure 5.3. When a problem is solved, we will represent it as the pair $\langle , \theta \rangle$ that has no equation component.

In Chapter 4, we already remarked that rule schemata need to be left-linear in order to produce a finite complete set of unifiers and thus a finite complete set of critical pairs. This restriction will be properly explained in Section 5.6 when we show how the algorithm generalizes to systems of equations. Given this restriction, the problem of solving a system of equations $\{s_1 =^? t_1, s_2 =^? t_2\}$ can be broken down to solving individual equations and combining the answers — if σ_1 and σ_2 are solutions to each individual equation, then $\sigma_1 \circ \sigma_2$ is a solution to the combined problem as σ_1 and σ_2 do not share (list) variables.

As stressed in the beginning of this chapter, the unification algorithm should take into account the axioms valid in the label algebra. This this end, we define unifiers (the solutions returned by our algorithm) to respect the axioms of list concatenations, namely associativity and unity:

$$\mathrm{AU} = \{\mathtt{x} : (\mathtt{y} : \mathtt{z}) = (\mathtt{x} : \mathtt{y}) : \mathtt{z},\ \mathtt{empty} : \mathtt{x} = \mathtt{x},\ \mathtt{x} : \mathtt{empty} = \mathtt{x}\}$$

where `x,y,z` are list variables. Note that brackets are part of the axioms even though the GP 2 syntax omits brackets (exactly because list concatenation is associative). Let the relation $=_{\mathrm{AU}}$ be the equational theory on list expressions induced by these axioms.

Definition 5.3 (Unifier)**.** Given a unification problem $P = \langle s =^? t, \theta \rangle$ a *unifier* of P is a substitution σ such that

$$\sigma(s) =_{\mathrm{AU}} \sigma(t) \text{ and } \sigma(\mathtt{x_i}) =_{\mathrm{AU}} \sigma(\mathtt{e_i})$$

for each binding $\{\mathtt{x_i} \mapsto \mathtt{e_i}\}$ in θ . □

As mentioned, in general unification modulo associativity has an infinite number of unifiers. However, one need not compute all unifiers but only a representative set from which all other unifiers can be obtained by instantiation. For this purpose, we define a quasi-ordering $\leqq$ on unifiers.

Definition 5.4 (More general unifier; Unifier Instance)**.** A substitution σ is *more general* on a set of variables X than a substitution θ if there exists a substitution λ such that $x\theta =_{\mathrm{AU}} x\sigma\lambda$ for all $x \in X$. In this case we write $\sigma \leqq_X \theta$ and say that θ is an instance of σ on X. Substitutions σ and θ are *equivalent* on X, denoted by $\sigma =_X \theta$, if $\sigma \leqq_X \theta$ and $\theta \leqq_X \sigma$. □

If the set of variables X is irrelevant or clear from the context, we will often omit it for clarity.

Remark 5 (Identity Substitution). The identity substitution $id = \{\}$ maps every variable to itself, and for every other substitution θ we have that $id \leqq \theta$ because $id \circ \theta =_{\mathrm{AU}} \theta$. This will be useful later in the proof of completeness of the unification algorithm. □

Definition 5.5 (Set of All Unifiers; Failure; Initial and Solved Problem)**.** The set of all unifiers of a unification problem P is denoted by $\mathcal{U}(P)$. We say that P is *unifiable* if $\mathcal{U}(P) \neq \emptyset$. The special unification problem `fail` represents failure and has no unifiers. A problem $P = \langle s =^? t, id \rangle$ is *initial* and $P = \langle\ , \theta \rangle$ is *solved*.

Note that if a problem $P = \langle\ , \theta \rangle$ is solved, then $\theta \in \mathcal{U}(P)$ by definition. This observation will be useful when we prove the correctness of unification.

Definition 5.6 (Complete set of unifiers [Plo72])**.** A distinguished set $\mathcal{C}$ of substitutions is a *complete set of unifiers* of a unification problem P if:

1. (Soundness) $\mathcal{C} \subseteq \mathcal{U}(P)$, that is, each substitution in $\mathcal{C}$ is a unifier of P, and
2. (Completeness) for each $\theta \in \mathcal{U}(P)$ there exists $\sigma \in \mathcal{C}$ such that $\sigma \leqq_X \theta$, where $X = \mathrm{Var}(P)$ is the set of variables occurring in P.

Set $\mathcal{C}$ is also *minimal* if each pair of distinct unifiers in $\mathcal{C}$ are incomparable with respect to $\leqq_X$.

If a unification problem P is not unifiable, then by convention the empty set $\varnothing$ is a minimal complete set of unifiers of P. For reasons of efficiency, a complete set of unifiers should be as small as possible, minimal at best. When minimality cannot be achieved, the set should be at most finite, then its size is of practical importance. For simplicity, we replace $=^?$ with $=$ in unification problems from now on.

Example 17 (Minimal Complete Set of Unifiers). The minimal complete set of unifiers of the problem $\langle \mathtt{a} : \mathtt{x} = \mathtt{y} : 2 \rangle$ (where a is an atom variable and x,y are list variables) is $\{\sigma_1, \sigma_2\}$ with

$$\sigma_1 = \{\mathtt{a} \mapsto 2, \mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{empty}\} \quad \text{and} \quad \sigma_2 = \{\mathtt{x} \mapsto \mathtt{z} : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{z}\}.$$

We have $\sigma_1(\mathtt{a} : \mathtt{x}) = 2 : \mathtt{empty} =_{\mathrm{AU}} 2 =_{\mathrm{AU}} \mathtt{empty} : 2 = \sigma_1(\mathtt{y} : 2)$ and $\sigma_2(\mathtt{a} : \mathtt{x}) = \mathtt{a} : \mathtt{z} : 2 = \sigma_2(\mathtt{y} : 2)$. Other unifiers such as $\sigma_3 = \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\}$ are instances of σ_2.

5.3 Unification Algorithm

Next, we present our rule-based unification algorithm UNIFY that solves systems of label equations in the presence of list concatenation. The algorithm is essentially a tree generation process where nodes are labelled with unification problems and edges represent applications of unification rules. Due to left linearity, the algorithm can consider each input equation in turn. Each input label equation is the root of a (unification) tree, and its solutions are leaves of that tree.

The algorithm is similar to the A-unification algorithm presented in [Sch92] which also terminates when the input problem has no repeated (list) variables, and is sound and complete. Our approach can be seen as an extension from A-unification to AU-unification, to handle the unit equations, and presented in the rule-based style of [BS01]. Furthermore, the algorithm deals with GP 2's type system. The algorithm produces a complete set of unifiers, but it is not minimal.

Notation. We start with some notational conventions for the rest of this section:

- L, M stand for simple GP 2 expressions given in Definition 2.16,
- $\mathtt{x}, \mathtt{y}, \mathtt{z}$ stand for variables of any type (unless otherwise specified),
- $\mathtt{a}, \mathtt{b}$ stand for
  - (i) simple string or integer expressions, or
  - (ii) string, integer or atom variables

Preprocessing. Given a unification problem $P = \langle s =^? t,\ \theta \rangle$, we rewrite the expressions s and t using the reduction rules

$$L : \texttt{empty} \to L \quad \text{and} \quad \texttt{empty} : L \to L$$

where L ranges over list expressions. These reduction rules are applied exhaustively before any of the transformation rules. For example,

$$\texttt{x : empty : 1 : empty} \to \texttt{x : 1 : empty} \to \texttt{x : 1}.$$

We call this process *normalization*. In addition, the rules are applied to each instance of a unification rule (that is, once the formal parameters have been replaced with actual parameters) before it is applied, and also after each unification rule application.

Unique smallest type. As mentioned, the algorithm deals with GP 2's typing system. This is done by incorporating type checks when applying the unification rules. Informally, they ensure variables can only be assigned expressions of equal or lesser type. Formally, we say that each expression l (after being normalized) has a unique smallest type, denoted by type(l), which can be read off the hierarchy in Figure 2.7. We write type$(l_1) <$ type(l_2) or type$(l_1) \leq$ type(l_2) to compare types according to the subtype hierarchy. If the types of l_1 and l_2 are incomparable, we write type$(l_1) \parallel$ type(l_2).

Unification rules. Figure 5.1 shows the unification rules, the essence of our approach, in an inference system style where each rule consists of a premise and a conclusion.

Remove:	deletes trivial equations
Decomp1:	syntactically equates a list variable with an atom expression or list variable
Decomp1':	syntactically equates an atom variable with an expression of lesser type
Decomp2/2':	assigns a list variable to start with another list variable
Decomp3:	removes identical symbols from the head
Decomp4:	solves an atom variable
Subst1:	solves a variable
Subst2:	assigns `empty` to a list variable
Subst3:	assigns an atom prefix to a list variable
Orient1/2:	moves variables to left-hand side
Orient3:	moves variables of larger type to left-hand side
Orient4:	moves a list variable to the left-hand side

The rules induce a rewrite relation $\Rightarrow$ on unification problems. In order to apply any of the rules to a problem P, the problem part of its premise needs to be

$$\frac{\langle L = L, \sigma\rangle}{\langle\ , \sigma\rangle}\ \textsf{Remove}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{s} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle L = M,\ \sigma \circ \{\mathtt{x} \mapsto \mathtt{s}\}\rangle}\ \textsf{Decomp1}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{a} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{a}) \leq \mathrm{type}(\mathtt{x}) \leq \mathtt{atom}}{\langle (L = M)\{\mathtt{x} \mapsto \mathtt{a}\},\ \sigma \circ \{\mathtt{x} \mapsto \mathtt{a}\}\rangle}\ \textsf{Decomp1}'$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathrm{type}(\mathtt{y}) = \mathtt{list} \quad \mathtt{x}' \text{ is fresh list variable}}{\langle \mathtt{x}' : L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{y} : \mathtt{x}'\}\rangle}\ \textsf{Decomp2}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathrm{type}(\mathtt{y}) = \mathtt{list} \quad \mathtt{y}' \text{ is fresh list variable}}{\langle L = \mathtt{y}' : M, \sigma \circ \{\mathtt{y} \mapsto \mathtt{x} : \mathtt{y}'\}\rangle}\ \textsf{Decomp2}'$$

$$\frac{\langle \mathtt{s} : L = \mathtt{s} : M, \sigma\rangle}{\langle L = M, \sigma\rangle}\ \textsf{Decomp3} \qquad \frac{\langle \mathtt{x} = \mathtt{a} : \mathtt{y}, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) = \mathtt{atom} \quad \mathrm{type}(\mathtt{y}) = \mathtt{list}}{\langle \mathtt{empty} = \mathtt{y}, \sigma \circ \{\mathtt{x} \mapsto \mathtt{a}\}\rangle}\ \textsf{Decomp4}$$

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathtt{x} \notin \mathrm{Var}(L) \quad \mathrm{type}(\mathtt{x}) \geq \mathrm{type}(L)}{\langle\ , \sigma \circ \{\mathtt{x} \mapsto L\}\rangle}\ \textsf{Subst1}$$

$$\frac{\langle \mathtt{x} : L = M, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{empty}\}\rangle}\ \textsf{Subst2}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{a} : M\}, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathtt{x}' \text{ is a fresh list variable} \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle \mathtt{x}' : L = M, \sigma \circ \{\mathtt{x} \mapsto \mathtt{a} : \mathtt{x}'\}\rangle}\ \textsf{Subst3}$$

$$\frac{\langle \mathtt{a} : M = \mathtt{x} : L, \sigma\rangle \quad \mathtt{a} \text{ is not a variable}}{\langle \mathtt{x} : L = \mathtt{a} : M, \sigma\rangle}\ \textsf{Orient1}$$

$$\frac{\langle \mathtt{x} : L = \mathtt{y}, \sigma\rangle \quad L \neq \mathtt{empty} \quad \mathrm{type}(\mathtt{x}) = \mathrm{type}(\mathtt{y})}{\langle \mathtt{y} = \mathtt{x} : L, \sigma\rangle}\ \textsf{Orient2}$$

$$\frac{\langle \mathtt{y} : M = \mathtt{x} : L, \sigma\rangle \quad \mathrm{type}(\mathtt{y}) < \mathrm{type}(\mathtt{x})}{\langle \mathtt{x} : L = \mathtt{y} : M, \sigma\rangle}\ \textsf{Orient3} \qquad \frac{\langle \mathtt{empty} = \mathtt{x} : L, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\langle \mathtt{x} : L = \mathtt{empty}, \sigma\rangle}\ \textsf{Orient4}$$

Figure 5.1: Unification rules.

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathtt{x} \in \mathrm{Var}(L) \quad \mathtt{x} \neq L \quad \mathrm{type}(\mathtt{x}) = \mathtt{list}}{\mathrm{fail}}\ \mathsf{Occur} \qquad \frac{\langle \mathtt{a} : L = \mathtt{empty}, \sigma\rangle}{\mathrm{fail}}\ \mathsf{Clash2}$$

$$\frac{\langle \mathtt{a} : L = \mathtt{b} : M, \sigma\rangle \quad \mathtt{a} \neq \mathtt{b} \quad \mathrm{Var}(\mathtt{a}) = \emptyset = \mathrm{Var}(\mathtt{b})}{\mathrm{fail}}\ \mathsf{Clash1} \qquad \frac{\langle \mathtt{empty} = \mathtt{a} : L, \sigma\rangle}{\mathrm{fail}}\ \mathsf{Clash3}$$

$$\frac{\langle \mathtt{x} = L, \sigma\rangle \quad \mathrm{type}(\mathtt{x}) \parallel \mathrm{type}(L)}{\mathrm{fail}}\ \mathsf{Clash4}$$

Figure 5.2: Failure rules.

matched onto P. Subsequently, the boolean condition of the premise is checked and the rule *instance* is normalized so that its premise is identical to P.

For example, the rule Orient3 can be matched to $P = \langle \mathtt{a} : 2 = \mathtt{m},\ \theta\rangle$ (where a and m are variables of type atom and list, respectively) by setting $\mathtt{y} \mapsto \mathtt{a}$, $\mathtt{x} \mapsto \mathtt{m}$, $M \mapsto 2$, and $L \mapsto \mathtt{empty}$. The rule instance and its normal form are then

$$\frac{\langle \mathtt{a} : 2 = \mathtt{m} : \mathtt{empty}, \theta\rangle}{\langle \mathtt{m} : \mathtt{empty} = \mathtt{a} : 2, \theta\rangle} \quad \text{and} \quad \frac{\langle \mathtt{a} : 2 = \mathtt{m}, \theta\rangle}{\langle \mathtt{m} = \mathtt{a} : 2, \theta\rangle}$$

where the conclusion of the normal form is the result of applying Orient3 to P.

Unification Failure. Showing that a unification problem has no solution can be a lengthy affair because we need to compute all normal forms with respect to $\Rightarrow$. Instead, the rules Occur and Clash1 to Clash4, shown in Figure 5.2, introduce *failure*. Failure cuts off parts of the unification tree for a given problem P. This is because if $P \Rightarrow$ fail, then P has no unifiers and it is not necessary to compute another normal form. Effectively, the failure rules have precedence over the other rules.

The correctness of the failure rules are justified by the following lemmata.

Lemma 7 (Correctness of Occur)**.** *A normalised equation* $\mathtt{x} = L$ *with* $\mathtt{x} \neq L$ *has no solution if* L *is a simple expression,* $\mathtt{x} \in Var(L)$ *and* $\mathrm{type}(\mathtt{x}) = \mathtt{list}$.

Proof. Since $\mathtt{x} \in Var(L)$ and $\mathtt{x} \neq L$, $L = a_1 : a_2 : \ldots : a_n$ consists of several atom expressions with $n \geq 2$ and $\mathtt{x} = a_i$ for some $1 \leq i \leq n$. As L is normalised, none of the expressions a_i are the constant empty. Also, since L is simple, it contains no list variables other than x and x is not repeated. It follows $\sigma(\mathtt{x}) \neq_{\mathrm{AU}} \sigma(L)$ for every substitution σ. □

Lemma 8 (Correctness of Clash1)**.** *An equation* $\mathtt{a} : L = \mathtt{b} : M$ *with* $\mathtt{a} \neq \mathtt{b}$ *has no solution if* a *and* b *are atom expressions without variables.*

Lemma 9 (Correctness of Clash2/3)**.** *Equations of the form* $\mathtt{a} : L = \mathtt{empty}$ *or* $\mathtt{empty} = \mathtt{a} : L$ *have no solution if* a *is an atom expression.*

```
Input : An initial unification problem $P$
output: A complete set of unifiers $C$

initialize $C \leftarrow \emptyset$
initialize empty queue $Q$
normalize $P$
$Q$.enqueue($P$)
while $Q$ is not empty do
    next $\leftarrow$ $Q$.dequeue
    if next is in the form $\langle\ ,\sigma\rangle$ then
        add $\sigma$ to $C$
    else if next $\not\Rightarrow$ fail then
        foreach $P'$ such that next $\Rightarrow P'$ do
            normalize $P'$
            $Q$.enqueue($P'$)
        end
    else
        skip
    end
end
return $C$
```

Algorithm 3: The UNIFY algorithm.

The UNIFY algorithm. The unification algorithm in algorithm 3 starts by normalizing the input equation. It uses a queue of unification problems to explore the unification tree of P w.r.t $\Rightarrow$ in a breadth-first manner. The first step is to enqueue the (normalized) input problem P.

The variable `next` holds the current unification problem (which is also the head of the queue). If the problem `next` is in the form $\langle\ ,\sigma\rangle$, then σ is a unifier of the original problem and is added to the set `U` of solutions. Otherwise, the algorithm constructs all problems P' such that `next` $\Rightarrow P'$ by applying the unification rules shown above. If P' is `fail` signalling failure, then the unification tree below `next` is ignored, otherwise P' gets normalized and enqueued.

Example 18 (Unification Tree). An example unification tree traversed by the algorithm is shown in Figure 5.3. Nodes are labelled with unification problems and edges represent applications of unification rules. The root of the tree is the problem $\langle \mathtt{y} : 2 = \mathtt{a} : \mathtt{x}\rangle$ to which the rules Decomp1, Subst2 and Subst3 can be applied. The three resulting problems form the second level of the tree and are processed in turn. Eventually, the unifiers

$$\begin{aligned}\sigma_1 &= \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto \mathtt{a}\}\\ \sigma_2 &= \{\mathtt{x} \mapsto \mathtt{y}' : 2, \mathtt{y} \mapsto \mathtt{a} : \mathtt{y}'\}\\ \sigma_3 &= \{\mathtt{a} \mapsto 2, \mathtt{x} \mapsto \mathtt{empty}, \mathtt{y} \mapsto \mathtt{empty}\}\end{aligned}$$

are found, which represent a complete set of unifiers of the initial problem. Note that the set is not minimal because σ_1 is an instance of σ_2. □

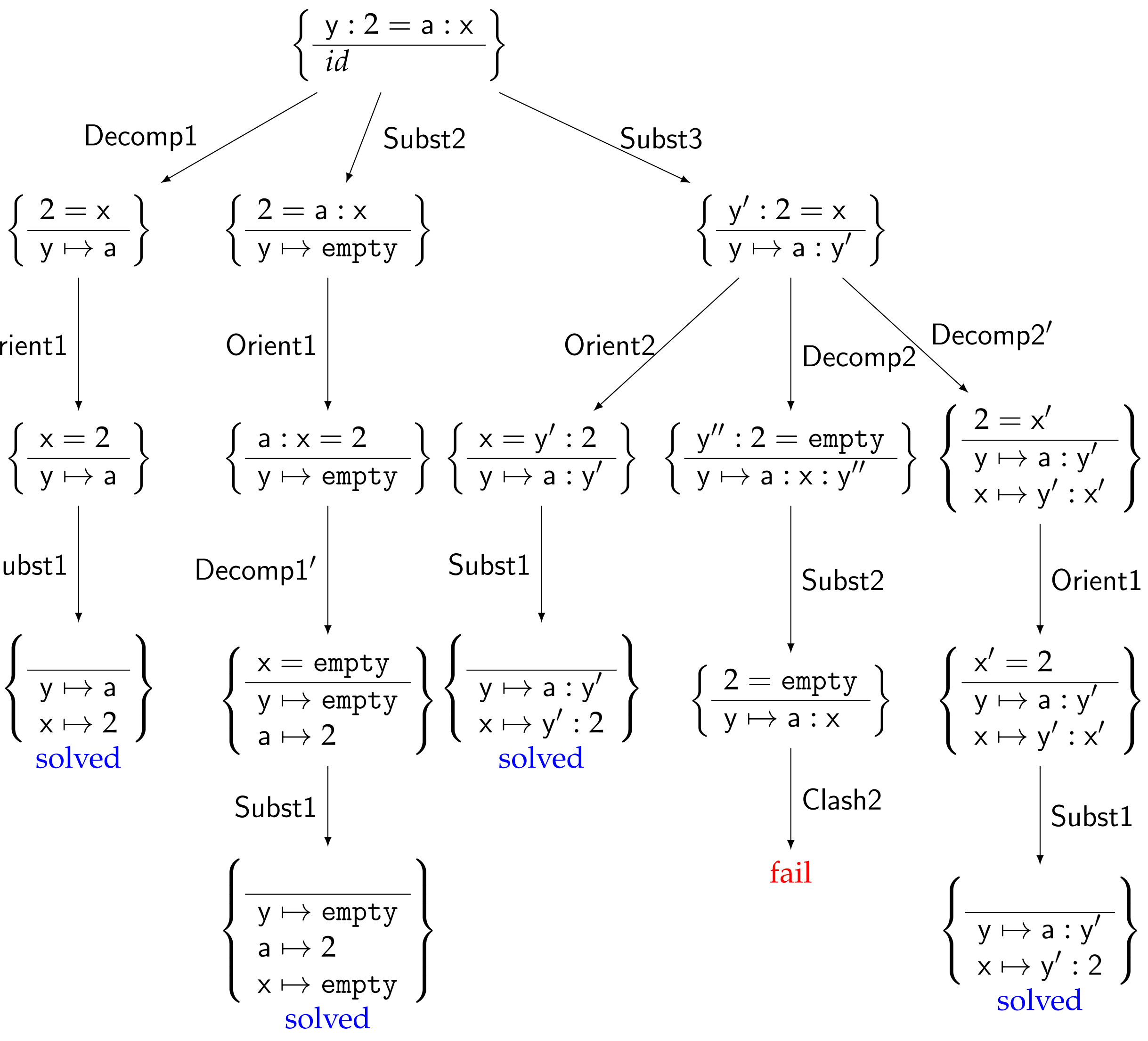


Figure 5.3: Unification tree example.

String concatenation and Left-linearity (again). Since string concatenation is associative and has a unit element (the empty string), the problem of unifying two string expressions is inherently the same as unifying two list expressions. In other words, the relationship `char--string` is the same as `atom--list`. In this thesis, we do not show the unification rules for unifying string expressions as that would obfuscate the algorithm without posing a substantial challenge. With these considerations in mind, one can also restate the left-linearity restriction as "no sharing of list or string variables".

To specify unification between string expressions, one can create variants of the unification rules using a simple renaming strategy — for each rule, the empty list (`empty`) becomes the empty string "", list concatenation ':' becomes string concatenation '.', the `list/atom` types become `string/char` types (used for type comparisons). Some of the unification rules are already general enough not to require such variants, e.g. Subst1, Orient3. Preprocessing would also involve normalization of the empty string.

At the time of writing, we do not have the proofs that these variant rules have the desired properties exhibited by the existing unification rules. We are confident that these properties hold due to the similar nature of the problem. A full completeness proof would also involve an extended selector algorithm (Section 5.5).

5.4 Termination and Soundness

In this section and the next, we consider some important properties of the unification algorithm we have introduced. First, we consider the termination of the algorithm given the condition of left-linearity. Informally, this establishes that the unification tree for a given problem has finite depth. Then, we follow with technical result about soundness of the algorithm.

We show that the unification algorithm terminates if the input problem contains no repeated list variables, where termination of the algorithm follows from termination of the relation $\Rightarrow$ on unification problems. This argument relies on a measure function that maps a unification problem to a tuple of natural numbers and showing each rule decreases w.r.t. the tuple's lexicographic order.

Theorem 5.1 (Termination). *If P is a unification problem without repeated list variables, then there is no infinite sequence $P \Rightarrow P_1 \Rightarrow P_2 \Rightarrow \dots$*

Proof. Define the *size* $|L|$ of an expression L by

- 0 if $L = \texttt{empty}$,
- 1 if L is an expression of category Atom (see Figure 2.8) or a list variable,
- $|M| + |N| + 1$ if $L = M : N$.

We define a lexicographic termination order by assigning to a unification problem $P = \langle L = M, \sigma \rangle$ the tuple (n_1, n_2, n_3, n_4), where

	n_1	n_2	n_3	n_4
Subst1 − 3	$>$			
Decomp1 − 4	$>$			
Decomp1′, 2′	$>$			
Remove	$>$			
Orient1	$=$	$>$		
Orient4	$=$	$>$		
Orient3	$=$	$=$	$>$	
Orient2	$=$	$=$	$=$	$>$

Figure 5.4: Lexicographic termination order.

- n_1 is the size of P, that is, $n_1 = |L| + |M|$;
- $n_2 = \begin{cases} 0 & \text{if } L \text{ starts with a variable} \\ 1 & \text{otherwise} \end{cases}$
- $n_3 = \begin{cases} 1 & \text{if type}(\mathtt{x}) > \text{type}(\mathtt{y}) \\ 0 & \text{otherwise} \end{cases}$

  where x and y are the starting symbols of L and M
- $n_4 = |L|$

The table in Figure 5.4 shows that for each unification step $P \Rightarrow P'$, the tuple associated with P' is strictly smaller than the tuple associated with P in the lexicographic order induced by the components n_1 to n_4.

For most rules, the table entries are easy to check. All the rules except for Orient decrease the size of the input equation $s =^? t$. The Orient rules keep the size equal, but move variables and expressions of a bigger type to the left-hand side. □

In order to prove soundness, we use the following helper lemma.

Lemma 10. *If $P \Rightarrow P'$, then $\mathcal{U}(P) \supseteq \mathcal{U}(P')$.*

Proof. We show that for each unification rule, a unifier θ of the conclusion unifies the premise, i.e. $\theta \in \mathcal{U}(P')$ implies $\theta \in \mathcal{U}(P)$. The substitutions of P and P' are referred to as σ and σ'.

Note that for Remove, Decomp3 and Orient1-4, we have that $\sigma' = \sigma$.

Remove:
$$\begin{aligned} \theta \in \mathcal{U}(\langle\ , \sigma'\rangle) &\iff \theta \in \mathcal{U}(\langle\ , \sigma\rangle) \\ &\iff \theta \in \mathcal{U}(\langle\ , \sigma\rangle) \wedge L\theta = L\theta \\ &\iff \theta \in \mathcal{U}(\langle L = L, \sigma\rangle) \end{aligned}$$

Decomp3:
$$\begin{aligned} \theta \in \mathcal{U}(\langle L = M, \sigma'\rangle) &\iff \theta \in \mathcal{U}(\langle L = M, \sigma\rangle) \\ &\iff \theta \in \mathcal{U}(\langle L = M, \sigma\rangle) \wedge \mathtt{s}\theta = \mathtt{s}\theta \\ &\iff \theta \in \mathcal{U}(\langle \mathtt{s} : L = \mathtt{s} : M, \sigma\rangle) \end{aligned}$$

Decomp1: We have $\mathtt{x}\theta = \mathtt{s}\theta$ and $L\theta = M\theta$.

Then $(\mathtt{x} : L)\theta = (\mathtt{s} : L)\theta = (\mathtt{s} : M)\theta$ as required.

Decomp1′: We have $\mathtt{x}\theta = \mathtt{a}\theta$ and $L\theta = M\theta$.

Then $(\mathtt{x} : L)\theta = (\mathtt{a} : L)\theta = (\mathtt{a} : M)\theta$ as required.

Decomp2: We have $\mathtt{x}\theta = (\mathtt{y} : \mathtt{x}')\theta$ and $(\mathtt{x}' : L)\theta = M\theta$.

Then $(\mathtt{x} : L)\theta = (\mathtt{y} : \mathtt{x}' : L)\theta = (\mathtt{y} : M)\theta$ as required.

Decomp2′: We have $\mathtt{y}\theta = (\mathtt{y} : \mathtt{x}')\theta$ and $L\theta = (\mathtt{y}' : M)\theta$.

Then $(\mathtt{y} : M)\theta = (\mathtt{x} : \mathtt{y}' : M)\theta = (\mathtt{x} : L)\theta$ as required.

Decomp4: We have $\mathtt{x}\theta = \mathtt{a}\theta$ and $\mathtt{y}\theta = (\mathtt{empty})\theta = \mathtt{empty}$.

Then $(\mathtt{a} : \mathtt{y})\theta = (\mathtt{x} : \mathtt{y})\theta = (\mathtt{x} : \mathtt{empty})\theta = \mathtt{x}\theta$ as required.

Subst1: We have $\mathtt{x}\theta = L\theta$

$$\begin{aligned}
\theta \in \mathcal{U}(\langle\ ,\sigma'\rangle) &\iff \theta \in \mathcal{U}(\langle\ ,\sigma_S \circ (\mathtt{x} \mapsto L)\rangle) \\
&\iff \theta \in \mathcal{U}(\langle\ ,\sigma_S \circ (\mathtt{x} \mapsto L)\rangle) \wedge \mathtt{x}\theta = L\theta \\
&\iff \theta \in \mathcal{U}(\langle \mathtt{x} = L, \sigma_S\rangle)
\end{aligned}$$

Subst2: We have $\mathtt{x}\theta = (\mathtt{empty})\theta$ and $L\theta = M\theta$.

Then $(\mathtt{x} : L)\theta = (\mathtt{empty} : L)\theta = L\theta = M\theta$ as required.

Subst3: We have $\mathtt{x}\theta = (\mathtt{a} : \mathtt{x}')\theta$ and $(\mathtt{x}' : L)\theta = M\theta$.

Then $(\mathtt{x} : L)\theta = (\mathtt{a} : \mathtt{x}' : L)\theta = (\mathtt{a} : M)\theta$ as required.

Orient1-4: Since $=_{\mathrm{AU}}$ is an equivalence relation and hence symmetric, a unifier of the conclusion is also a unifier of the premise.

□

Theorem 5.2 (Soundness). *If $P \Rightarrow^+ P'$ with $P' = \langle\ ,\sigma'\rangle$ in solved form, then σ' is a unifier of P.*

Proof. We have that σ' is a unifier of P' by definition. A simple induction with Lemma 10 shows that σ' must be a unifier of P. □

5.5 Completeness

In order to prove that our algorithm is complete, by Definition 5.6 we have to show that for any unifier δ, there is a unifier in our solution set that is more general.

Our proof involves using a special algorithm SELECT that takes an (initial) unification problem $\langle s =^? t\rangle$ together with an arbitrary unifier δ, and outputs a branch of the unification tree associated with a more general unifier than δ. The intuition here is that the computation of such branches proves a correspondence between the leaves of the unification tree and the desired complete set of unifiers. This is very similar to [Sie78] where completeness of a A-unification algorithm is shown using the same idea of a selector.

Lemma 11 (SELECT Lemma). *There exists an algorithm SELECT($\langle s =^? t\rangle, \delta$) that takes a unification problem $\langle s =^? t\rangle$ and a unifier δ as input and produces a branch of its unification tree represented as a sequence of rule selections $B = (b_1, \dots, b_k)$ such that:*

- *UNIFY($\langle s =^? t\rangle$) has a branch specified by B.*
- *For the sequence of rules $b \in B$: if σ is the substitution corresponding to b, then there exists an instantiation λ such that $\sigma \circ \lambda \leq \delta$.*

For example, consider the unification problem $\langle \mathtt{y : 2 = a : x}\rangle$ and its unifier $\delta = (\mathtt{x} \mapsto \mathtt{1 : 2}, \mathtt{y} \mapsto \mathtt{a : 1})$. The unification tree was shown in Figure 5.3. Applying SELECT would produce the sequence of unification rules (Subst3, Decomp2′, Orient1, Subst1), which corresponds to the right-most branch in the tree. The unifier at the end of this branch is $\sigma = (\mathtt{x} \mapsto \mathtt{y' : 2}, \mathtt{y} \mapsto \mathtt{a : y'})$ which is more general than δ using the assignment $\lambda = (\mathtt{y'} \mapsto \mathtt{1})$.

Now we are able to state our completeness result, which follows directly from Lemma 11.

Theorem 5.3 (Completeness). *For every unifier δ of a unification problem $\langle s =^? t\rangle$, there exist a unifier σ generated by UNIFY such that $\sigma \leq \delta$.*

5.5.1 Proving Completeness: The SELECT Algorithm

The SELECT algorithm takes a unification problem $\langle s =^? t\rangle$ together with a unifier δ, and outputs a branch of the unification tree associated with a unifier that is more general than δ. Here we present the core of the algorithm together with several examples of how it works. The full algorithm details are given in Appendix B. Afterwards, we show the proof of SELECT Lemma which establishes that SELECT is correct w.r.t. UNIFY.

SELECT Algorithm. Informally, the algorithm does some preprocessing, e.g. expanding and recording list variables, and then, depending on the current state, selects a unification rule and updates the state. The algorithm itself has *state* which is the given problem, together with some precomputed information based on the input unifier. It essentially behaves like a 2-tape TM with look-ahead which examines both input tapes simultaneously (the "head" of the current problem) as defined by pointers m and n, and can move each pointer separately and *only to the right*.

Given a unification problem $\langle s =^? t\rangle$ and a unifier δ for that problem, the algorithm proceeds as follows:

1. Preprocess s and t according to δ to obtain expanded expressions $\bar{s}, \bar{t}$

2. Initialize the state by setting the pointers m and n to 1 (the head of the initial problem)

3. Do case analysis based on the predicates $L, AtEnd$ and $Type$, and current state, until the pointers have reached the end of the input expressions. In each case, say which unification rule is selected, and update the current state by either moving the pointers or swapping the tapes.

The major part of the algorithm is the big case split depending on the predicate values at locations m and n, with the result that a specific rule is selected (added to the output) and pointers updated. This is repeated until both pointers reach the end of the input. The number of cases is rather larger due to the intricate dependencies between the different predicates and the current state of the algorithm. See Appendix B for the full details.

Preprocessing Preprocessing of the problem $\langle s =^? t\rangle$ w.r.t. a unifier δ involves:

- expanding each variable to the correct number of symbols, i.e. $x \to x_1 : \ldots : x_n$ where $n = |\delta(x)|$ is the size of the expression $\delta(x)$. Call x the *parent symbol* of $x_1, \ldots, x_n$, and the expanded strings $\bar{s}$ and $\bar{t}$. Each symbol x_i is called a 'pseudo-variable'.

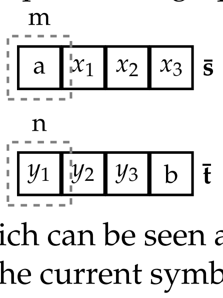
- for each binding $\{x \mapsto \mathit{empty}\}$ in δ, replacing x with x_e in $\bar{s}$ and $\bar{t}$. When SELECT sees such an empty list variable, it will select the rule Subst2.

- padding the shorter string with extra `empty` symbols to make $\bar{s}$ and $\bar{t}$ of equal length.

For example, the problem $\langle \mathtt{a} : \mathtt{x} =^? \mathtt{y} : \mathtt{b}\rangle$ has unifier $\delta = (\mathtt{x} \mapsto \mathtt{1} : \mathtt{2} : \mathtt{b}, \mathtt{y} \mapsto \mathtt{a} : \mathtt{1} : \mathtt{2})$. Preprocessing would produce the expanded problem $\langle \bar{s} =^? \bar{t}\rangle$ as

$$\mathtt{a} : \mathtt{x}_1 : \mathtt{x}_2 : \mathtt{x}_3 =^? \mathtt{y}_1 : \mathtt{y}_2 : \mathtt{y}_3 : \mathtt{b}$$

where each $\mathtt{x_i}$ and $\mathtt{y_i}$ are pseudo-variables. The expanded expressions $\bar{s}$ and $\bar{t}$ can be represented graphically:



which can be seen as the *state* of the SELECT algorithm. The pointers m and n point to the current symbols examined by the algorithm, and are graphically represented by the dashed boxes.

Predicates Apart from the current state, the execution of the algorithm is guided by several predicates. These predicates allow the algorithm to determine by how much it should move the above pointers, or whether it should move each pointer individually, or whether it should swap tape contents as required by the Orient rules. The predicates also allow the algorithm to infer the *shape* of the problem represented by the current state, and thus ensure the selected unification rule is indeed applicable.

L: The predicate $L(l_k)$ accepts a list expression l (either $\bar{s}$ or $\bar{t}$) together with a position value k (either m or n) and returns the number of adjacent pseudo-variables of l_k to the right of l_k. If l_k has no pseudo-variables adjacent to it or is a constant then $L(l_k) = 0$, i.e. it returns positive values only for symbols that have list variables as parents.

For example, for the above expanded unification problem:

$L(\bar{s},1) = 0$	a is a constant
$L(\bar{s},2) = 2$	after x_1 there is x_2 and x_3
$L(\bar{s},3) = 1$	after x_2 there is x_3
$L(\bar{s},4) = 0$	after x_3 there is no x_4
$L(\bar{t},1) = 2$	after y_1 there is y_2 and y_3
$L(\bar{t},2) = 1$	after y_2 there is y_3
$L(\bar{t},3) = 0$	after y_3 there is no y_4
$L(\bar{t},4) = 0$	b is a constant

Type: The predicate

$$Type(s_k) \rightarrow \{ListVar,\ AtomVar,\ AtomExpr,\ EmptyListVar,\ empty\}$$

accepts a list expression (either $\bar{s}$ or $\bar{t}$) together with a position value (m or n) and returns the type of the symbol of s_k.

For example for the expanded list expression $\bar{s} = a : "a" : x_e : y : empty$

$Type(\bar{s},1) = AtomVar$
$Type(\bar{s},2) = AtomExpr$
$Type(\bar{s},3) = EmptyListVar$
$Type(\bar{s},4) = ListVar$
$Type(\bar{s},5) = empty$

AtEnd: The predicate $AtEnd(s_k) \rightarrow \{true, false\}$ determines if the symbol to the right of the specified symbol (or sequence of pseudo-variables when $Type(s_k) = ListVar$) is not the empty list constant (`empty`) or the blank symbol Δ.

For example for the expression $\bar{s} = x_1 : x_2 : a$

$AtEnd(\bar{s},1) = true$	because of a
$AtEnd(\bar{s},2) = true$	because of a
$AtEnd(\bar{s},3) = false$	because at end of string

5.5.2 SELECT Examples

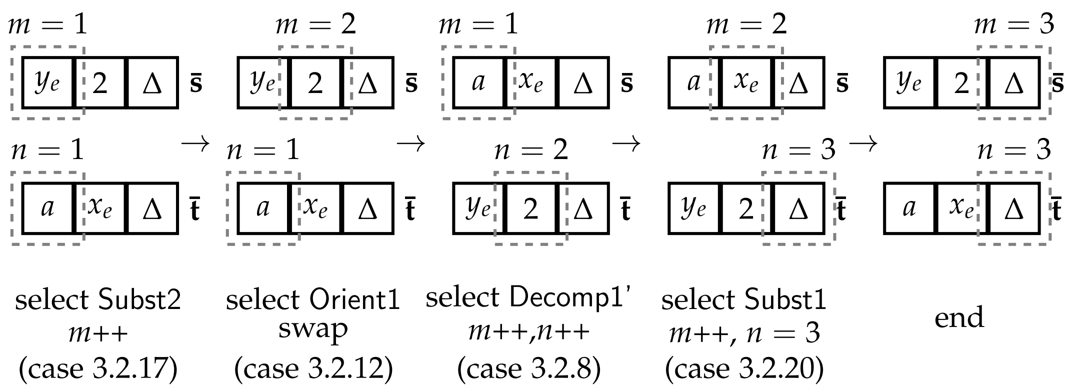
Next we give some examples of how the algorithm works in terms of its state, pointers and rule selection.

The first two examples are of the problem $P = \langle \mathtt{y:2} =^{?} \mathtt{a:x}\rangle$, with its unification tree already shown in Figure 5.3. The third example is of the problem $P = \langle \mathtt{n} =^{?} \mathtt{y:n}\rangle$ that has a single most general unifier $\sigma = \{\mathtt{y} \mapsto \mathtt{empty}\}$.

Example 19.

$$P = \langle \mathtt{y:2} =^{?} \mathtt{a:x}\rangle$$

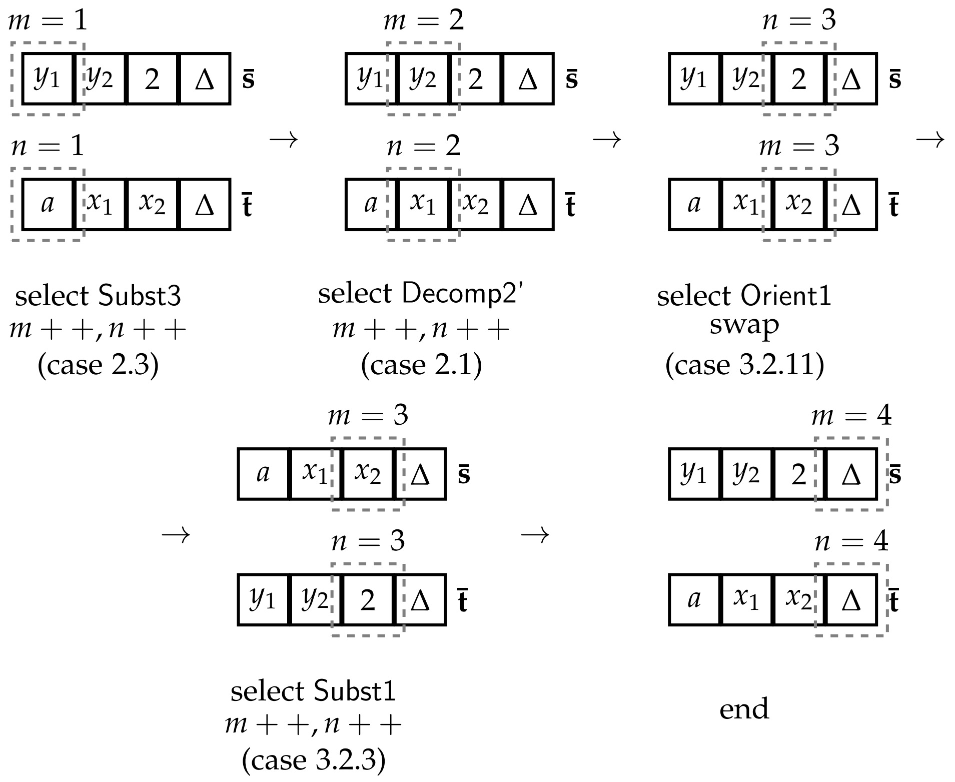
$$\delta = \{\mathtt{a} \mapsto \mathtt{2},\ \mathtt{x} \mapsto \mathtt{empty},\ \mathtt{y} \mapsto \mathtt{empty}\}$$



Example 20.

$$P = \langle \mathtt{y:2} =^{?} \mathtt{a:x}\rangle$$

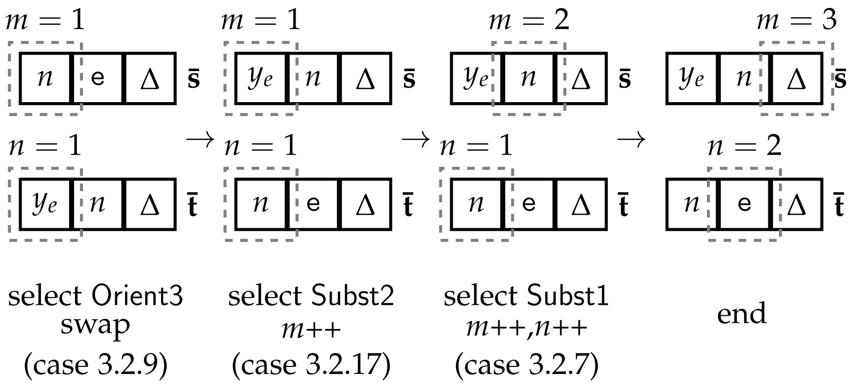
$$\delta = \{\mathtt{x} \mapsto \mathtt{a:1},\ \mathtt{y} \mapsto \mathtt{1:2}\}$$



Example 21.

$$P = \langle \mathtt{n} =^? \mathtt{y} : \mathtt{n} \rangle$$

$$\delta = \{\mathtt{y} \mapsto \mathtt{empty}\}$$



5.5.3 Putting it all together: Proving The SELECT Lemma

Next we give a proof sketch of the helper lemma used in proving completeness of UNIFY. The full proof is rather long (see Appendix C.1) due to the number of combinations of the various predicates for remaining length L, typing given by *Type* and check for end of input (*AtEnd*).

Proof Sketch of Lemma 11. The idea is to use induction on the number of rule selections $(b_1, b_2, \ldots)$ of B. For the base case, since the initial unification problem is equipped with the identity substitution id, then just picking $\lambda = id$ gives us $\sigma \circ \lambda = id \leq \delta$.

For the inductive case, we have to show that from an arbitrary unification problem P_k at node b_k of the unification tree, (1) UNIFY can apply the rule given by SELECT, and (2) there exists an instantiation λ_{k+1} such that when combined with the new unifier (resulting from applying the rule), it still produces a substitution more general than δ.

The rest of the proof is about considering all the cases for $P_k \Rightarrow P_{k+1}$, a case split involving the above predicates. Here we only give the top-level considerations.

Case 1. $L(\bar{s}, m) < L(\bar{t}, n)$. It must be the case that $\bar{t}_n$ is a list variable because $L(\bar{t}, n) > 0$ only for list variables. Then the problem is in one of the following forms depending on the other predicates: $\mathtt{x} : L = \mathtt{y} : M$ (to which rule Decomp2′ is applicable); $\mathtt{a} : L = \mathtt{y} : M$ (orientable by Orient3); $\mathtt{x} : L = M$ with $\delta(\mathtt{x}) = \mathtt{empty}$ (Subst2 is applicable).

Case 2. $L(\bar{s}, m) > L(\bar{t}, n)$. As in the previous case, $\bar{s}_m$ is a list variable because $L(\bar{s}, m) > 0$ only for list variables. Then the problem is in one of the following forms depending on the other predicates: $\mathtt{x} = L$ (to which rule Subst1 is applicable); $\mathtt{x} : L = \mathtt{y} : M$ (Decomp2 is applicable); $\mathtt{x} : L = \mathtt{a} : M$ (Subst3 is applicable);

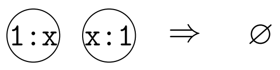
Case 3. $L(\bar{s}, m) = L(\bar{t}, n)$. This is the largest case. Most of the subcases are repeated, and their treatment is analogous. For $L(\bar{s}, m) = L(\bar{t}, n) = 0$ we have all combinations of types ($5^2 = 25$) for the parent symbols represented by the current state. $\square$

5.6 From Single Equations to Systems of Equations

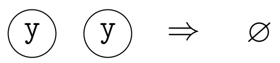
Now that we have specified how the algorithm produces a complete set of unifiers for a given label equation, we generalize the algorithm to handle systems of (independent) label equations. This is because when constructing critical pairs, we get systems of label equations rather than individual equations.

Before we go into how we specify our generalized version of UNIFY, we explain what is the problem with non-left-linear rules.

Non-left-linear rules. Consider the following rule schemata:

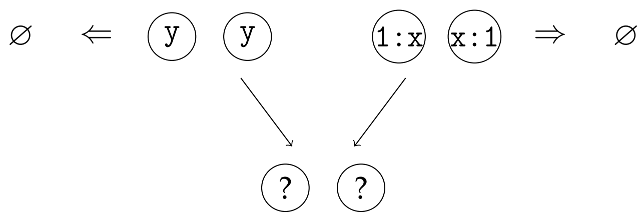


and



where `x` and `y` are `list` variables. These schemata are non-left-linear. The first schema has the label variable `x` which is shared between different graph nodes. The same situation arises in the second schemata and the variable `y`.

The overlap of these schemata in a critical way is as illustrated:



This overlap induces a system of equations $\{y = 1 : x,\ y = x : 1\}$. We can solve each equation individually: $\{\mathtt{y} \mapsto \mathtt{1} : \mathtt{x}\}$ and $\{\mathtt{y} \mapsto \mathtt{x} : \mathtt{1}\}$ but now due to sharing their combination is not a substitution as it maps `y` to different expressions. If we apply the first solution to the second problem, we get $\{\mathtt{1} : \mathtt{x} = \mathtt{x} : \mathtt{1}\}$ which is one of the well known cases where AU-unification is infinitary, i.e. has an infinite solution set: $\{\mathtt{x} \mapsto \mathtt{empty}\}$, $\{\mathtt{x} \mapsto \mathtt{1}\}$, $\{\mathtt{x} \mapsto \mathtt{1} : \mathtt{1}\}$ . . .. This kind of infinite solutions leads to an infinite number of critical pairs like the one shown above. For this reason, we forbid the sharing of list variables.

Remark 6 (Sharing of integer variables). It is not a problem to share integer variables. We have that left-hand graphs of schemata do not contain list variables as list expressions must be *simple*, and the unification algorithm presented previously does not introduce integer expressions, meaning that if integer expressions are part of the output of UNIFY then they are variables and original from the input schemata.

Now, integer variables cannot contain other expressions, and can only be assigned to numbers or renamed to other integer variables, thus not having an effect on subsequent application of unification rules or on other unification problems in the same system.

□

Generalized UNIFY. As already noted in Section 5.2, having restricted ourselves to left-linear schemata, the problem of solving a system of unification problems $\{s_1 =^? t_1, s_2 =^? t_2\}$ can be broken down to solving individual problems and combining the answers. If σ_1 and σ_2 are solutions to each individual equation, then $\sigma_1 \circ \sigma_2$ is a solution to the combined problem as σ_1 and σ_2 do not share variables. This intuition provides us with a generalization of UNIFY.

Definition 5.7 (Independent unification problems)**.** Two unification problems P_1 and P_2 are independent if they do not share list variables. A system of unification problems is independent if its unification problems are pairwise independent.

Definition 5.8 (Generalized UNIFY)**.** For a finite system of independent unification problems $(P_1, \ldots, P_n)$, the output of the generalized UNIF is the set of unifiers obtained by combining the unifiers of each individual unification problem:

$$\mathsf{UNIFY}(P_1, \ldots, P_n) = \left\{\sigma_1 \circ \ldots \circ \sigma_n \mid \sigma_i \in \mathsf{UNIFY}(P_i),\ 1 \le i \le n\right\}$$

□

For example, given two unification problems P_1 and P_2 (that do not share list variables) for which solving them individually produces $\mathsf{UNIFY}(P_1) = \{\alpha, \beta\}$ and $\mathsf{UNIFY}(P_2) = \{\lambda\}$, then $\mathsf{UNIFY}(P_1, P_2) = \{\alpha \circ \lambda,\ \beta \circ \lambda\}$. In practice, a system of such equations induces a *forest* of unification trees. The algorithm may generate/explore them one at a time. Furthermore, one may implement the sharing of integer variables by applying the generated substitution for an integer variable to the rest of the problems before they are solved. For example, if α in the above solution set solves an integer variable i by the assignment $\{\mathtt{i} \to 5\}$, then instead of calling on UNIFY on P_2 directly, one first applies the mapping to it, i.e. calls $\mathsf{UNIFY}(P_2\{\mathtt{i} \to 5\})$. This potentially introduces further non-determinism in the unification algorithm.

Theorem 5.4 (Soundness, Termination and Completeness of Generalized UNIFY)**.** *The Generalized version of UNIFY is sound, terminating and complete for a given finite system of independent unification problems $(P_1, \ldots, P_n)$ that do not share list variables.*

Proof. Termination follows from the fact that UNIFY terminates for each of the finite number of unification problems (Theorem 5.1). Soundness follows as UNIFY is sound and that the problems do not share list variables. Integer variables are not an issue due to the above remark about integer variables. Completeness follows from the fact that every possible combination of unifiers $\sigma_i \in \mathsf{UNIFY}(P_i)$ is considered in the output of the generalized UNIFY and that the basic unification algorithm is complete. □

5.7 Related Work

In this section, we give some bibliographic notes on unification for problems without types, and also on the use of unification in critical pair construction.

Unification. Unification theory is a widely studied topic in automated deduction. The technicalities of (syntactic) unification have been investigated since J.A. Robinson [Rob65]. The notions "unification" and "most general unifier" have been (independently) introduced by Knuth and Bendix [KB70] as means of testing term rewriting systems for confluence by computing critical pairs. *Equiational* unification has been introduced around the same time in the area of theorem proving [Plo72] as it had become clear that certain axioms (associativity, commutativity) need to be treated in a special manner.

Unification modulo associativity was shown to be decidable (i.e. whether there exists a unifier for a word equation) by Makanin [Mak77] for the case of unification with constants (albeit being in PSPACE [Pla99]), and a (non-minimal) unification algorithm given by [Sie78]. Siekmann [Sie78] also gave an extension to AU-unification, and later Jaffar [Jaf90] gave a minimal AU-unification algorithm. Plotkin [Plo72] describes a minimal A-unification procedure which need not terminate for non-solvable problems or problems with finite minimal complete sets of unifiers.

All of the above theory considers untyped variables, whereas in our case we require that unifiers are well-typed in accordance with the GP 2 type system. We encode this restriction in the special unification rules rather than e.g. filtering out substitutions that are not well-typed.

Unification for Critical Pairs. The earliest use of unification in the context of computing labels/attributes of critical pairs comes from [HKT02]. In this paper, the proposed critical pairs over attributed rules are labelled using the most general unifier of the overlapped terms, and the authors also give an alternative option to compute and compare normal forms w.r.t. the confluent and terminating term rewriting system (if it exists) induced by the equational axioms of the label algebra. However, using syntactic unification leads to incomplete critical pairs [EEPT06]. A restriction that avoids the need for unification altogether is to only allow variable or variable-free terms in rules, as developed in [EPT04] and implemented in [AGG].

A similar approach to avoiding unification is that of symbolic graph transformation [Dec17, KDL$^+$15], where graphs do not contain label expressions but instead have an associated logical formula describing the graph labels. Instead of computing labels of critical pairs, the unification process is avoided by including the induced system of label equations into the logical formula associated with the critical pair, which is left unresolved during computation.

Furthermore, both of the above approaches consider the label algebra as a parameter, i.e. it is fixed but arbitrary, and thus no general construction procedure is possible that includes the computation of labels. Moreover, even placing more severe restrictions on terms need not help with undecidability — there exist theories for which the ground word problem is undecidable [BS01].

5.8 Summary

In this chapter we have:

- described how to solve systems of label equations arising from the computation of critical pairs labelled with expressions;
- defined a rule-based unification algorithm;
- proven the soundness and termination of our algorithm;
- proven the completeness of the algorithm by defining a separate 'selector' algorithm;
- briefly reviewed alternative approaches to computing labels of critical pairs in graph transformation with attributes/labels, and the use of unification in such approaches.

Chapter 6

Joinability and Local Confluence Analysis

In Chapter 4 and Chapter 5, we introduced our notion of critical pairs for GP 2 schemata, showed how to construct such critical pairs by graph overlaps and label unification, and proved some useful properties about them, namely that there are finitely many of them (finiteness) and that they represent all possible conflicts (completeness).

The next step we take in this thesis is to study *how critical pairs are used* in the context of confluence analysis: to (dis)prove confluence, one constructs all critical pairs and analyses them for (strong) joinability. The correctness of this approach is established by showing the Local Confluence Theorem for GP 2 schemata. The practical side of this approach lies with the specification of a confluence analysis algorithm which takes a finite set of critical pairs, analyses them for joinability, and then outputs a confluence answer.

This chapter begins by introducing our notion of strong joinability of critical pairs, based on rewriting of graphs labelled with expressions. This is necessary as graphs in critical pairs are, in general, labelled with expressions rather than concrete values. Then, we discuss our approach to confluence analysis and its theoretical and practical apsects. The main theoretical result of this chapter is the Local Confluence Theorem for GP 2 schema rewriting, which in effect allows for the specification of a sound confluence analysis algorithm based on critical pairs. We discuss the practical aspects of the algorithm by looking at two refinements that reduce its search space and improve its accuracy. Finally, we explore some related and historical work on confluence analysis.

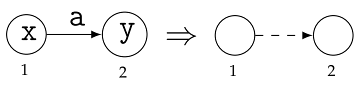
This chapter is based on [HP17a]. Schemata are assumed to be unconditional.

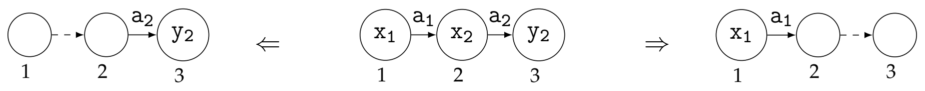
6.1 Confluence Analysis with Critical Pairs

Critical pair analysis allows for rigorous reasoning about the confluence of a set of rules. The main idea is checking of each critical pair $T_1 \Leftarrow S \Rightarrow T_2$ for *joinability* which in the simplest terms can be described as searching for a common graph derivable from T_1 and T_2. In such situations where this property holds for each critical pair – and ignoring the issue of termination – we can establish that the set of rules is confluent.

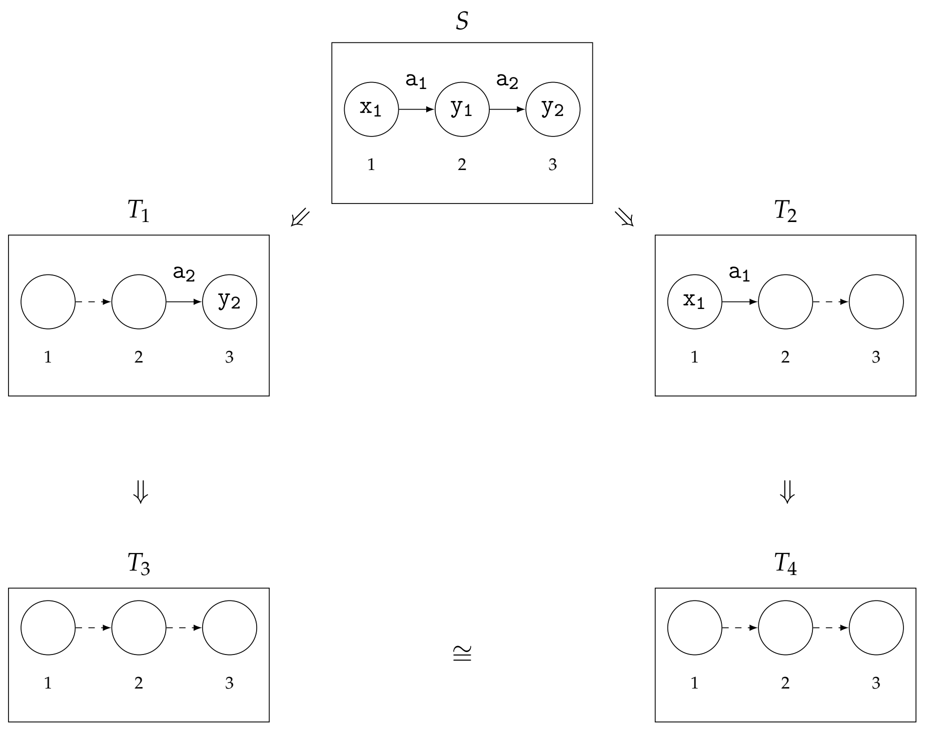
Fundamentally, joinability involves rewriting the graphs of critical pairs. Take, for example, the rule schema `unlabel` introduced in Chapter 4, and one of its critical pairs constructed from overlapping the schema in conflicting ways (previously shown in Figure 4.4):

`unlabel(a,x,y:list)`





where the variables are indexed to signify from which instance of `unlabel` they originate. The critical pair represents a conflict over a three-node sequence where the middle node gets relabelled to the empty list (relabelling conflict). The graphs of the critical pair are non-isomorphic unless all variables get instantiated to the empty list as well (e.g. see Figure 4.1). To *join* this critical pair means to keep applying the schema to reach a common graph, or equivalently, a pair of isomorphic graphs. In this case, this is possible:



The graphs resulting from the derivations $T_1 \Rightarrow T_3$ and $T_2 \Rightarrow T_4$[1] are isomorphic: they have the same structure, all nodes/edges are labelled with the empty list, and all edges are dashed. These graphs are also normal forms (no rules are further applicable) due to the semantics of the dashed mark.

We can therefore state the practical problem of confluence analysis we explore: Find a condition, called strong joinability, such that a terminating set $\mathcal{R}$ of rule schemata is confluent if each symbolic critical pair involving rules in $\mathcal{R}$ is strongly joinable.

Due to the informal nature of the above statement, some clarifications are needed. The above problem does not state at all the nature of strong joinability, but in practice, it is about rewriting the result graphs of critical pairs, and that is due to the definition of confluence. This leads to a subproblem, namely how to apply schemata to graphs labelled with expressions, which we solve first. The evidence of strong joinability, i.e. the joining derivations, are used in the proof of the Local Confluence Theorem. Regarding the practical side of joinability, the notions it is composed of should be computable, or at least we should be able to distinguish between different sources of non-computability. An obvious source of non-computability is the kind of label expressions that appear in rules — they could involve non-linear arithmetic, and hence figuring out whether the graphs derived from the critical pair are isomorphic becomes undecidable. Last but not least, we want not only to establish soundness of the approach, i.e. proving the Local Confluence Theorem, but also to give an *algorithm* for confluence analysis, and discuss the practical implications of strong joinability checking.

In this chapter we propose our solution to the above problem. First, we define *symbolic rewriting*, an approach for applying rule schemata to graphs labelled with expressions. This kind of rewriting is shown to be sound in the sense that a symbolic derivation can be instantiated to the host graph derivations it represents by assigning the variables occurring in the derivation with concrete data. Second, we define our notion of strong joinability based on symbolic rewriting. Third, we prove the Local Confluence Theorem, i.e. that using out notion of strong joinability is sufficient for showing local confluence of a set of terminating rules. To do so, we closely follow the proof of the same theorem in other graph rewriting frameworks, e.g. [EPT04]. Last but not least, we give our confluence analysis algorithm, and propose several extensions that allow us to (1) restrict the joinability search space using so-called *persistent reducts*, and (2) increase the precision of the analysis by using a more relaxed notion of graph isomorphism.

The above approach to confluence analysis is appealing because it breaks down the requirements of a confluence proof into several independent joinability proofs that can be established independently. In other words, a confluence checker can present individual critical pairs it was (un)able to join to a user who can decide on what further action is necessary. Indeed, existing tools do that, and this kind of separation of concerns is very helpful when trying to establish the confluence of existing GP 2 programs (discussed in Chapter 7).

[1]Since $\Rightarrow$ is not defined over such graphs, we abuse notation for now and fix it immediately at the start of this chapter.

A related problem is that of showing non-confluence. Certainly when the confluence analysis techniques presented in this thesis are unable to show that a set of rules is confluent, the next reasonable question to ask is *why?*. Our approach to non-confluence is to give two sufficient conditions that guarantee the soundness of a non-confluence answer. We discuss those at the end of this chapter.

6.2 Symbolic Rewriting

In this section we propose *symbolic rewriting* for GP 2 allowing for the application of rule schemata to the graphs in critical pairs. The current formalism of schema application is defined in the setting of host graphs (labelled with concrete values). The major difference between symbolic rewriting and the previous notion of schema rewriting is that symbolic rewrites use *substitutions*, i.e. mappings from variables to expressions, rather than assignments, i.e. mappings from variables to variable-free data. The overall aim is to provide a mechanism for establishing the (strong) joinability of critical pairs.

Informally, symbolic rewriting introduces a relation on graphs ($\Rrightarrow$) where matching is done by treating variables as typed symbols/constants. Symbolic rewriting allows for the representation of multiple host graph direct derivations. Here the notion of representation is similar to that of Chapter 4: symbolic critical pairs represent an infinite number of minimal conflicts the same way symbolic derivations represent an infinite number of direct derivations at the host level. This type of rewriting is very similar to symbolic graph transformation, e.g. [OL12].

To apply a rule schema to a graph, the graphs of the schema are first instantiated by replacing their labels according to some substitution σ. A substitution σ maps each variable occurring in a given graph to an expression in GP 2's label algebra. Its unique extension σ^* replaces the graph's label expressions according to σ. In the terminology of Chapter 2, the instantiation of a schema is done by means of a substitution, and the rule instance is a graph transformation rule with relabelling, albeit the labels being expressions rather than data values.

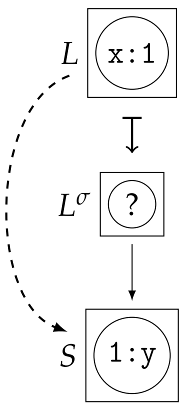
Definition 6.1 (Instance by substitution)**.** Consider a graph G in $\mathcal{G}_\perp$ and a substitution $\sigma\colon X \to RSLabel$. The graph instance G^σ is the graph in $\mathcal{G}_\perp$ obtained from G by replacing each label l with $\sigma^*(l)$. The instance of a rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ is the rule $r^\sigma = \langle L^\sigma \leftarrow K^\sigma \rightarrow R^\sigma\rangle$.

Definition 6.2 (Symbolic direct derivation)**.** A symbolic direct derivation using rule schema r, substitution σ, and match $g : L^\sigma \to S$, between graphs $S, T \in \mathcal{G}(RSLabel)$ consists of two natural pushouts as in Figure 6.1a.

We denote symbolic derivations by $S \overset{r,g,\sigma}{\Rrightarrow} T$. Symbolic critical pairs, discussed in Chapter 4, involve such symbolic derivations. Operationally, constructing symbolic derivations involves obtaining a substitution σ for the variables of L given a premorphism $L \to S$, and then constructing a direct derivation with relabelling as in Chapter 2. The problem can be seen in Figure 6.1b.

$$\begin{tikzcd}
L \arrow[d, mapsto] & K \arrow[l] \arrow[r] \arrow[d, mapsto] & R \arrow[d, mapsto] \\
L^\sigma \arrow[d, "g"'] \arrow[dr, phantom, "\text{NPO}"] & K^\sigma \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{NPO}"] & R^\sigma \arrow[d] \\
S & O \arrow[l] \arrow[r] & T
\end{tikzcd}$$

(a) A symbolic direct derivation.



(b) Symbolic matching.

Figure 6.1: Symbolic rewriting.

Example 22 (Symbolic derivation). In Figure 6.2 we give a symbolic derivation involving the rule add, part of the Shortest Distances program previously shown in Subsection 2.2.3.2. (The rule is reproduced at the top of the figure). The rule is applied to a graph labelled with expressions S that appears during the confluence analysis of the given program (see Chapter 7). For simplicity we have changed to node identifiers of the rule schema to make obvious the match $L \to G$ rather than explicitly naming the mapping.

The rule instance involves the substitution $\sigma = \{\mathtt{x} \mapsto \mathtt{y}, \mathtt{m} \mapsto \mathtt{m} + \mathtt{n}, \mathtt{y} \mapsto \mathtt{y}', \mathtt{n} \mapsto \mathtt{n}\}$ where the domain of the substitution are the variables of the schema add and the variable range are the variables of S, the input symbolic graph. An important point is that the variables of L are different from the variables of L^σ and S: the variables $\mathtt{x}, \mathtt{y}, \mathtt{m}, \mathtt{n}$ in the top row are local to the schema, whereas the others variables (on middle and bottom row) are part of the symbolic graphs L^σ, S, R^σ, T.

□

When we apply a schema to a graph S, it is impossible to replace the variables of S as the substitution involved only names the variables of the rule schema. Thus for the premorphism $L \to S$ in Figure 6.1b there is no valid substitution to make the symbolic application work — one would need to substitute the variable y with $\mathtt{y}'\!:\!\mathtt{1}$ where $\mathtt{y}'$ is fresh. As a result, this kind of rewriting is *incomplete* in that not all host graph derivations can be represented by symbolic derivations.

Remark 7 (Variable scope). The overall purpose of symbolic rewriting is to apply schemata to graphs in critical pairs. It is important to note where the variables of symbolic graphs come from. For schemata, variables are local to the schema and never appear in host graphs. For symbolic graphs however, these variables are local to the critical pair being investigated for joinability. Rules cannot create variables, and hence no new variables can be created during symbolic rewriting.

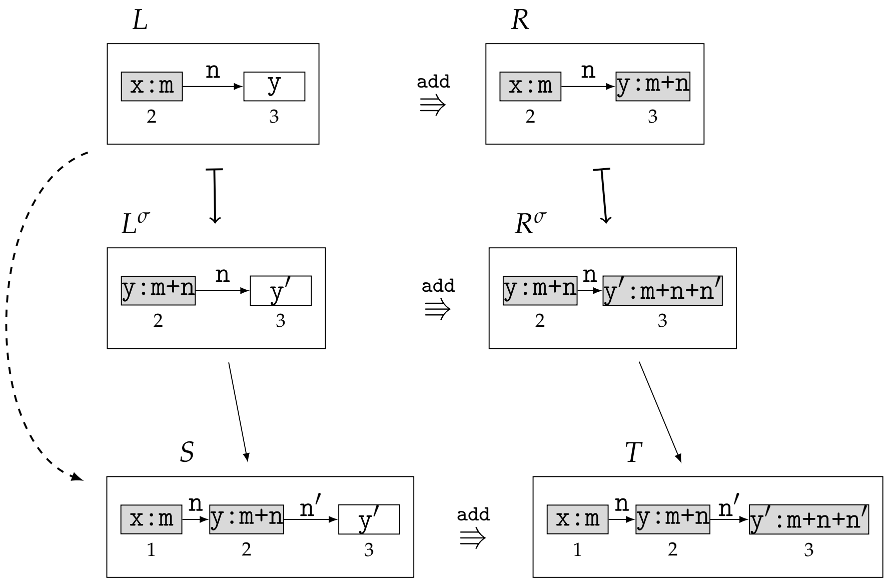


Figure 6.2: Symbolic derivation using the rule add.

Remark 8. We can treat symbolic rewriting as the application of a rule schema to a graph with label set being a term algebra, similar to the approach of [HP16a]. This is consistent with the fact that assignments $X \to A$ become substitutions $X \to T(X)$ (set $A = T(X)$).

Another property of this type of rewriting is that derivations cannot introduce new variables. This ensures that symbolic derivations are determined uniquely by a match and substitution for the variables occurring in the left-hand graph of the schema.

Proposition 1 (Existence and uniqueness of symbolic derivations). *Consider a rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, an injective premorphism $g\colon L \to S$ with S in $\mathcal{G}_\perp(T(X))$, and a substitution $\sigma\colon X \to T(X)$. Then there exists a symbolic direct derivation $S \overset{r,g',\sigma}{\Rrightarrow} T$ such that g' is induced by g and σ, if and only if g satisfies the dangling condition and each item x in L satisfies:*

$$l_S(g(x)) = \sigma^*(l_L(x)).$$

Moreover, in this case T is determined uniquely up to isomorphism.

Proof. Follows directly from Proposition 1 of [HP16a] and Remark 8. □

The following lemma states that the application of symbolic rule schema coincides, in some sense, with respect to the host graph derivations it represents. It relates a symbolic derivation $S \Rrightarrow T$ and any host graph derivation that involves an instance of its input graph S using the same rule and match.

Lemma 12 (Soundness of symbolic rewriting)**.** *For each symbolic derivation $S \overset{r,g,\sigma}{\Rrightarrow} T$, host graph G and assignment λ such that $G = S^\lambda$, there exists a direct derivation $G \overset{r,g,\alpha}{\Rightarrow} H$ where $H = T^\lambda$ and $\alpha = \lambda \circ \sigma$.*

Proof. The double-pushout of $G \overset{r,g,\alpha}{\Rightarrow} H$ is the same as the one of $S \overset{r,g,\sigma}{\Rrightarrow} T$ when ignoring labels. Define $\alpha = \lambda \circ \sigma$. The label preservation property of the match $L^\sigma \to S$ implies the label preservation of morphism $(L^\sigma)^\lambda \to S^\lambda$, i.e. of $L^\alpha \to G$. Since the dangling condition is not concerned with labels, $L^\alpha \to G$ is a valid match for r^α in G. The labelling functions of H and T are uniquely determined by r^α, r^σ, S and $G = S^\lambda$, as shown in [HP16a, Proposition 1] and Proposition 1 above. As a consequence, we get that $T^\lambda = H$. $\square$

6.3 Joinability

Confluence analysis is based on the joinability of critical pairs. Informally, a symbolic critical pair $T_1 \Lleftarrow S \Rrightarrow T_2$ is *joinable* if there exist symbolic derivations from T_1 and T_2 to a common graph $T_1 \Rrightarrow^* X \Lleftarrow^* T_2$.

As explained in Chapter 2, it is known that joinability of all critical pairs is not sufficient to prove local confluence [Plu93]. Instead, one needs to consider a slightly stronger notion called *strong* joinability that, apart from joinability, requires a special set of so-called *persistent* items to be preserved by the joining derivations. This is because extending the joining derivations is not always possible in the context of graph transformation. The definition of persistent items is given below, where we consider the graphs T_1 and T_2 (although distinct) to have common items.

Definition 6.3 (Persistent items)**.** The set of *persistent items* $\mathcal{P}$ of a critical pair $T_1 \Lleftarrow S \Rrightarrow T_2$ consists of all items in S that are preserved by both steps: $\mathcal{P} = T_1 \cap T_2$.

A similar formulation of joinability is based on *strict* joinability where instead of talking about persistent nodes, one talks about the pullback of the inclusions $O_1 \subseteq S$ and $O_2 \subseteq S$ where O_1 and O_2 are the intermediate graphs in the critical pair of derivations (see Figure 6.1a). A formulation based on pullbacks is more suitable when the subsequent proofs are done in an algebraic setting, specifically in adhesive categories (e.g. see [EHPP04]), because the expression of set intersection in such a setting is a pullback.

Even so, the intuition is the same: the common graph N, which is preserved by both symbolic derivations of a critical pair, must be preserved by the joining derivations.

Definition 6.4 (Strong joinability)**.** A symbolic critical pair $T_1 \Lleftarrow S \Rrightarrow T_2$ is strongly joinable if we have the following:

1. joinability: there exist symbolic derivations $T_1 \Rrightarrow^* X_1 \cong X_2 \Lleftarrow^* T_2$ where $i : X_1 \to X_2$ is an (E-)isomorphism.

2. strictness: let N be the pullback object of $O_1 \to S \leftarrow O_2$ (1). Then there exist morphisms $N \to O_3$ and $N \to O_4$ such that the squares (2), (3) and (4) commute:

$$\begin{tikzcd}
 & & S & & \\
 & O_1 \arrow[ur] \arrow[dl] & (1) & O_2 \arrow[ul] \arrow[dr] & \\
T_1 & (2) & N \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] & (3) & T_1 \\
 & O_3 \arrow[ul] & (4) & O_4 \arrow[ur] & \\
 & & X_1 \arrow[ul] \arrow[r, "i"] & X_2 \arrow[u] &
\end{tikzcd}$$

We will allow the morphism $i : X_1 \to X_2$ to be an E-isomorphism where labels are required to be *equivalent* (in GP's label algebra) rather than equal: two graphs $G, H \in \mathcal{G}(T(X))$ are E-isomorphic (denoted by $G \cong_E H$) if there exists a bijective premorphism $i : G \to H$ such that $l_H(i(x)) \approx_E l_G(x)$ for all items of G. (This is further discussed in subsection 6.5.1.) Here $\approx_E$ is the equivalence relation on GP2 expressions given by all the equations valid in GP 2's label algebra of integer arithmetic and list/string concatenation. This allows for a confluence algorithm that is more accurate in terms of when are critical pairs joinable.

The strictness condition can be restated in terms of the track morphisms of the joining derivations, as in [Plu93]: the track morphisms $track_{S \Rightarrow T_1 \Rightarrow^* X_1}$ and $track_{S \Rightarrow T_1 \Rightarrow^* X_2}$ are defined and commute on the persistent nodes of the critical pair, i.e. $i(track_{S \Rightarrow T_1 \Rightarrow^* X_1}(v)) = track_{S \Rightarrow T_2 \Rightarrow^* X_2}(v)$ for each $v \in \mathcal{P}$. The graphs O_3 and O_4 in the above definition are the derived spans of the joining derivations. For a formal comparison between the two versions of the strong joinability definition, see Section C.2.

Lemma 13. *If a critical pair is strongly joinable according to Definition 6.4, then its set of persistent nodes exist in N, i.e. $\mathcal{P} \subseteq N$.*

Proof. The definition of a persistent set of items can be interpreted as the existence of inclusions $\mathcal{P} \subseteq O_1$ and $\mathcal{P} \subseteq O_2$ such that the square $\mathcal{P}O_1O_2S$ commutes. By the universal property of pullbacks, there exists a morphism $\mathcal{P} \to N \in \mathcal{M}$. Finally, this morphism is an inclusion as the morphisms $\mathcal{P} \subseteq O_1$ and $\mathcal{P} \subseteq O_2$ are also inclusions.

$$\begin{tikzcd}
\mathcal{P} \arrow[dr, dashed, "\subseteq"] \arrow[drr, bend left, "\subseteq"] \arrow[ddr, bend right, "\subseteq"'] & & \\
 & N \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{PB}"] & O_1 \arrow[d] \\
 & O_2 \arrow[r] & S
\end{tikzcd}$$

$\square$

To prove the Local Confluence Theorem, we will need the following lemma that relates the joinability of a symbolic critical pair and the joinability of all minimal host graph conflicts it represents.

Lemma 14 (Joinability preservation)**.** *If a symbolic critical pair $T_1 \Lleftarrow S \Rrightarrow T_2$ is strongly joinable, then for each of its instances according to any assignment λ for its variables, we have that $T_1^\lambda \Leftarrow S^\lambda \Rightarrow T_2^\lambda$ is also strongly joinable.*

Proof. An instance $T_1^\lambda \Leftarrow S^\lambda \Rightarrow T_2^\lambda$ involves the same double-pushouts as $T_1 \Lleftarrow S \Rrightarrow T_2$ due to Lemma 12, with the only difference being graph labels. From the E-isomorphism $i : X_1 \to X_2$ we get that $l_{X_2}(i(x)) \approx_E l_{X_1}(x)$ for all nodes and edges of X_1. Applying λ to both sides, we get $\lambda(l_{X_2}(i(x))) \approx_E \lambda(l_{X_1}(x))$ and thus $l_{X_2^\lambda}(i(x)) \approx_E l_{X_1^\lambda}(x)$ meaning i induces an E-isomorphism $X_1^\lambda \cong_E X_2^\lambda$. Moreover, since X_1^λ, X_2^λ are host graphs, this also means $X_1^\lambda \cong X_2^\lambda$. $\square$

6.4 Local Confluence Theorem

In this section we present the Local Confluence Theorem which establishes the local confluence of $\mathcal{R}$ given all symbolic critical pairs are strongly joinable. It was first shown in [Plu93] for the (hyper)graph case and later extended to (weak) adhesive categories in [EHPP04], and closely follows the Local Confluence Theorem proof of [EEPT06, Theorem 6.28].

The proof loosely proceeds as follows. For a given pair of direct derivations $H_1 \overset{r_1,m_1,\alpha}{\Leftarrow} G \overset{r_2,m_2,\alpha}{\Rightarrow} H_2$, we have to show the existence of derivations $H_1 \Rightarrow^*_{\mathcal{R}} X_1' \cong X_2' \Leftarrow^*_{\mathcal{R}} H_2$ as the outer part of Figure 6.3. If the given pair is independent, this follows from the Church-Rosser Theorem (Theorem 3.1). If the given pair is in conflict, the critical pair Completeness Theorem 4.2 implies the existence of a symbolic critical pair $T_1 \overset{r_1,m_1',\sigma}{\Lleftarrow} S \overset{r_2,m_2',\sigma}{\Rrightarrow} T_2$. By assumption, this critical pair is strongly joinable. The proof then uses the definition of joinability to prove the joining derivations of the critical pair can be extended by the morphism $e : P \to G$.

Theorem 6.1 (Local Confluence Theorem)**.** *A set $\mathcal{R}$ of left-linear rule schemata is locally confluent if all of its symbolic critical pairs are strongly joinable.*

Remark 9 (Auxiliary results). The full proof of the theorem requires the construction of the boundary/context graph of $e : P \to G \in \mathcal{N}$. This is always possible in our setting - use the same definition as in the unlabelled case (e.g. see [EEPT06, Example 6.2]) and omit labels as done in Figure 6.4. Other necessary results include the Embedding and Extension Theorems, pair factorization, pushout-pullback decomposition. These are easily obtained by inspecting the proofs in [EHPP04] which already considers categories with a special set of vertical morphisms.

$$\begin{tikzcd}
 & & & G \arrow[dddlll, Rightarrow] \arrow[dddrrr, Rightarrow] & & & \\
 & & & P \arrow[u, "e"] \arrow[ddll, Rightarrow] \arrow[ddrr, Rightarrow] & & & \\
 & & & S \arrow[u, mapsto] \arrow[dl, Rrightarrow] \arrow[dr, Rrightarrow] & & & \\
H_1 \arrow[dddrr, Rightarrow, "*"] & Q_1 \arrow[l] \arrow[ddr, Rightarrow, "*"] & T_1 \arrow[l, mapsto] \arrow[d, Rrightarrow, "*"] & & T_2 \arrow[r, mapsto] \arrow[d, Rrightarrow, "*"'] & Q_2 \arrow[r] \arrow[ddl, Rightarrow, "*"'] & H_2 \arrow[dddll, Rightarrow, "*"'] \\
 & & X_1 \arrow[d, mapsto] & \cong_E & X_2 \arrow[d, mapsto] & & \\
 & & X_1^\lambda \arrow[d] & \cong & X_2^\lambda \arrow[d] & & \\
 & & X_1' & \cong & X_2' & &
\end{tikzcd}$$

Figure 6.3: Local Confluence diagram.

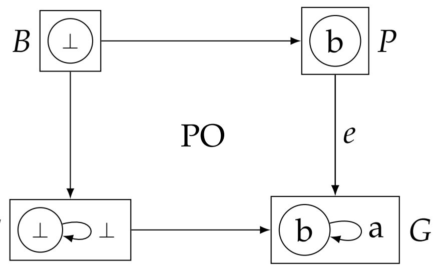


Figure 6.4: Example initial pushout in $\mathcal{G}_\perp$.

Proof. For a given pair of direct derivations $H_1 \Leftarrow_{r_1,m_1,\alpha} G \Rightarrow_{r_2,m_2,\alpha} H_2$, we have to show the existence of derivations $t_1' : H_1 \overset{*}{\Rightarrow}_{\mathcal{R}} X$ and $t_2' : H_2 \overset{*}{\Rightarrow}_{\mathcal{R}} X$.

If the given pair is independent, this follows from Theorem 3.1.

If the given pair is in conflict, Theorem 4.2 implies the existence of a symbolic critical pair $T_1 \overset{r_1,m_1',\sigma}{\Lleftarrow} S \overset{r_2,m_2',\sigma}{\Rrightarrow} T_2$ with extension diagrams as shown in the upper part of Figure 6.3 involving an instance of the critical pair, and $e : K \to G \in \mathcal{N}$. These diagrams are obtained from the construction in Theorem 4.2. By assumption, this critical pair is strongly joinable. By Lemma 14, this leads to derivations $t_1 : Q_1 \cong T_1^\lambda \Rightarrow^* X_1^\lambda$ and $t_2 : Q_2 \cong T_2^\lambda \Rightarrow^* X_2^\lambda$ with $X_1^\lambda \cong X_2^\lambda$, since the critical pair instance $Q_1 \Leftarrow P \Rightarrow Q_2$ is strongly joinable. Furthermore, the graphs involved in the strong joinability of $Q_1 \Leftarrow P \Rightarrow Q_2$ are instances of the same graphs in Definition 6.4

according to the assignment λ.

Since squares NO_1O_2S and $N^\lambda O_1^\lambda O_2^\lambda P$ are $\mathcal{M}$-pullbacks and $\mathcal{M}$ is stable under $\mathcal{M}$-pullbacks (Fact A.1.3), then $N \to O_1, N \to O_2, N^\lambda \to O_1^\lambda, N^\lambda \to O_2^\lambda \in \mathcal{M}$. Since squares (2), (3) and (4) in Definition 6.4 are commutative, and $O_3^\lambda \to Q_1, O_4^\lambda \to Q_2, O_1^\lambda \to Q_1, O_2^\lambda \to Q_2 \in \mathcal{M}$, then $N \to O_3, N \to O_4, N^\lambda \to O_3^\lambda, N^\lambda \to O_4^\lambda \in \mathcal{M}$ since $\mathcal{M}$ is closed under composition and decomposition (Fact A.1.1).

Let B and C be the boundary/context graphs of $e : P \to G \in \mathcal{N}$. We have that there exist morphisms $B \to O_1^\lambda, C \to D_1 \in \mathcal{M}$ such that $B \to P = B \to O_1^\lambda \to P$, $C \to G = C \to D_1 \to G$ (by [Dec17, Def. 3.26]),

By the closure property of the boundary/context construction ([Dec17, Fact. 3.28]), the same graphs B and C are also boundary/context graphs of $Q_1 \to H_1$ and $Q_2 \to H_2$. Similarly, there exist morphisms $B \to O_2^\lambda, C \to D_2 \in \mathcal{M}$ such that $B \to P = B \to O_2 \to P, C \to G = C \to D_2 \to G$.

Combining with the above, this means that square $BO_1^\lambda O_2^\lambda P$ commutes. Due to the uniqueness property of square $N^\lambda O_1^\lambda O_2^\lambda P$ (an $\mathcal{M}$-pullback), we get a unique morphism $B \to N^\lambda$ such that $B \to O_1^\lambda = B \to N^\lambda \to O_1^\lambda$ and $B \to O_2^\lambda = B \to N^\lambda \to O_2^\lambda$. We have that $B \to O_1^\lambda, N^\lambda \to O_1^\lambda \in \mathcal{M}$ which means that $B \to N^\lambda \in \mathcal{M}$ since $\mathcal{M}$ is closed under decomposition (Fact A.1.1).

Now we show that $Q_1 \to H_1$ is consistent w.r.t. $t_1 : Q_1 \Rightarrow^* X_1^\lambda$. Let $B \to O_3^\lambda = B \to N^\lambda \to O_3^\lambda$, which is in $\mathcal{M}$ since both morphisms are in $\mathcal{M}$ and $\mathcal{M}$ is closed under composition (Fact A.1.1). Due to commutativity of (2), we get the following equalities $B \to Q_1 = B \to O_1^\lambda \to Q_1 = B \to N^\lambda \to O_1^\lambda \to Q_1 = B \to N^\lambda \to O_3^\lambda \to Q_1$. Hence the morphism $B \to O_3^\lambda$ as defined shows that $Q_1 \to H_1$ is consistent w.r.t. $t_1 : Q_1 \Rightarrow^* X_1^\lambda$. Analogously, $B \to O_4^\lambda = B \to N^\lambda \to O_4^\lambda$ shows that $Q_2 \to H_2$ is consistent w.r.t. $t_2 : Q_2 \Rightarrow^* X_2^\lambda$ (using commutativity of (3)).

By Theorem 6.2, we obtain derivations $t_1' : H_1 \Rightarrow^* X_1'$ and $t_2' : Q_2 \Rightarrow^* X_2'$ as the bottom of Figure 6.3. We just need to show that $X_1' \cong X_2'$. We show this using pushout uniqueness. By Theorem 6.2, the graphs X_1' and X_2' are pushouts of morphism spans $C \leftarrow B \to O_3^\lambda \to X_1^\lambda$ and $C \leftarrow B \to O_4^\lambda \to X_2^\lambda$. We have that both are based on $B \to C$. We only need to show that the morphisms $B \to O_3^\lambda \to X_1^\lambda$ and $B \to O_4^\lambda \to X_2^\lambda$ are equal. We have that (4) is commutative: $N^\lambda \to O_3^\lambda \to X_1^\lambda\ (\cong\ X_2^\lambda) = N^\lambda \to O_4^\lambda \to X_2^\lambda$. Combining with $B \to N^\lambda$, we get $B \to N^\lambda \to O_3^\lambda \to X_1^\lambda (\cong X_2^\lambda) = B \to N^\lambda \to O_4^\lambda \to X_2^\lambda$, as required.

This concludes the proof that the pair of derivations $H_1 \Leftarrow_{r_1,m_1,\alpha} G \Rightarrow_{r_2,m_2,\alpha} H_2$ is joinable.

□

6.4.1 Auxiliary Results

In order to talk about consistency of an extension morphism w.r.t. a derivation and thus when is an extension of a derivation possible, first we define so-called "derived spans". A derived span allows us to precisely define when can a derivation be extended to a 'larger' derivation.

Definition 6.5 (Extension diagram)**.** An extension diagram between two derivations $t : G \Rightarrow^* G_n$ and $t' : G' \Rightarrow^* G'_n$ with an (extension) morphism $e : G \to G' \in \mathcal{N}$ consists of the following pushouts (right):

$$\begin{tikzcd}
t : G \arrow[r, Rightarrow, "*"] \arrow[d, "e"'] & G_n \arrow[d] \\
t' : G' \arrow[r, Rightarrow, "*"] & G'_n
\end{tikzcd}
\qquad
\begin{tikzcd}
r_i^{\alpha_i} : L_i^{\alpha_i} \arrow[d, "m_i"] \arrow[dr, phantom, "\text{NPO}"] & K_i^{\alpha_i} \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{NPO}"] & R_i^{\alpha_i} \arrow[d] \\
G_i \arrow[d, "e_i"] \arrow[dr, phantom, "\text{NPO}"] & D_i \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{NPO}"] & G_{i+1} \arrow[d, "e_{i+1}"] \\
G'_i & D'_i \arrow[l] \arrow[r] & G'_{i+1}
\end{tikzcd}$$

Definition 6.6 (Derived span; Consistency)**.** The derived span of an identity derivation $G \Rightarrow^* G$ is the span $(G \leftarrow G \rightarrow G)$. The derived span of a direct derivation $G \Rightarrow^* H$ is the span $(G \leftarrow D \rightarrow H)$. For a derivation $t : G \Rightarrow^* G_{n-1} \Rightarrow G_n$, the derived span is $der(t) = (G \leftarrow D_n \rightarrow G_n)$ where D_n is the *pullback* of $D_{n-1} \rightarrow G_n \leftarrow D_n$, and D_{n-1} is defined by $der(G \Rightarrow^* G_{n-1}) = (G \leftarrow D_{n-1} \rightarrow G_{n-1})$.

$$\begin{tikzcd}
 & & D_n \arrow[dl, dotted] \arrow[dr, dotted] \arrow[d, phantom, "PB"] & & \\
 & D_{n-1} \arrow[dl] \arrow[r] & G_{n-1} & D_n \arrow[l] \arrow[r] & G_n \\
G & & & &
\end{tikzcd}$$

An (extension) morphism $k : G \to G'$ is *consistent* w.r.t. a derivation $t : G \Rightarrow^* G_n$ with derived span $der(t) = (G \leftarrow D \rightarrow G_n)$ if there exists a morphism $B \rightarrow D \in \mathcal{M}$ between the boundary B of k and the graph G of $der(t)$ such that $B \rightarrow D \rightarrow G = B \rightarrow G$:

$$\begin{tikzcd}
B \arrow[r] \arrow[d] \arrow[rr, dotted, bend left, "="] & G \arrow[d, "k"] & D \arrow[l] \arrow[r] & G_n \\
C \arrow[r] & G' & &
\end{tikzcd}$$

□

Note that the derived span construction is always possible since all $\mathcal{M}$-pullbacks exist in $\mathcal{G}_\perp$ Fact A.1.2 (the horizontal morphisms in derivations are in $\mathcal{M}$). The construction produces a result unique up to isomorphism and does not depend on the order of pullback constructions. Furthermore, it can be shown that consistency is a necessary and sufficient condition for the existence of extension diagrams.

Embedding Theorem. The Embedding Theorem establishes that a derivation can be extended to a larger one only when the extension morphism satisfies the consistency condition (see above). Furthermore, it shows the final graph of the extended derivation can be constructed as a pushout using the boundary/context graphs of the extension morphism (see Remark 9).

Theorem 6.2 (Embedding Theorem)**.** *Given a derivation $t : G \Rightarrow^* G_n$ and a morphism $k : G \rightarrow G' \in \mathcal{N}$ which is consistent with respect to t, then there is an extension diagram over t and k and morphisms $b_n : B \rightarrow G_n$ and $c_n : C \rightarrow G'_n$ such that the square $BCG_nG_n{}'$ is a pushout.*

Proof. The proof is very similar to the proof of [EEPT06, Theorem 6.14]. More specifically, the proof can be obtained from the proof in [EEPT06] by using initial pushouts over $\mathcal{N}$ (i.e. initial pushouts in the category $\mathcal{G}_\perp$) instead of initial pushouts in any $\mathcal{M}, \mathcal{M}'$-adhesive category. □

6.5 Confluence Analysis Algorithm

Next we give our semi-decision procedure for confluence based on symbolic critical pair analysis. The algorithm expects a finite set of left-linear rules $\mathcal{R}$ together with their associated set of symbolic critical pairs, and analyses each critical pair for strong joinability in order to output a confluence answer.

Informally, the algorithm proceeds as follows (algorithm 4). The main part of the computation is done per critical pair $T_1 \Leftarrow S \Rightarrow T_2$: construct all symbolic derivations from T_1 and T_2 that result in normal forms, store those in two sets PR_1 and PR_2, and for each pair of graphs in those sets try to compute an isomorphism compatible with the node's persistent nodes. If such an isomorphism exists, then flag the critical pair as strongly joinable. After all critical pairs are processed, if all are flagged as strongly joinable, then end the analysis as the input set of rules is confluent, otherwise output 'unknown'.

There are two main points of contention to the above description. First, when pairs of graphs are checked for an isomorphism (that is also compatible with the persistent set of nodes), the check may may be too conservative if the standard graph isomorphism of Chapter 2 is used, namely that all labels should be (syntactically) equal. We address this point first in subsection 6.5.1.

The second point is about the exploration of the derivation trees of T_1 and T_2. Since such exploration can be expensive, we show a sufficient condition that allows us to stop the search at so-called *persistent reducts* which are graphs to which the only applicable rules would delete persistent items. In other words, if a critical pair is strongly joinable, then necessarily there exist a pair of persistent reducts that are isomorphic. We explore this point in subsection 6.5.2.

Proposition 2 (Soundness)**.** *The Confluence Analyis algorithm is sound in that if the algorithm outputs 'confluent' for a given terminating set of left-linear rules $\mathcal{R}$, then $\mathcal{R}$ is confluent.*

Proof. The algorithm outputs 'confluent' when all given critical pairs are flagged as strongly joinable, which implies $\mathcal{R}$ is locally confluent by Theorem 6.1. Together with termination, this implies $\mathcal{R}$ is confluent. □

```
Input : A terminating set of left-linear rules $\mathcal{R}$
1  construct the symbolic critical pairs of each pair of rules in $\mathcal{R}$
2  let $CP$ be the set of all computed symbolic critical pairs
3  foreach $cp = (T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2)$ in $CP$ do
4      for $i = 1, 2$ do
5          construct all derivations $T_i \Rightarrow^*_{\mathcal{R}} X_i$ where $X_i$ is a persistent reduct
6          {let $PR_i$ be the set of all persistent reducts $X_i$}
7      end
8      foreach pair of graphs $(A, B)$ in $PR_1 \times PR_2$ do
9          if there exists a strong isomorphism $A \to B$ then
10             mark $cp$ as strongly joinable
11         end
12     end
13 end
14 if all critical pairs in CP are strongly joinable then
15     return “confluent”
16 else
17     return “unknown”
18 end
```

Algorithm 4: Confluence Analysis Algorithm

Studying the complexity of the given algorithm is non-trivial, largely because joinability analysis involves exploring the derivation trees of critical pairs, which can be unbounded. The exploration, as specified, can be interleaved with non-deterministically computing and comparing two persistent reducts, and continuing if the isomorphism check fails. Furthermore, the computation of critical pairs can be interleaved with joinability analysis rather than computing all critical pairs first. Last but not least, the derivation tree exploration can be restricted to check derivations only up to a certain length, e.g. a static bound as done in [Wel14]. Other kinds of bounding might be of use, for example a bound based on rule size or label size. However, introducing bounds reduces the potential applicability of the analysis.

6.5.1 Refinement 1: Graph Isomorphism

Since we consider graphs involving GP 2 label expressions, we relax the definition of isomorphism presented in Chapter 2 by replacing label equality with *equivalence*. In the following, let $\approx_E$ be the equivalence relation on GP2 expressions given by all the equations valid in GP 2's label algebra of integer arithmetic and list/string concatenation. Furthermore, since graph isomorphism is central to the discussion of joinability of critical pairs, we define how isomorphism relates to a critical pair's set of persistent nodes.

Definition 6.7 (E-isomorphism; ($\mathcal{P}$-)strong isomorphism)**.** Two labelled graphs G, H are E-isomorphic (denoted by $G \cong_E H$) if there exists a bijective premorphism $i : G \to H$ such that $l_H(i(x)) \approx_E l_G(x)$ for all nodes and edges of G. G and

H are $\mathcal{P}$-strongly isomorphic if $G \cong_E H$, $\mathcal{P} \subseteq V_G, \mathcal{P} \subseteq V_H$, and $i(v) = v$ for all $v \in \mathcal{P}$.

Here $\mathcal{P}$ is the set of persistent nodes of a critical pair, and when it is obvious from the context we drop it.

For an example of why a more general notion of isomorphism is needed, consider the schemata r_1: $\boxed{\texttt{m:n}} \Rightarrow \boxed{\texttt{m+n}}$ and r_2: $\boxed{\texttt{m:n}} \Rightarrow \boxed{\texttt{n+m}}$ which both match a node labelled with a list of two integers (`m` and `n`) but relabel the node to (syntactically) different expressions. The derivations $\boxed{\texttt{n+m}} \Leftarrow_{r_1} \boxed{\texttt{m:n}} \Rightarrow_{r_2} \boxed{\texttt{m+n}}$ represent a symbolic critical pair (conflict due to relabelling). The resulting graphs are normal forms, and isomorphic only if one considers the commutativity of addition.

Isomorphism checking is an integral part of joinability analysis. Since at the host graph level every label is taken from the concrete GP 2 label algebra without variables, checking for isomorphism ($\cong$) is decidable. However, when analysing graphs at the symbolic level, the problem of E-isomorphism ($\cong_E$) involves deciding validity of equations in Peano arithmetic. To the best of our knowledge, the problem is open for pure equations (no negation). Nevertheless, decidable fragments exist such as Presburger Arithmetic (with a double exponential lower bound on the worst-case time complexity), whose decision procedures can be used during the analysis of the shortest distances case study (see Chapter 7).

6.5.2 Refinement 2: Persistent Reducts

In the context of a critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ with a set of persistent nodes $\mathcal{P}$, a graph X derivable from T_1 or T_2 is a *persistent reduct* if the only rules of $\mathcal{R}$ that can be applied to X would delete an item of $\mathcal{P}$. Such graphs are useful when searching for joining derivations – one need not consider graphs derivable from such reducts because strong joinability requires the existence of all persistent items. However, it is not enough to nondeterministically compute a pair of persistent reducts and then compare them for strong joinability. Instead, one needs to consider *all* such reducts.

Consider the terminating set of rules in Figure 6.5. This system is non-confluent because of the derivations $A \overset{r_3}{\Leftarrow} T_1 \overset{r_1}{\Leftarrow} S \overset{r_1}{\Rightarrow} T_2 \overset{r_2}{\Rightarrow} D \overset{r_4}{\Rightarrow} \bullet$, which are two different normal forms. However, a confluence checker needs to search for strong joinability of critical pairs first. A strongly joinable critical pair $T_1 \Leftarrow_{r_1} S \Rightarrow_{r_1} T_2$ is given. All the nodes of S are persistent nodes ($\mathcal{P} = V_S$). The graphs T_1 and T_2 have multiple persistent reducts - T_1 reduces to A and B while T_2 reduces to C and D. The isomorphism $A \cong C$ is strong, $B \cong D$ violates the strictness condition, $A \ncong D$ and $B \ncong C$, thus a confluence checker needs to compare all persistent reducts for possible strong isomorphism until one is found.

Next we show that it is sufficient to consider only persistent reducts when trying to establish the strong joinability of a critical pair.

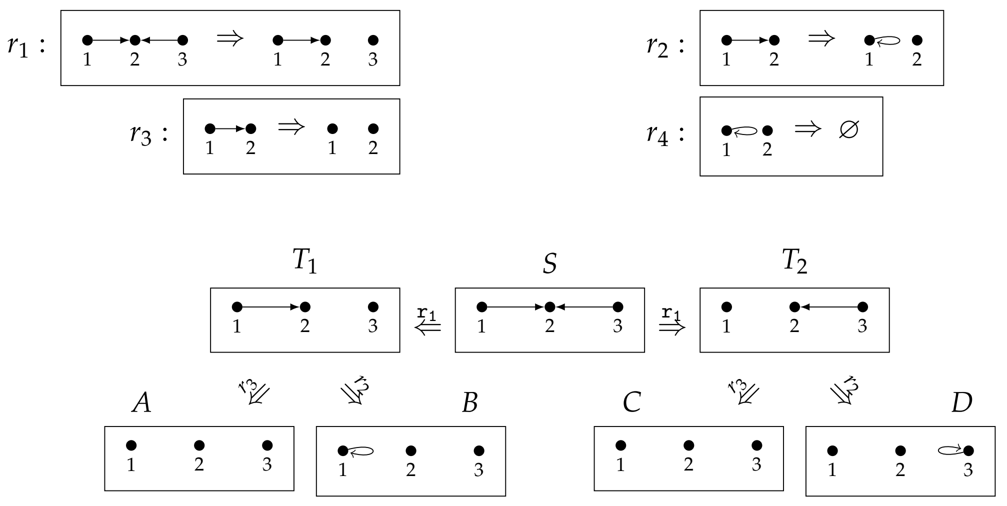


Figure 6.5: Joinability analysis with persistent reducts.

Proposition 3. *If a critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ with a persistent set of items P is strongly joinable, then there exist a pair of strongly isomorphic graphs X_1 and X_2 such that X_i is a persistent reduct derivable from T_i, $i = 1, 2$.*

Proof. Let PR_1 and PR_2 be the sets of persistent reducts of T_1 and T_2, respectively. Assume no pair of graphs in $PR_1 \times PR_1$ are strongly isomorphic. Because the critical pair is strongly joinable, according to Definition 6.4 there must be graph X derivable from T_1 and T_2 such that the track morphisms of the joining derivations are defined and equal on $\mathcal{P}$. If X is already a persistent reduct, then it directly contradicts the assumption, and if X is derivable from a persistent reduct, then it contradicts the fact that all persistent items $\mathcal{P}$ exist in X. Therefore, it must be the case that X is reducible to a persistent reduct X' ($\mathcal{R}$ is terminating). Then X' would be in both PR_1 and PR_2 - a contradiction. □

6.5.3 The issue with non-confluence

In the current formalization, the confluence algorithm does not determine non-confluence. This is due to the limitations of symbolic rewriting: not every host graph derivation can be represented by a symbolic derivation, as previously shown in Figure 6.1b. This limitation means that normal forms at symbolic level need not be instantiated to normal forms at the host graph level.

However, in certain restricted cases the algorithm would be able to report non-confluence: it shows non-confluence when the compared graphs are normal forms (rather than persistent reducts) that are variable-free. This result is of limited applicability since the algorithm prioritises establishing confluence, and thus only explores derivation trees up to persistent reducts. A more elaborate algorithm would go for a second phase trying to establish non-confluence by non-deterministically computing pairs of normal forms and checking whether they are non-isomorphic.

Proposition 4. *Given a symbolic critical pair $T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2$ and a pair of non-*

isomorphic variable-free normal forms A and B such that $A \Leftarrow^*_{\mathcal{R}} T_1 \Leftarrow_{\mathcal{R}} S \Rightarrow_{\mathcal{R}} T_2 \Rightarrow^*_{\mathcal{R}} B$, the set of rules $\mathcal{R}$ is not confluent.

Proof. Since A is a normal form w.r.t. $\Rightarrow_{\mathcal{R}}$ and variable-free, that means it must also be a normal form w.r.t $\Rightarrow_{\mathcal{R}}$, and the same holds for B. Since $A \ncong B$, then every instance of the given critical pair $A \Leftarrow^*_{\mathcal{R}} T_1^{\lambda} \Leftarrow_{\mathcal{R}} S^{\lambda} \Rightarrow_{\mathcal{R}} T_2^{\lambda} \Rightarrow^*_{\mathcal{R}} B$ is a counterexample to confluence. □

6.6 Related Work

In this section, we look at relevant literature, regarding both the theoretical aspects of confluence analysis and the practical considerations involving specifying a confluence algorithm.

Symbolic Graphs. SyGrAV [Dec17, DKL+16, KDL+15] is a tool prototype aimed at formal verification of symbolic graph transformations. The critical pair component performs construction of critical pairs for symbolic rules and also subcommutativity analysis (0-1 step joinability). The tool supports attributes that are integers, reals, bitvectors (with finite domains) by means of the SMT-solver Z3 [dMB08], but without support for strings in labels nor (non-nested) application conditions nor control constructs. It sets itself apart from other tools like AGG [RET11] because (1) symbolic graphs are more general than attributed graphs (a single symbolic graph represents a set of attributed graphs); (2) it performs 0-1-step joinability analysis of critical pairs rather than stopping after critical pair construction; (3) it uses the powerful SMT solver Z3 to handle a large variety of attributes/formulas. Furthermore, the framework avoids the need for unification during critical pair analysis, but instead records the systems of label equations as part of the logic formulas attached to symbolic graphs.

Comparing with the approach presented with this thesis, we can see several differences. First, we allow arbitrary step joinability by exploring the derivation trees of computed critical pairs. To help with the inherent complexity, we showed one need only consider persistent reducts, which cuts off part of the relevant computation. Second, although symbolic graph transformation is more general than GP 2 rewriting and supports more data types of attributes/labels, we explicitly fix our label algebra of heterogeneous lists of strings and integers, which already has a non-trivial matching problem. However, we also use the solver Z3, specifically for our case studies (see Chapter 7). Last but not least, we do not defer dealing with overlapping labels. In our setting, if the system of equations for a given overlap has no unifier then no critical pairs induced by the given overlap are constructed.

Confluence Algorithms. Similar to the confluence algorithm presented in Section 6.5 was given in [Plu94]. The paper gives a decision procedure for confluence of term graph rewriting, a form of term rewriting where terms are represented as directed acyclic graphs. In that setting, confluence is a decidable property, so it is safe for an algorithm to perform joinability analysis by nondeterministically computing normal forms and checking for a strong isomorphism. That algorithm is an extension of the classical confluence decision procedure for term rewriting of Knuth and Bendix, which is justified by the Critical Pair Lemma of Huet [Hue80].

The situation with (hyper)graph rewriting is more complicated as confluence is undecidable in general [Plu93]: it is not necessary, given a confluent graph rewriting system, for all critical pairs to be strongly joinable. The issue is that if all critical pairs are joinable but only some are strongly joinable, then confluence cannot be decided; making the strong joinability check sufficient but not necessary. (The graph-specific result was given in [Plu05]). When graph transformation is extended with so called *covers* then confluence becomes decidable as shown in [Plu10]; that paper also specifies a confluence algorithm based on nondeterministic reduction to normal forms. A similar notion of coverability is that of using graph transformation with *interfaces*, where confluence is also decidable.

6.7 Summary

In this chapter, we have:

- developed the notion of *symbolic rewriting* for graphs labelled with GP 2 expressions, to facilitate the joinability analysis for critical pairs;
- defined and studied *strong joinability* as a condition that is sufficient for establishing the confluence of a set of terminating left-linear GP 2 rule schemata;
- proven the Local Confluence Theorem based on the strong joinability of all critical pairs;
- defined and studied an analysis algorithm focusing on establishing confluence;
- presented two improvements to the algorithm that increasing efficiency (using persistent reducts) and improve accuracy (relaxing isomorphism requiring label equivalence rather than equality).

Chapter 7

Confluence Case Studies

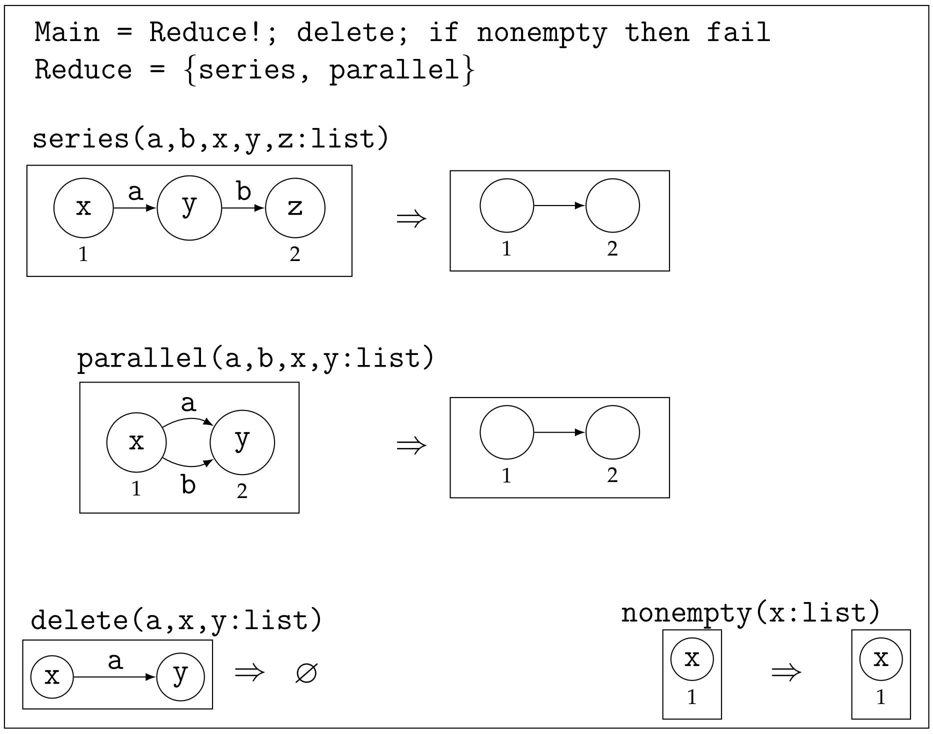
In the previous chapters, we have described (1) how to construct critical pairs for pairs of schemata; and (2) how to analyse critical pairs for strong joinability in the context of a confluence checker. Up to this point, we have demonstrated the concepts, definitions and theoretical results on small and partial examples. This is fine for the purpose of understanding the particular technical details. But what we have not yet shown is how our ideas can be put together to check interesting existing graph programs for confluence.

In this chapter we revisit the graph programs of Chapter 2, and apply our previously developed ideas to check whether they are confluent or not. We focus on using our notions of critical pairs and joinability, but later consider how the results of the analysis can be used to inform a programmer who is interested to modify their program to achieve confluence without compromising semantics.

For each case study, we use the following general structure. We revisit the problem and its GP program in sufficient detail, then list all critical pairs, grouping them by rules involved and by conflict type. Afterwards we perform joinability analysis on each critical pair. Finally, based on the joinability results, we attempt to give a confluence answer for the original GP program. If the program is not confluent, we give a specific counter example, and discuss how the program can be 'repaired' to obtain a confluent version.

The chosen examples are existing GP programs: Series-parallel Recognition (Section 7.1), concerned with a specific class of graphs; Shortest Distances (Section 7.2), which computes the shortest path in a graph from a given source node to all other reachable nodes; Computing 2-colouring (Section 7.3), concerned with computing the 2-colouring of a graph (if it exists). All of these programs (or parts of them) are meant to be confluent, so it is important to try and prove and refute whether they are so. Furthemore, the Vertex Colouring (Section 7.4) case study shows the program for computing any colouring of a graph rather than a 2-colouring, and it is a highly non-deterministic program which we can prove to be non-confluent.

The Shortest Distances case study was presented in [HP17a], and its joinability analysis discussion appeared in the long version only.



```
Main = Reduce!; delete; if nonempty then fail
Reduce = {series, parallel}
```

Figure 7.1: A program that checks whether a graph is series-parallel or not, repeated from Subsection 2.2.3.1,.

7.1 Series-Parallel Graphs

First, we return to the recognition of series-parallel graphs introduced in Subsection 2.2.3.1, and use our critical pair analysis techniques to check whether it is confluent or not. The result of the analysis is that the program as originally given in [Plu16] is not confluent due to edge labels, and give a confluent version of the program that deals with this problem.

The program implementing the recognition of series-parallel graphs is given in Figure 7.1. Given an (input) host graph G, the program first applies the set of reduction rules as long as possible, resulting in some graph H. To determine whether H has the correct shape, the program first deletes the predefined shape $\bigcirc\!\!\longrightarrow\!\!\bigcirc$, then checks whether the result is the empty graph. If either the deletion or the non-empty checks fail, then the program fails. In this context, termination of the program with a proper graph means the input graph G is series-parallel, and failure means G is *not* series-parallel.

Termination. The program always terminations due to the following reasons. The first part of the program applies size-reducing rules as long as possible: `series` reduces the number of nodes, and `parallel` reduces the number of edges. The `delete` rule is applied (at most) once so it also terminates. The final check ap-

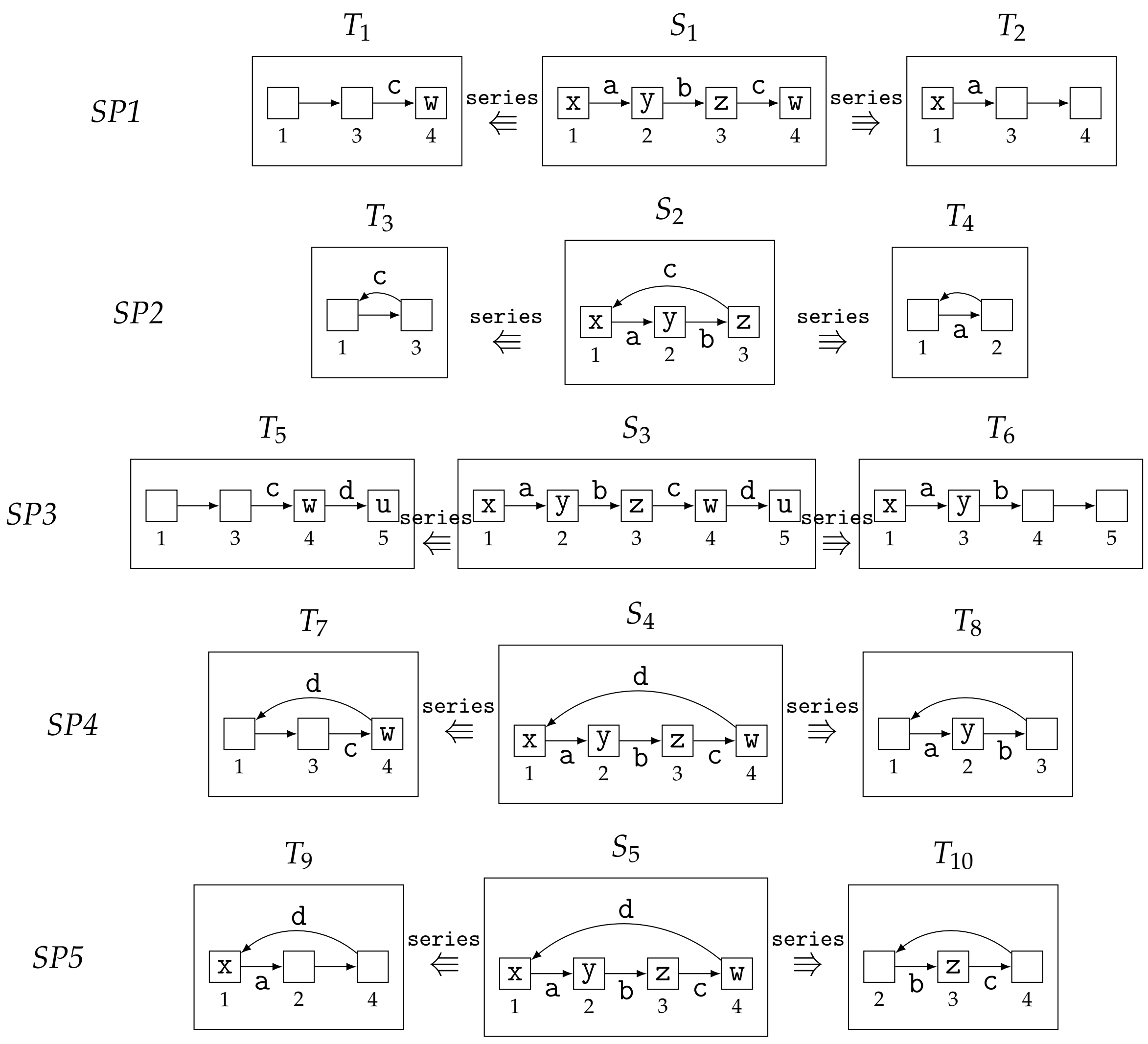


Figure 7.2: Series-parallel critical pairs involving `series` with itself.

plies the `nonempty` rule at most once so it also terminates. Since the sequence of terminating programs terminates, the overall program also terminates.

Symbolic critical pairs. In figures Figure 7.2, Figure 7.3 and Figure 7.4, we list all symbolic critical pairs of `Reduce = {series, parallel}`. Grouping is done by rules involved.

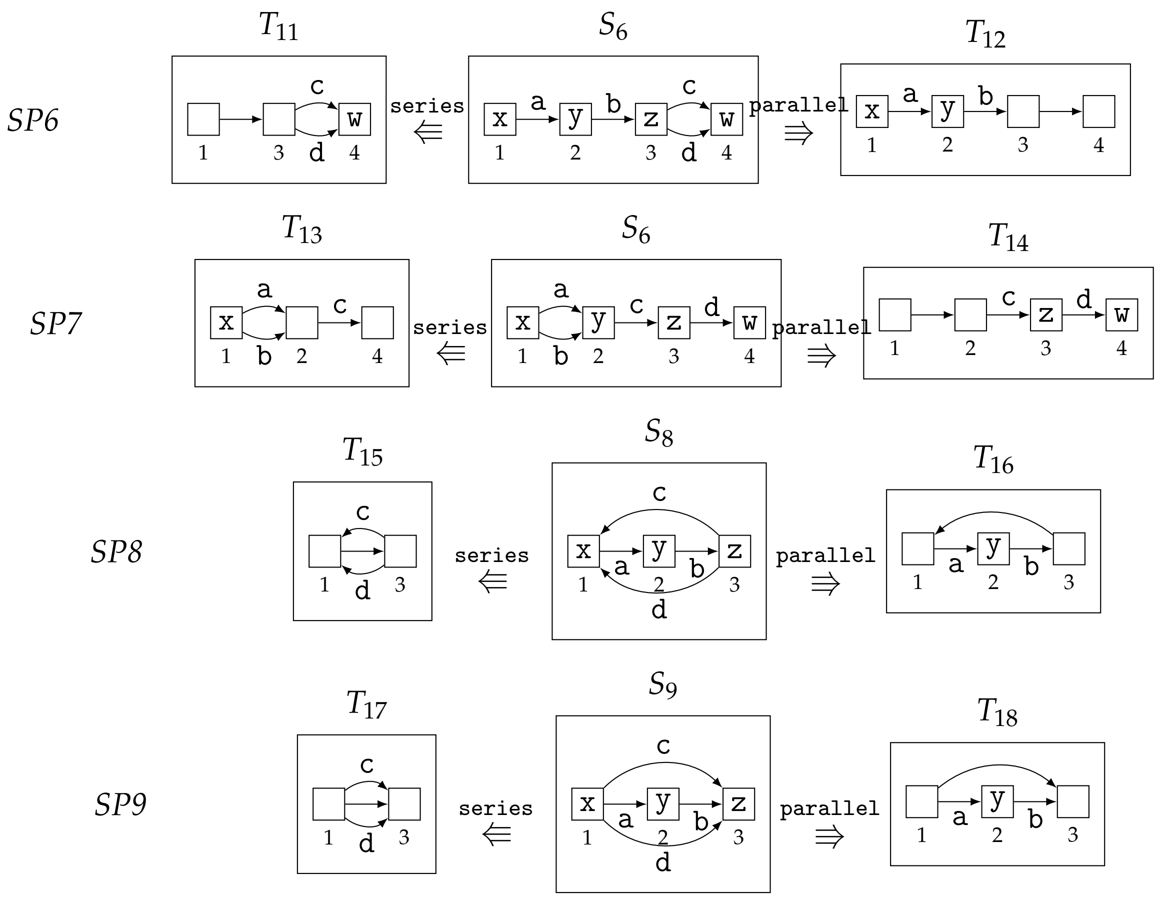


Figure 7.3: Series-parallel critical pairs involving both rules `series` and `parallel`.

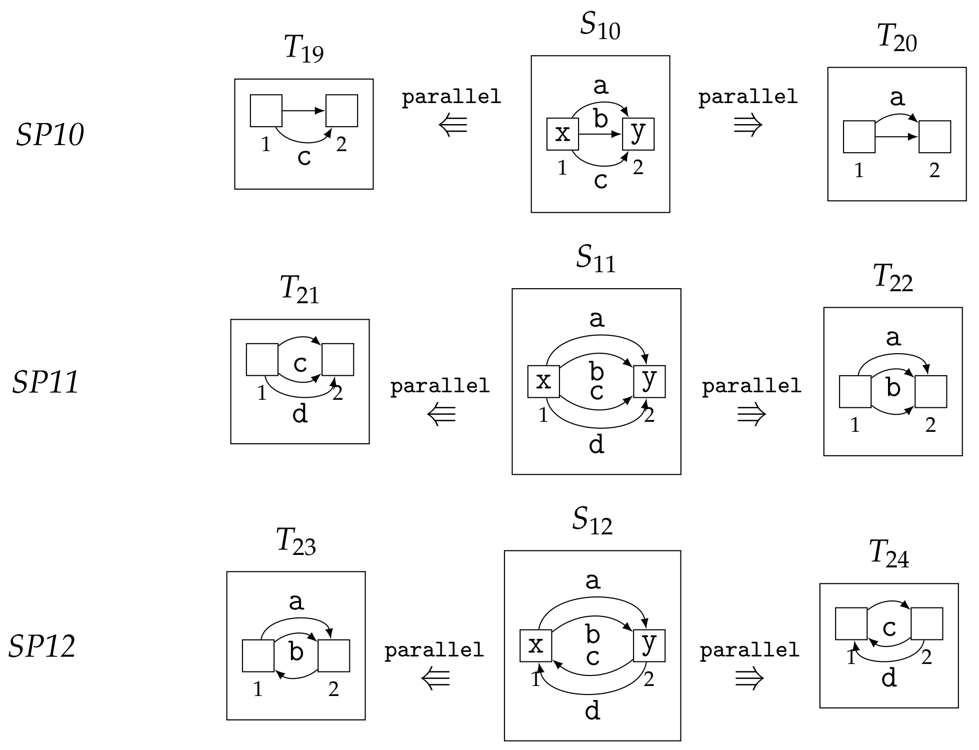
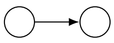


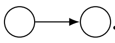
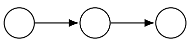
Figure 7.4: Series-parallel critical pairs involving `parallel` with itself.

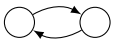
Joinability Analysis. Next, we give the joinability analysis of each critical pair. As done previously, we group the critical pairs depending on which rules are involved. All critical pairs are strongly joinable, except critical pair SP2. This critical pair is not joinable and we are able to instantiate it to a confluence counter example.

<u>`series` $\times$ `series`</u>

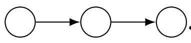
SP1 *strongly joinable* - both result graphs reduce to 

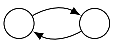
SP2 *not joinable* - the result graphs are normal forms, there are 2 bijective premorphisms $iso_1 : T_1 \to T_2 = \{1 \to 1, 3 \to 2\}$ and $iso_2 : T_1 \to T_2 = \{1 \to 2, 3 \to 1\}$ which induce the systems of label equations $\{\mathtt{a} = \mathtt{empty}, \mathtt{c} = \mathtt{empty}\}$ and $\{\mathtt{a} = \mathtt{c}, \mathtt{empty} = \mathtt{empty}\}$ that are universally quantified. (Here `empty` stands for the empty list). If either are true, then iso_1 or iso_2 is an isomorphism and the critical pair is joinable. However, neither conjectures are true. Finding values for the variables such that $\mathtt{a} \neq \mathtt{c}$ produces a concrete counter example to confluence, e.g. $\mathtt{a} = \mathtt{1}, \mathtt{c} = \mathtt{3}$ (see Subsection 7.1).

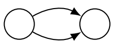
SP3 *strongly joinable* - both result graphs reduce to . The joining derivations delete one of the protected nodes (3). However, when we look at the full derivation tress, both reduce to the graph  which contains all persistent nodes. Therefore the pair is strongly joinable.

SP4/5 *strongly joinable* - both result graphs reduce to 

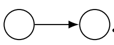
<u>`series` $\times$ `parallel`</u>

SP6/7 *strongly joinable* - graphs reduce to . Note that this graph is a *persistent normal form*, similar to the critical pair SP3 above.

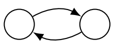
SP8 *strongly joinable* - graphs reduce to 

SP9 *strongly joinable* - graphs reduce to 

<u>`parallel` $\times$ `parallel`</u>

SP10 *strongly joinable* - graphs reduce to . Note the graphs of the critical pair are isomorphic ($T_{19} \cong T_{20}$) only up to the label equality $\{\mathtt{a} = \mathtt{c}\}$, which is satisfiable but not valid.

SP11 *strongly joinable* - same as above.

SP12 *strongly joinable* - graphs reduce to 

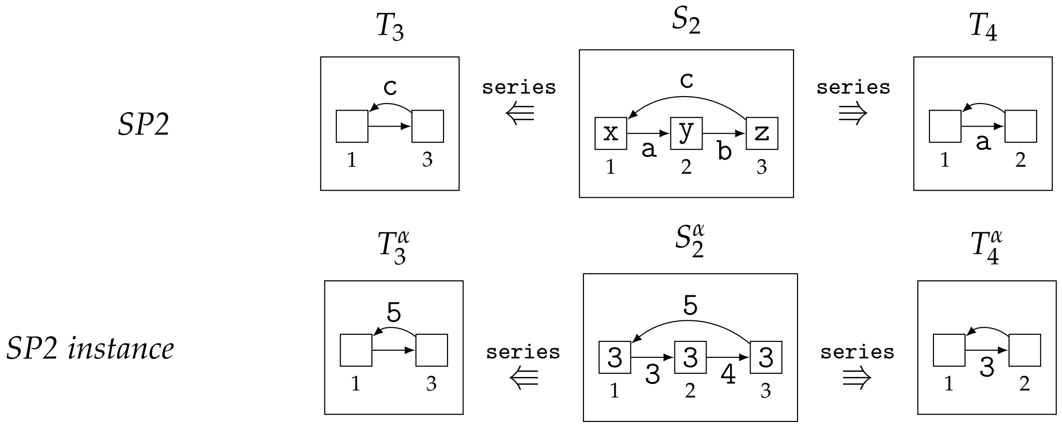
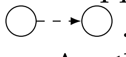


Figure 7.5: Confluence counter example: an instance of SP2.

Confluence Result. Since our analysis found a symbolic critical pair that is *not joinable* (SP2), then one would imagine it would be possible to construct a concrete counter example to confluence. Concrete in this setting means the graphs involved are labelled over concrete data, i.e. GP 2 host graphs.

Indeed it is possible to instantiate the critical pair as in Figure 7.5 with two normal forms that are not isomorphic. The instantiation is by the assignment $\alpha = \{\mathtt{x} \mapsto \mathtt{3}, \mathtt{y} \mapsto \mathtt{3}, \mathtt{z} \mapsto \mathtt{3}, \mathtt{a} \mapsto \mathtt{3}, \mathtt{b} \mapsto \mathtt{4}, \mathtt{c} \mapsto \mathtt{5}\}$, where we have made sure that $\alpha(\mathtt{a}) \neq \alpha(\mathtt{c})$.

Therefore, Reduce is *not confluent*. The problem is that the program only handles edge labels which are matched explicitly by Reduce.

Confluence repair. There are several ways to repair the program w.r.t. confluence. One option is to have a preprocessing stage before Reduce that removes all edge labels. This does not affect the original algorithm since the series-parallel property of a graph is not concerned with labels. Such removal can be done by the unlabel schema introduced in Chapter 4. (Note the schema marks matched edges as dashed to avoid non-termination.) The Reduce rules can then be simplified to not contain variables, together with final step of the algorithm which checks that, after applying Reduce as long as possible, the resulting graph is isomorphic to .

Another option is to examine the non-joinable critical pair and introduce *new rules* to Reduce that make it joinable. In this case it would be a rule that takes a 2-cycle graph and removes its labels. However, adding more rules may introduce extra critical pairs. This idea is known as Knuth-Bendix completion in term rewriting. See e.g. [Hue80, Hue81] or the textbook [BN98].

This idea is realised in Figure 7.6 where we use the rule unlabel applied as long as possible before the start of the algorithm. This changes the analysis as the rules of Reduce are schemata without variables. The critical pairs of Reduce are as before except all the variables are replaced with empty lists, making the confluence counter example disappear since it relies on instantiating variables in a particular way. As extra work, we need to consider the critical pairs of `unlabel` and only of that schema with itself, which were given in Figure 4.5. These extra critical pairs are strongly joinable also since the rule can be applied to each pair of result graphs. Therefore, the repaired Series-Parallel program is confluent as it is a composition of confluent sub-programs. Note however our analysis does not cover the last step of the program, the `if` subprogram, as it would require confluence reasoning beyond critical pairs.

```
Main = unlabel!; Reduce!; delete; if nonempty then fail
Reduce = {series, parallel}
```

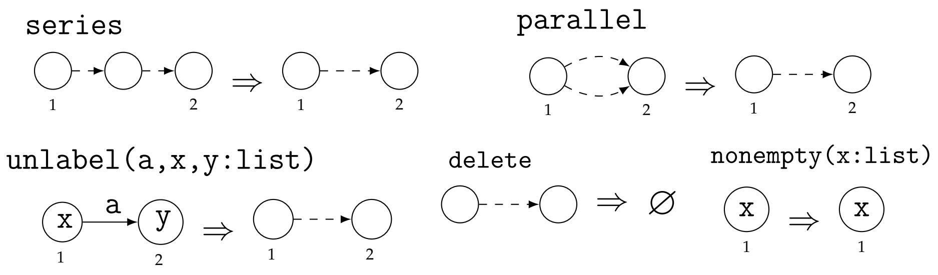
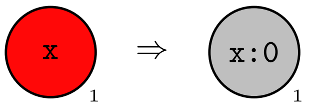


Figure 7.6: Confluent version of Series-Parallel program.

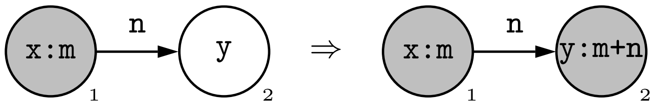
Case Study Summary. After performing our analysis for the Series-Parallel GP 2 program, we learned the following:

- confluence analysis can be effectively done by means of symbolic critical pair construction and joinability analysis;
- even when a given program is not confluent, a counter example can be constructed by instantiating a chosen (non-joinable) critical pair;
- it is non-trivial to repair programs as it may introduce new rules and thus new critical pairs.

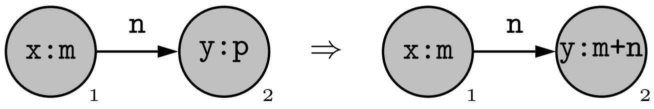
```
Main = init; {add, reduce}!
init(x: list)
```



```
add(x, y: list; m, n: int)
```



```
reduce(x, y: list; m, n, p: int)
```



```
where m + n < p
```

Figure 7.7: Shortest Distances program, repeated from Subsection 2.2.3.2.

7.2 Shortest Distances

The next case study is our program for computing shortest distances, given in Subsection 2.2.3.2. The shortest distances problem is about calculating the paths between a given node (the *source* node) and all other nodes in a graph such that the sum of the edge weights on each path is minimized. An assumption made is that there is no *negative cycle* (a cycle whose edge weights sum to a negative value) that is reachable from the source, in which case there is no shortest path.

A GP 2 program that implements the Bellman–Ford algorithm is shown in Figure 7.7. Distances from the source node are recorded by appending the distance value to each node's label. Nodes *marks* are used: the source node is red, visited nodes are gray, and unvisited nodes are unmarked. Given an input graph G with a unique source node and no negative cycle, the program initializes the distance of the source node to 0. The `add` rule explores the unvisited neighbours of any visited nodes, assigns them a tentative distance and marks them as visited to avoid non-termination. The `reduce` rule finds occurrences of visited nodes whose current distance is higher than alternative distances, i.e. only when the application condition ($\mathtt{m} + \mathtt{n} < \mathtt{p}$) is satisfied by the schema instantiation. The program terminates when neither `add` or `reduce` rules can be further applied.

However, since rule application is non-deterministic, different graphs may result from a program execution. The above algorithm is correct only if the loop {`add,reduce`}`!` is confluent. In the absence of a full program verification, a programmer may want to check that this loop indeed returns unique results.

The program uses several of GP's features. List expressions, e.g. `x:m`, are used to append the node's distance during computation. (Due to list expressions, other approaches to critical pairs will not attempt to analyse these rules.) Marks are used to record visited and unvisited nodes. Since marks are separate to the list expression of a label, we consider them as constants when computing critical pairs. Last but not least, the application condition of `reduce` plays a vital role in the program's correctness. Although our analysis does not deal with application conditions in general, we can give specific intuition of how to include these specific conditions in the analysis. We can do so since the application condition of `reduce` involves integer expressions and comparison, and hence falls in the scope of the SMT-solver Z3.

Termination. The program starts with the single application of `init` which is executed at most once, and the program fails if it fails. Otherwise, it proceeds with application `add` and `reduce` as long as possible. To see this terminations, consider the opposite: that there is an infinite derivation sequence involving both rules. Such a sequence cannot contain an infinite number of `add` applications as the rule reduces the number of unmarked nodes in the graph and the other rules do not increse the number of such nodes. Hence, at some point there are no more matches for that rule, from which point there must only be (an infinite number of) matches for `reduce`. But the `reduce` rule, when applied as long as possible on its own, terminates under the assumption of no negative cycles. This is a contradiction and therefore the program must terminate.

Symbolic Critical Pairs. The symbolic critical pairs of the above program are given in Figure 7.8 and Figure 7.9. There are 7 critical pairs in total: two between `add` and itself (SD1/2), one between `add` and `reduce` (SD3), and four between `reduce` and itself (SD4-7). All of the conflicts are due to relabelling of a common node. Note that due to the semantics of GP 2 marks (marked cannot match unmarked), other conflicts are not possible. Variables have been renamed where necessary.

The critical pairs SD1/2 are between the rule `add` with itself where an unvisited node can get initialized with different distance values, either from 2 neighbouring nodes or from the same node but different (parallel) edges. In SD3 the distance of a node in a path is used in different ways: either to initialize the distance of a neighbouring node (using `add`), or to have its own distance updated (using `reduce`). Application conditions is recorded as part of the critical pair. The critical pairs SD4/5 represent a conflict of `reduce` with itself where a node may get different updated distance values depending on which path is chosen, similar to SD1/2. SD6 involves a 2-cycle where either node gets its distance updated by `reduce`. SD7 involves a sequence of three nodes, similar to SD3.

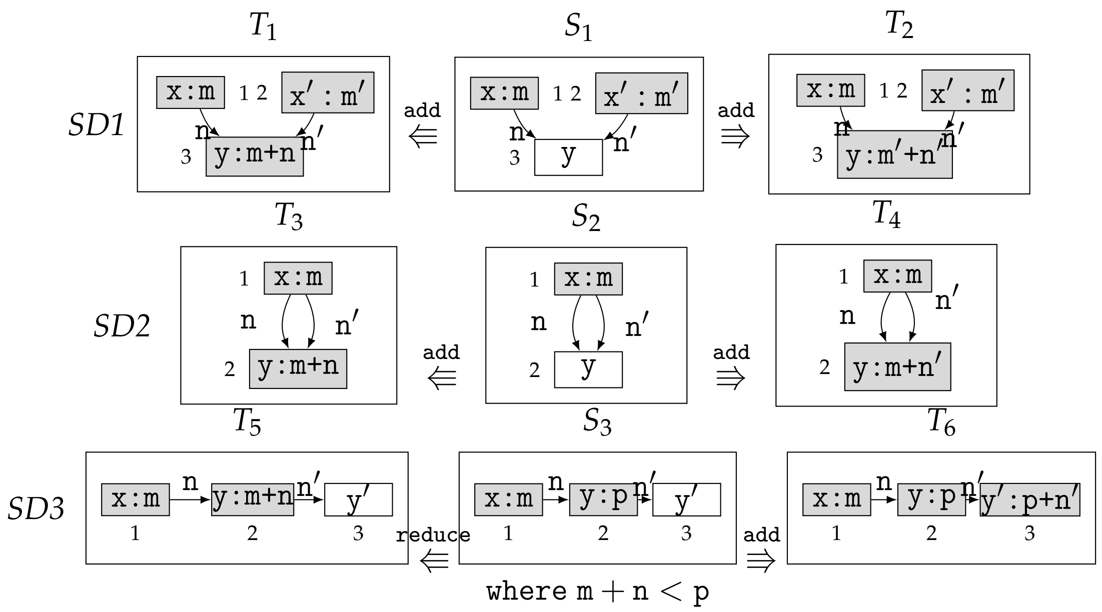


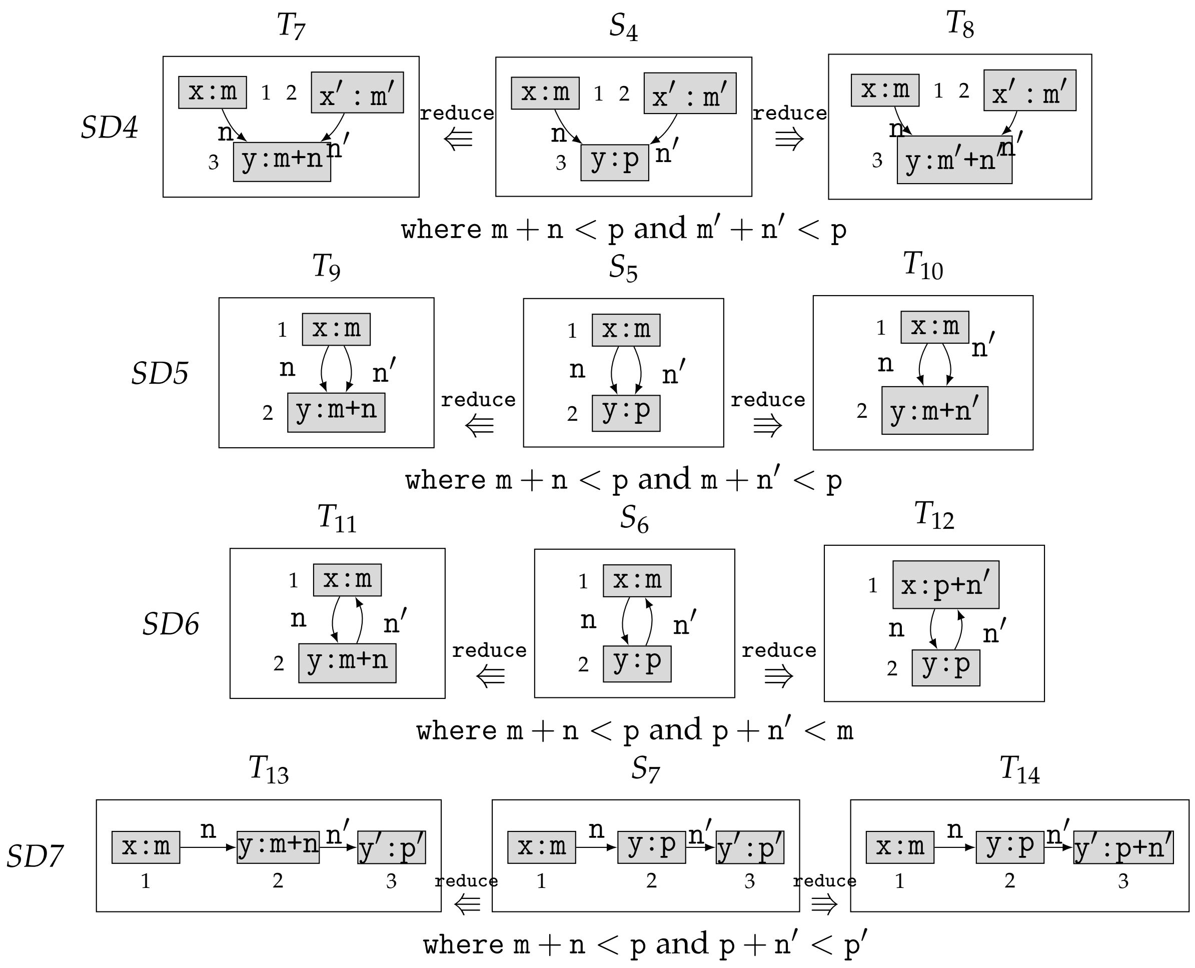
Figure 7.8: Shortest Distances critical pairs involving add.

Joinability Analysis. In the following, we give the strong joinability analysis for each critical pair of Figure 7.8 and Figure 7.9. The result of the analysis is that all critical pairs are strongly joinable except the 2-cycle critical pair SD6 whose label condition is unsatisfiable assuming non-negative cycles and the semantic argument that both schemata do not modify edge labels. (Without using this information, the critical pair is not joinable.) Hence the loop {add, reduce}! is confluent.

An interesting practical aspect of joinability is that it involves, in most cases, checking label equivalences for validity. (We check for validity rather than satisfiability since we need that all instances of a strongly joinable critical pair to be strongly joinable rather than at least one.) For this purpose, we use the SMT solver Z3 [dMB08]. It provides support for (linear) integer arithmetic, arrays, bit vectors, quantifiers, implications, etc.

For the critical pair SD1, the result graphs T_1 and T_2 are isomorphic only if the label equivalence $m + n = m' + n'$ is valid, which it is not (encoded as a forall expression in Figure 7.10a where variables have been renamed). The analysis proceeds by applying reduce to both T_1 and T_2, and the semantics of the reduce condition (containing comparison of integer expressions) guarantees a strong isomorphism between the results. Note that reduce is necessary for the joining derivations, meaning the rule add is not confluent on its own. The analysis of SD2 proceeds in a similar way as SD1 with the same conclusion.

For SD3, one needs to check *implications* between conditions to ensure strong joinability between a pair of derivable graphs. An implication that shows up during the analysis is shown in Figure 7.10b which Z3 reports to be valid. Therefore the critical pair is strongly joinable. For the critical pairs SD4/5, their result graphs are exactly the result graphs of SD1/2, which greatly simplifies the analysis. The



SD4
T_7
S_4
T_8
reduce
where $\mathtt{m}+\mathtt{n} < \mathtt{p}$ and $\mathtt{m}'+\mathtt{n}' < \mathtt{p}$
SD5
T_9
S_5
T_{10}
where $\mathtt{m}+\mathtt{n} < \mathtt{p}$ and $\mathtt{m}+\mathtt{n}' < \mathtt{p}$
SD6
T_{11}
S_6
T_{12}
where $\mathtt{m}+\mathtt{n} < \mathtt{p}$ and $\mathtt{p}+\mathtt{n}' < \mathtt{m}$
SD7
T_{13}
S_7
T_{14}
where $\mathtt{m}+\mathtt{n} < \mathtt{p}$ and $\mathtt{p}+\mathtt{n}' < \mathtt{p}'$

Figure 7.9: Shortest Distances critical pairs involving only `reduce`.

joint label condition concerns the integer variable p which disappears during rule application, effectively having no effect on the analysis.

The critical pair SD6 is different. Its label condition is satisfiable only when the sum of the edge labels is negative ($\mathtt{n}+\mathtt{n}' < 0$), which is not possible under the assumption of no negative cycles and the observation that no rules modify edge labels. Without this semantic information, it is possible to instantiate the critical pair to a concrete graph with non-isomorphic normal forms, and thus obtain an example of non-confluence.

Joinability of SD1, SD2, SD4 and SD5 (strongly joinable). Consider the graphs T_1 and T_2 of SD1 (top of Figure 7.11): the identity mapping is bijective premorphism that commutes on the persistent nodes of the critical pair (which are all of the nodes 1,2 and 3). However, it is not label preserving under equivalence, meaning that the following conjecture is not valid: $\forall \mathtt{y},\mathtt{m},\mathtt{m}',\mathtt{n},\mathtt{n}' \bullet \mathtt{y} : (\mathtt{m}+\mathtt{n}) = \mathtt{y} : (\mathtt{m}'+\mathtt{n}')$. Its invalidity can be checked by Z3 as explained above.

Next, we explore the derivation trees of T_1 and T_2. We apply `reduce` to T_1 to

```
(define-fun T1_T2() Bool
        (forall ((m1 Int) (m2 Int)
                         (n1 Int) (n2 Int))
            (= (+ m1 n1) (+ m2 n2)) ))
(assert T1_T2)
```

(a) Label equivalence example for SD1.

```
(define-fun T77_T888 () Bool
        (forall ((m Int) (p Int)
                         (n1 Int) (n2 Int))
                (=> (< (+ m n1) p)
                        (< (+ m n1 n2) (+ p n2)) )))
(assert T77_T888)
```

(b) Implication checking for SD3.

Figure 7.10: Z3 code for label equivalence analysis of shortest distances.

obtain graph T_1'. (The rule add is not applicable to either graph due to matching restrictions of marks.) This application has the effect of relabelling node 3 (the common node) to having distance $m' + n'$. The application condition of the rule is $m' + n' < m + n$. Again we want to look for an E-isomorphism $T_1' \to T_2$, and in this case the identity premorphism preserves labels — all graph labels are syntactically equal. However, the isomorphism is conditional on the existence of T_1'. Indeed, if one instantiates the critical pair to a concrete conflict that doesn't satisfy reduce's application condition, then the symbolic joining derivations cannot be instantiated to concrete ones. Therefore, $T_1' \cong_E T_2$ only when $m' + n' < m + n$.

Exploring the derivation tree of T_2 yields similar results. We can summarize the situation as follows:

- $T_1 \cong_E T_2$ only when $m + n = m' + n'$
- $T_1' \cong_E T_2$ only when $m + n < m' + n'$
- $T_1 \cong_E T_2'$ only when $m + n > m' + n'$
- $T_1' \cong_E T_2'$ only when $m + n > m' + n'$ and $m + n < m' + n'$

The symbolic critical pair is strongly joinable under certain label conditions, and the strong joinability witnesses are different depending on which condition is satisfied. At a first glance, it may be possible that none of the above conditions is satisfied by an instantiation of the critical pair. Only when we go into the semantics of comparison can it be deduced that there is no such situation. This can proven by Z3 for this specific system of conditions, so we can be sure at least one pair of graphs is strongly isomorphic. As a result, the critical pair SD1 is strongly joinable.

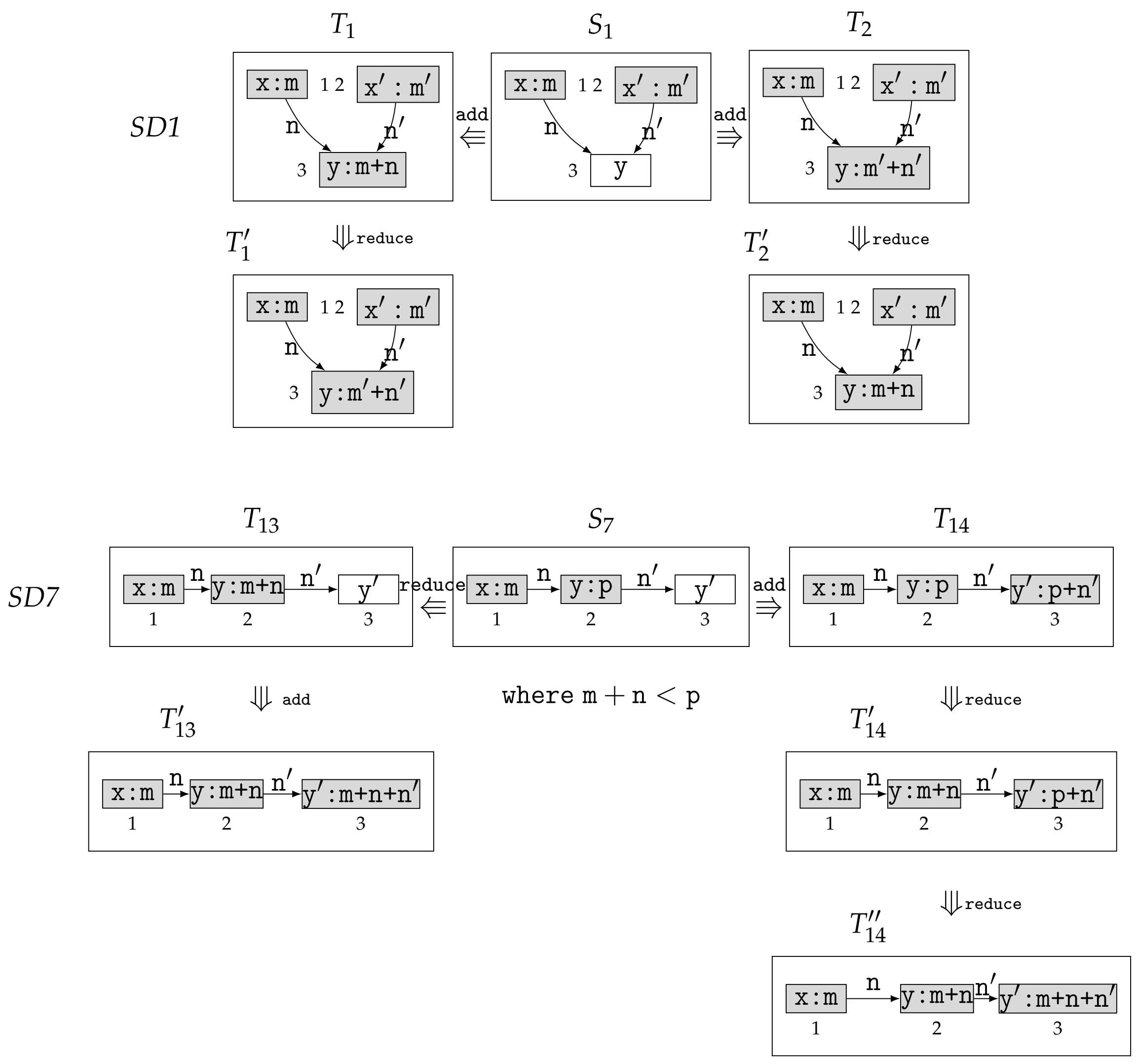


Figure 7.11: Joinability for Shortest Distances program.

The analysis for the critical pair SD2 proceeds in exactly the same way - there is one less node to consider (meaning less variables), and exactly the same conclusion is reached as SD1. For SD4, it is a conflict between two `reduce` applications.It turns out that the two result graphs are exactly the result graphs of SD1, i.e. $T_7 \cong T_1$ and $T_8 \cong T_2$, which greatly simplifies analysis. Note that the joint label condition concerns variable p which disappears in T_7 and T_8, effectively having no effect on the joinability analysis. For SD5, its result graphs are exactly the result graphs of SD2, and thus again the analysis is exactly the same.

Joinability of SD3 and SD7 (strongly joinable). Consider the derivation trees of T_5 and T_6, shown in Figure 7.11. We can summarize the analysis as follows:

- $T_5' \cong T_6$ only when $m + n = p$ and $m + n + n' = p + n'$
- $T_5' \cong T_6'$ only when $m + n + n' = p + n'$ and $m + n < p$

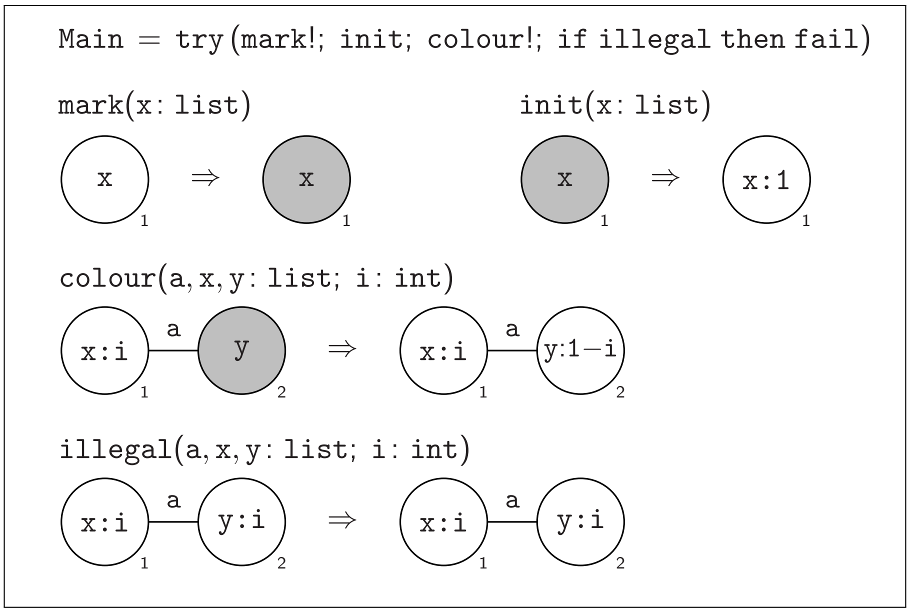
- $T_5' \cong T_6''$ only when $m + n + n' < p + n'$

The first and second conditions are invalid in the context of critical pair condition $m + n < p$. In the last case, the graphs T_5' and T_6'' have syntactically the same labels, and the given condition ensures $T6''$ exists. It can be shown that the critical pair condition on S_4 *implies* that condition of $T_5' \cong_E T_6''$. We encode this situation as Figure 7.10b which Z3 reports to be valid. In other words, every critical pair instance can be joined by instantiating the derivations $T_5 \Rrightarrow T_5' \cong_E T_6'' \Lleftarrow T_6' \Lleftarrow T_6$. Therefore, SD4 is strongly joinable.

For SD7, we can apply $T_{13} \overset{\texttt{reduce}}{\Rrightarrow} T_5$, and also we have $T_{14} \cong T_6$, meaning the analysis of SD3 is repeated (under the extra condition on the variable p$'$ which disappears in T_5 and T_6), and thus again obtaining strong joinability.

Case Study Summary In summary, we have learned the following:

- sometimes it is useful for a programmer to intervene during confluence analysis in the form of an oracle that discharges critical pairs with unsatisfiable label conditions;
- joinability analysis needs to consider there may be several different witnesses depending on specific labels and/or existence conditions;
- an SMT solver such as Z3 can be used to check equalities of labels, to instantiate variables with concrete values to obtain confluence counter examples, and to prove implications between label conditions.



```
Main = try(mark!; init; colour!; if illegal then fail)
```

Figure 7.12: 2-colouring program, repeated from Subsection 2.2.3.3.

7.3 2-colouring

This example is concerned with computing a vertex 2-colouring of an input graph if such a colouring is possible. To colour the nodes of a graph means to assign colours (as labels) to each node such that no two adjacent nodes have the same colour. We introduced the program for computing a 2-colouring in Subsection 2.2.3.3, repeated in Figure 7.12. The program assumes the input is a connected unmarked graph, and terminates with a valid 2-colouring if possible. If no such colouring exists, the program returns the original input graph.

As a brief summary, the program marks all nodes as unvisited (grey mark), then non-deterministically initializes a node with the colour 1 and marking it as visited (by removing its grey mark). The program then explores the neighbours of visited nodes by assigning them opposing colours. Specifically to this program, to avoid complications with the bidirectional edge of `colour` we assumed it is a set of two rules where the edge is in the forward (rule `colour1`) or backward direction (rule `colour2`).

Termination. The program terminates due to the following reasons. The `mark` rule reduces the number of unmarked nodes, so applying it as long as possible terminates. The `init` and `illegal` rules are applied at most once, so they will always terminate. The `colour` rule set reduces the number of marked nodes, and since there are no infinite number of marked nodes given the assumption of unmarked input and the property of `mark` to shade all the nodes of the input graph exactly once, it follows that the `colour!` sub-program also terminates. The sequencing of terminating sub-programs also terminates, and wrapping a terminating program in a `try` block does not affect termination, it follows that the program terminates.

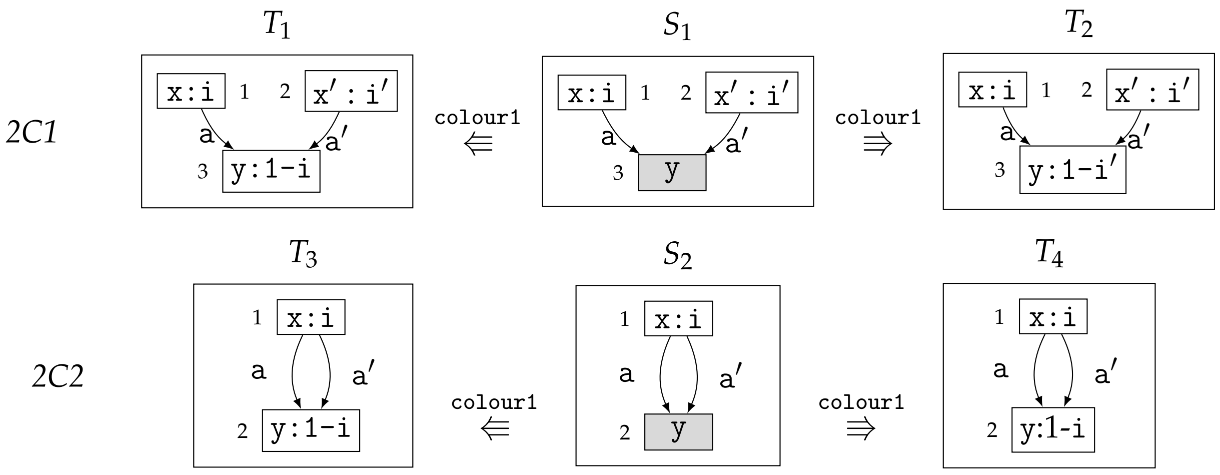


Figure 7.13: 2-colouring critical pairs involving `colour1` only.

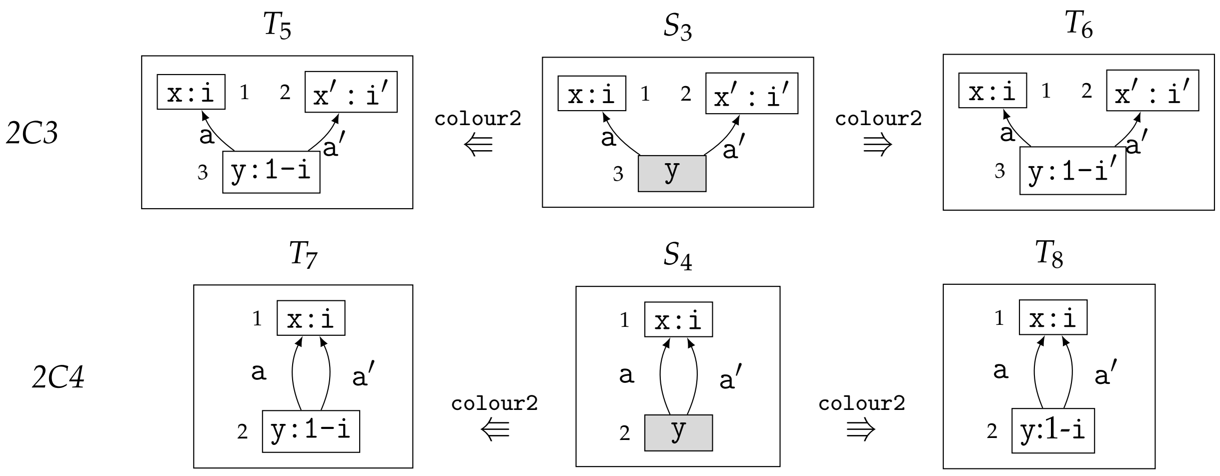


Figure 7.14: 2-colouring critical pairs involving `colour2` only.

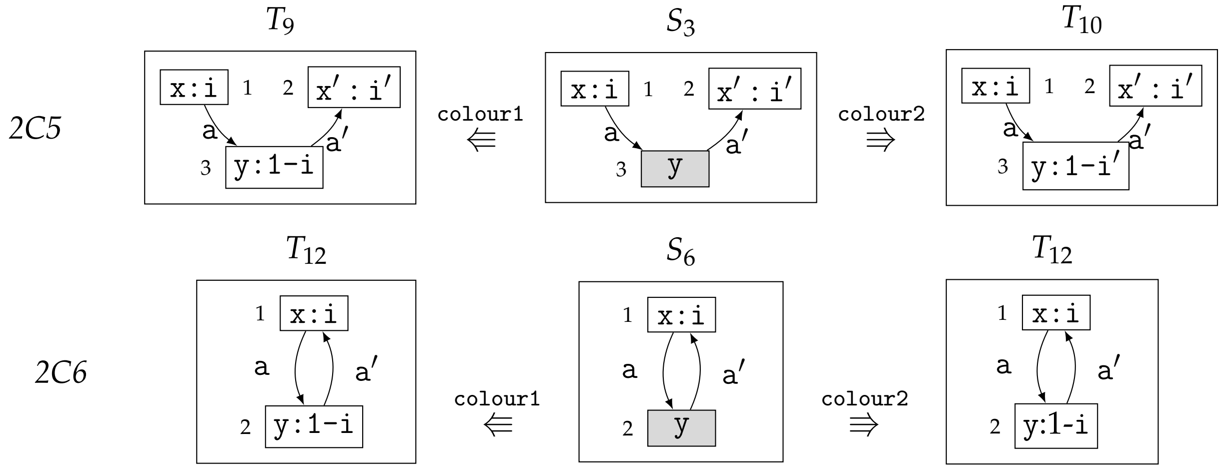


Figure 7.15: 2-colouring critical pairs involving both `colour1` and `colour2`.

Symbolic Critical Pairs. The critical pairs of `colour1` and `colour2` are given in Figure 7.13 (`colour1` and itself), Figure 7.14 (`colour2` and itself) an finally Figure 7.15 (involving both rules). The `mark` rule has no critical pairs as any overlap of the rule and itself would produce isomorphic results, i.e. the loop `mark!` is confluent.

The small number of critical pairs is due to the structure of the `colour1/2` rules. The rules only modify a single node - the marked (grey) node 2, by attaching a distance to its label and unmarking it. The semantics of matching GP 2 marks ensures marked nodes cannot match unmarked ones, making the rule set terminate, and in particular making the result graphs of the critical pairs (persistent) normal forms. Furthermore, in order to construct a valid conflict, the marked node of one rule instance can only overlap with a marked node of another rule instance, further reducing the number of critical overlaps.

Joinability Analysis. Exploring the derivation trees of the critical pairs is not necessary since all result graphs involved are (persistent) normal forms w.r.t. `colour`: there are no grey nodes for either rule to be applied any further. Therefore, joinability of any of the critical pairs relies entirely on whether the result graphs are E-isomorphic.

The critical pairs 2C2, 2C4 and 2C6 are strongly joinable because they produce isomorphic graphs with syntactically equal labels. This is no surprise from the view of the 2-colouring problem as parallel edges make no difference to the computation of such a colouring.

However, when we look at e.g. critical pair 2C1, it is not joinable because the induced label equality $\mathtt{i} = \mathtt{i}'$ is satisfiable but invalid. This also holds for the critical pairs 2C3 and 2C5.

Confluence of `colour!` and 2-colouring. We give a counterexample to confluence of the {`colour1`,`colour2`}`!` sub-program by instantiating the critical pair 2C1

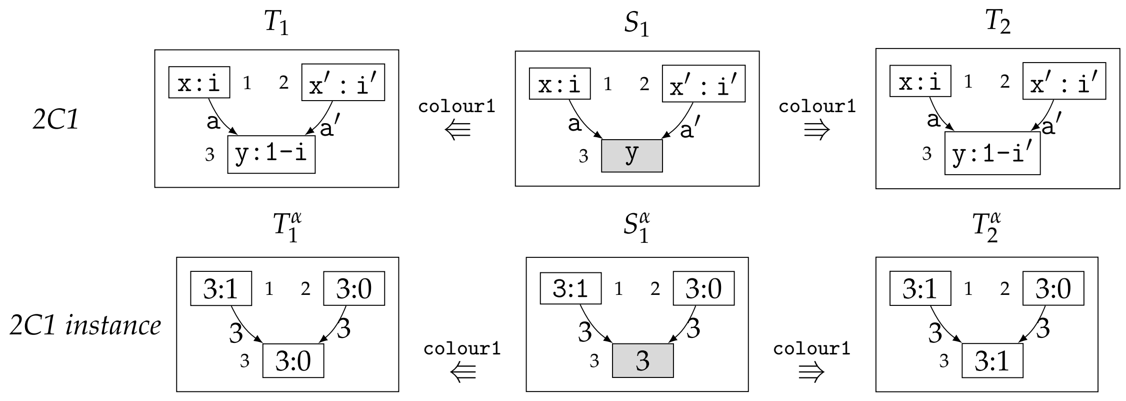


Figure 7.16: 2-colouring confluence counterexample - an instance of 2C1.

(Figure 7.16) with different colours for nodes 1 and 2, i.e. such that i $\neq$ i′ to obtain non-isomorphic normal forms. The remaining variables are instantiated to the integer 3 for simplicity. Therefore {colour1,colour2}! is not confluent.

An interesting issue is raised when one looks at the sub-program after colour: the code (if illegal then fail) checks for an illegal 2-colouring and fails if one exists. This essentially means that the above normal forms would both produce the fail result, but it is still an open question how fail factors into confluence analysis. The difficulty lies with the fact that fail is a *configuration* rather than a special kind of graph, so the notion of joinability would have to account for this special situation.

Suppose we are able to analyse the sub-program (if illegal then fail) for confluence: the rule illegal (a *predicate* rule) is either applicable or not[1], and the special fail command fails on any graph as input, i.e. the sub-program is confluent. Regardless, the 2-colouring program non-deterministically picks an initial node to visit and assign colour using the init rule. The problem now is that init is applied exactly once, and even though it has no critical pairs (any possible overlap violates the same-rule-different-match requirement), it introduces non-confluent behaviour: a graph 2-colouring (if it exists) is unique *up to swapping of colours*. In other words, our analysis cannot capture this more flexible notion of confluence even if it can deal with explicit failure and predicate rules in if-then-else.

Case Study Summary. Analysing this program, we have learned that:

- the sub-program computing a graph 2-colouring is not confluent by instantiating a non-joinable critical pair;
- GP 2 syntactic sugar constructs (e.g. bidirectional edges) inflate the number of rules to consider for confluence analysis, which in turn increases the number of critical pairs;

[1]The compiler exhaustively searches for a match, and does not backtrack if it finds several.

- considering failure explicitly is complicated as it is a program configuration rather than a graph;
- confluence *up to swapping of colours* of the 2-colouring program cannot be captured by the analysis, even if the above point is dealt with.

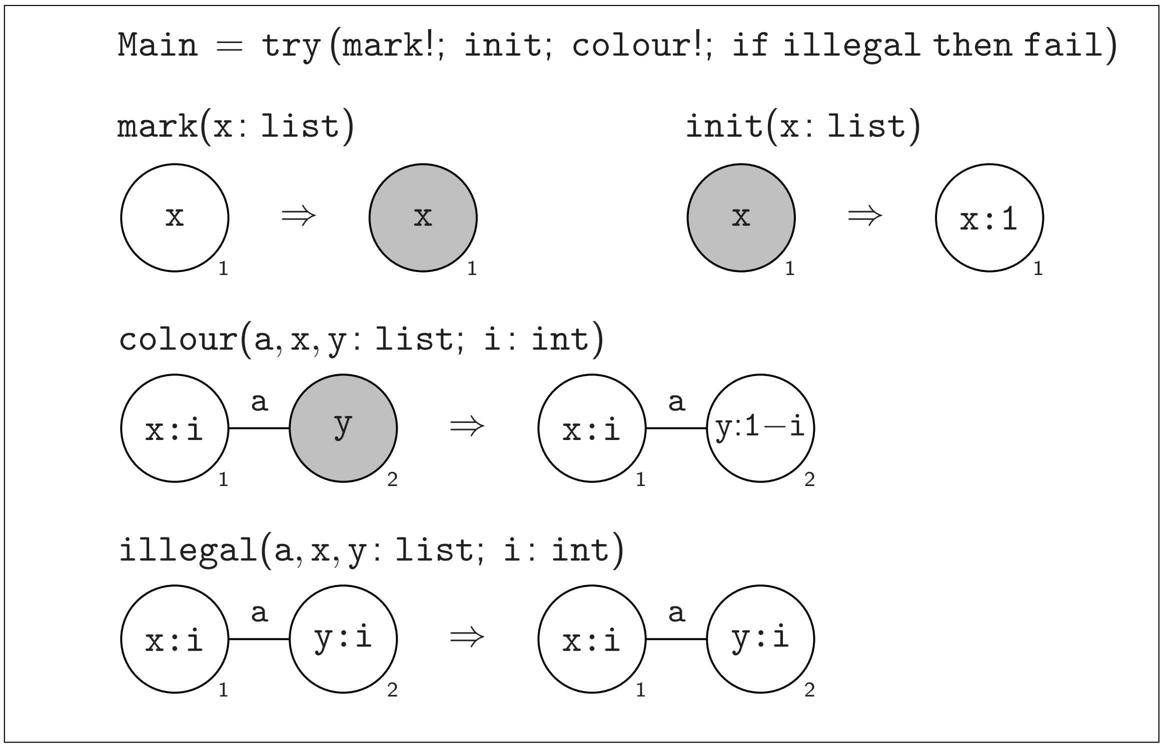

Main = try (mark!; init; colour!; if illegal then fail)
mark(x: list)
init(x: list)
colour(a, x, y: list; i: int)
illegal(a, x, y: list; i: int)
x
x:1
x:i
y
y:1−i
y:i
a
⇒
1
2

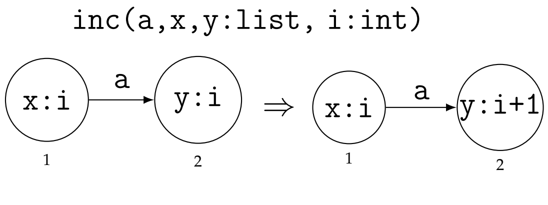
Figure 7.17: 2-colouring program, repeated from Subsection 2.2.3.4.

7.4 Vertex Colouring

The vertex colouring program computes *some* valid vertex colouring for a given graph [Plu16, Section 4], and was discussed in Subsection 2.2.3.4. Its program text is

$$\texttt{Main} = \texttt{mark!};\ \texttt{init!};\ \texttt{inc!}$$

where the rules are as the 2-colouring program (repeated in Figure 7.17) together with the `inc` rule:



Briefly, the program consists of two phases: first, assign an initial colour (the integer 1) to each node, and then increment colours of nodes with current colour equal to an adjacent node's colour. The first part of the computation is the sequential composition of (`mark!; init!`) and has a quadratic run time complexity (in the number of nodes). The second part is the as-long-as-possible application of the `inc` rule.

This second pass is highly non-deterministic and is guaranteed to terminate with quadratic run time complexity (for proof see [Plu16, Proposition 4]). What is special about this program is that there exists a symbolic critical pair which is non-joinable at the symbolic level, i.e. all of its instances are confluence counter examples. Therefore, a confluence algorithm can report 'non-confluence' safely in this specific case.

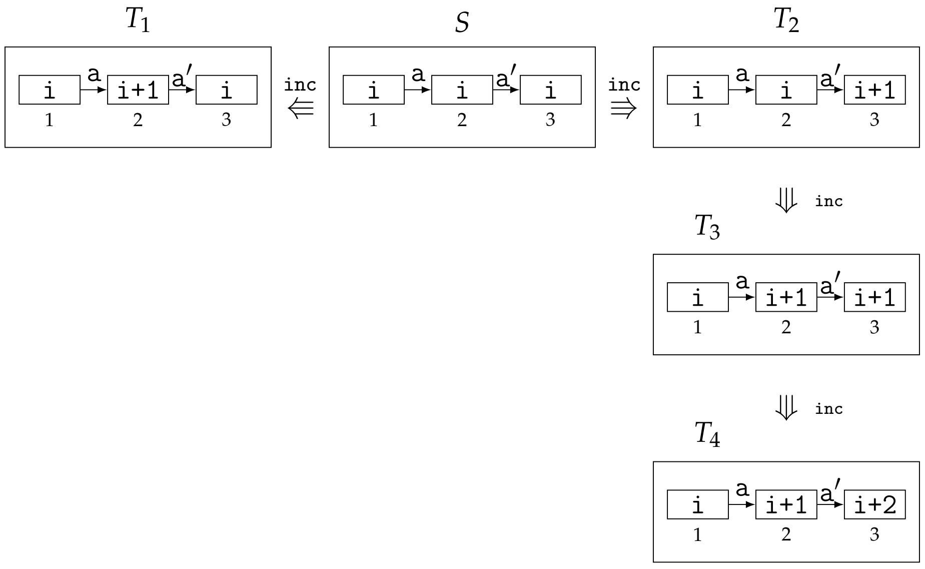


Figure 7.18: Critical pair of inc with itself.

Non-joinability. The mentioned critical pair is given in Figure 7.18. It is a sequence of three nodes where the middle node gets relabelled causing a conflict. The right result graph (T_2) is further modified by the sequence of symbolic derivations $T_2 \Rrightarrow T_3 \Rrightarrow T_4$. Now we will prove two important facts: that the graphs T_1 and T_4 represent normal forms, and that they are non-isomorphic.

Proposition 5. *All host graph instances of $T_1 \Lleftarrow S \Rrightarrow T_2 \Rrightarrow T_3 \Rrightarrow T_4$ are normal forms. Furthermore, for all instances T_1^α and T_4^α we have that $T_1^\alpha \ncong T_4^\alpha$.*

Proof. Suppose α is any valid GP 2 assignment for the integer variable i that is common to all the graphs of the critical pair. Then the rule inc cannot be applied to:

- T_1 - the symbolic application requires solving the unsatisfiable matching problem $\{i' = i,\ i' = i+1\}$ where we have renamed the matched variable of inc to avoid confusion; unsatisfiability can be checked by Z3
- T_4 - same as above, with systems of equations $\{i' = i,\ i' = i+1\}$ and $\{i' = i+1,\ i' = i+2\}$, again both of which are unsatisfiable and can be checked by Z3

Therefore, for any assignment, the graphs T_1^α and T_4^α are normal forms.

The given graphs are non-isomorphic for all instantiations because the (unique) bijective mapping $T_1 \to T_4$ induces a system of label equations

$$\{\mathtt{i} = \mathtt{i},\ \mathtt{i} = \mathtt{i} + 2,\ \mathtt{i} + 1 = \mathtt{i} + 1\}$$

that has no unifiers. Therefore, for any assignment $T_1^\alpha \ncong T_4^\alpha$. □

7.5 Chapter Summary

In conclusion, we have learned the following:

- full confluence analysis can be performed on existing GP programs that solve interesting graph problems;
- even when a given program is not confluent, a counter example can be constructed using the results of the analysis;
- several GP-specific features can be integrated into the analysis, e.g. list expressions, marks, (a fragment of) application conditions;
- an SMT solver can be used to solve several types of problems arising during the analysis such as label equalities, construction of confluence counter examples, and implications between application conditions;
- joinability analysis w.r.t. application conditions is difficult in that a specific critical pair can be strongly joinable given that a certain condition is true;
- some GP features appear harder to integrate into confluence analysis: we haven't considered programs that have so-called *roots* (a boolean flag attached to nodes, allows for constant-time matching), or programs that result in explicit *failure*.

Chapter 8

Conclusions and Future Work

In this chapter we draw the thesis to a close, first with some conclusions, and then with some suggestions for interesting future work.

8.1 Conclusions

In Section 1.2 we proposed a research hypothesis:

The language GP 2 can be effectively equipped with a confluence analysis system, facilitating proofs about confluence of many interesting graph programs

We also identifid criteria for the acceptability of such a system:

- *sound*: every confluence result must be valid with respect to the semantics of the language;
- *realistic*: in that it does not require impractical assumptions or restrictions on programs;
- *automatable*: all components of the system must be fully specified or implemented

We believe that the contributions of this thesis satisfy these criteria, and hence support the hypothesis. We began by introducing the notions of independence and conflict for rule schemata, central to the study of confluence. We lifted the notions of independence and conflict to rule schemata (Section 3.2), and proved the Local Church-Rosser Theorem (Theorem 3.1) which establishes that independent derivations are commutative and thus lead to the same result regardless of application order. This line of work is important not only because it is a paving stone for defining critical pairs, but also because without proving the commutativity of independent derivations, the confluence analysis based on critical pairs would be unsound.

Next, we developed critical pair analysis for sets of rule schemata by introducing symbolic critical pairs (Section 4.2), which are pairs of derivations at the level of schemata, i.e. labelled with expressions, that are minimal and in conflict. We defined our critical pairs at the level of graphs labelled with expressions to avoid an infinite number of such pairs. Then we gave an *algorithm* for their construction (algorithm 2 of Section 4.3), and showed the set of critical pairs is complete (Theorem 4.2) and finite (Corollary 1) under suitable but *realistic* restrictions. The construction algorithm is based on computing graph overlaps, and unifying overlapped labels using our rule-based unification algorithm for solving systems of label equations (Chapter 5). We showed that the unification algorithm terminates, is *sound* and also complete meaning that every unifier of the input system of equations is an instance of some unifier in the computed set of solutions (Theorem 5.4).

We continued by introducing our notion of strong joinability of critical pairs, based on rewriting of graphs labelled with expressions (Section 6.2). This is necessary as graphs in critical pairs are, in general, labelled with expressions rather than concrete values. Then, we established the Local Confluence Theorem for GP 2 schema rewriting (Theorem 6.1), which in effect allows for the specification of a *sound* confluence analysis *algorithm* based on critical pairs (algorithm 4 of Section 6.5). We discussed the practical aspects of the algorithm by looking at two refinements that reduce its search space and improve its accuracy.

Of course, we are not claiming to have equipped the language with the *most* effective confluence analysis system based on critical pair analysis. Indeed, much work remains to be done in addressing the deficiencies, amongst which the lack of means to detect non-confluence at the symbolic/expression level, and a formal treatment of (label) conditions present in conditional rule schemata, stand out as two of the most pressing — we go into more detail in the following section. But we are hopeful this thesis presents a step in the right direction, and has widened the class of graph programs whose (non-)confluent behaviour can be reasoned about.

8.2 Future Work

Here, we discuss some potential future work, including incorporating other GP 2 features into the analysis like application conditions, roots, the ‘any’ mark, formalizing our work in a confluence assistant, and more case studies. Topics we do not otherwise cover but are also important are the handling of control structures in a confluence calculus and possible external uses of the information obtained during confluence analysis.

Conditions. Currently, our analysis does not deal with GP 2 conditions as presented in Chapter 2. As presented, the theory only considers unconditional rule schemata. One of the issues is that considering only overlaps of left-hand graphs of schemata is insufficient for constructing a complete set of critical pairs. This is due to the more involved nature of conflicts: one rule can create items which are forbidden by another, and thus derivations which are not in delete-use conflicts can be in conflict. The way to deal with this problem is to consider extra overlaps that are constructed as follows: overlap the graph forbidden by one rule with the right-hand side of the other rule, and apply the inverse rule to reach the common graph of the critical pair. However, this works only for application conditions when expressed as morphisms rather than being textual as in the setting of GP 2.

The other issue with conditions is that extra results are necessary about how conditions behave in the various proofs of the classical results. Typically, one has to define how conditions are *shifted* over morphisms and over rules, and this is non-trivial to define and use in proofs.

Hopefully, it is not too difficult to solve the above problems. We have already presented a way to deal with textual conditions in schemata when they only restrict the values of label variables, as shown in the Shortest Distances case study (Chapter 7). Specifically, we record application conditions as assumptions that are to be resolved later by the SMT solver Z3, checked both for satisfiability and for implying label equivalences between symbolic graphs. Although it does not seem difficult to define shifting of conditions over morphisms, i.e. by applying the computed unifier to the condition text, this is a technical aspect that is crucial to the process of handling conditions. Last but not least, it would be useful to specify a fragment of conditions for which the relevant problems of isomorphism checking and implication checking are decidable, such as no arithmetic multiplication and no edge predicates.

A possible parallel to an existing implementation is the work of [Dec17] where handling of label conditions is delegated by Z3. What is more, the formal treatment of label conditions as part of the graphs is also covered by [Dec17]. (Note that application conditions are not part of the implementation nor the critical pair analysis framework.)

Future topics of work, among others, are the more advanced E-conditions and MSO-conditions of [PP14]. More specifically, the filtering of critical pairs based on a consistency check, although a sound idea, has to be done manually in Section 7.2 and it is not clear whether a condition such as 'there is no negative-weight cycle' can be even expressed, let alone automatically checked, in those existing more sophisticated frameworks.

Rooted Graph Transformation. *Rooted* graph transformation [BP12] is a formalism that employs specific nodes, called *roots*, that can be matched in constant time, and thus benefit from the high-level specification power of graph transformation while avoiding costs of matching. It is part of the compiler implementation of GP 2 [Bak16], and some compiled rooted graph programs are comparable in performance to purpose-built C implementations of graph algorithms [BP16]. Rooted graph programming is more difficult to get right, and would benefit from static confluence analysis as a form of correctness check.

There are some interesting issues with the framework. First, this kind of rewriting does not fall within the theory of $\mathcal{M},\mathcal{N}$-adhesiveness unlike rules with relabelling on which the rest of the language is based. The problem is that rules can create roots, which are treated as a separate boolean flag on nodes rather than part of its label. This becomes a problem as the second square in a double-pushout diagram is not natural. This complication of the underlying principles would imply heavy changes to accompanying confluence analysis results. A possible way to offset this challenge is the use of projective graph transformation [Dec17] that allows for the right morphism in rules to be from a special class of morphisms.

Syntactic sugar. Several of GP 2's features are useful when writing programs but can be considered syntactic sugar - the 'any' mark in rules allows matching any marked node; the bidirectional flag on edges matches host graph edges in either direction. These can be useful as they reduce the number of rules an end-user needs to write. However, our analysis does not handle them: we had to normalise the bidirectional edge in the 2-colouring program (Chapter 7), and did not look at a variation of the Shortest distances program that uses the *any* mark.

Dealing with these features as part of the analysis is not critical, as we showed they can be normalised away at the cost of having a larger number of rules to consider. This cost increases as programs become more complicated due to larger rules utilising these features. The difficulty with incorporating them into the analysis is specifying how the labels of overlaps are computed. For example, what happens when a node marked as 'any' is overlapped with a node marked as 'blue' or with a node that is unmarked? The solution is to substitute the any mark in the first instance, and forbid the overlap in the second. The situation with bidirectional edges is similar.

Implementation. Automatically checking whether a (sub-)program is confluent can be implemented with a lot of hard work. We gave a number of algorithms that are specified but are not implemented in code: (1) a unification algorithm for GP 2 lists that is rule-based and non-deterministic (algorithm 3 in Section 5.3); (2) a critical pair construction algorithm that relies on computing graph overlaps and unifying the induced systems of equations (algorithm 2 in Section 4.3); (3) a confluence analysis algorithm relying on a symbolic matcher for labels (algorithm 4 in Section 6.5). A number of tools already rely on implementing graph overlaps, e.g. [RET11], as that process is well-understood in mathematics (it is essentially set partitioning) and is inherent in computing critical pairs. Implementing confluence analysis is very useful as constructing critical pairs by hand is tedious and error-prone, often resulting in incorrect analysis due to a missed critical pair or incorrect strong joinability observations. An interactive confluence checker can use user input to guide joinability analysis, e.g. by specifying which rule to apply and where as given by the user.

Implementing the unification algorithm is non-trivial. However, the unification rules resemble Haskell's pattern matching very closely, and hence the relationship could be exploited to get a quick implementation. Furthermore, this line of work is supported by the fact our unification algorithm is modular: the rest of the framework only depends on what its inputs and outputs are, but not at exactly how it performs the computation, as long as the computation has the necessary properties of Termination, Soundness and Completeness (presented in Section 5.4 and Section 5.5).

A symbolic matcher is also non-trivial to implement, but the current GP 2 implementation could be used to this end. For a given host graph and schema, it computes a valid assignment during computing a valid match. This can be altered to try build substitutions instead, given that the input graph is labelled with expressions. This is interesting as it implies a connection between the host graph matcher and the symbolic matcher.

Appendix A

Basic Properties of $\mathcal{G}_\perp$

In this appendix we recall several properties of the category of partially labelled graphs $\mathcal{G}_\perp$. These properties, together with proofs and explicit constructions, can be found in [HP12] and [HP02].

Let $\mathcal{M}$ be the class of injective label preserving morphisms and $\mathcal{N}$ be the class of injective label and undefinedness preserving morphisms, the horizontal and vertical morphisms in double pushouts respectively. (Note that $\mathcal{N} \subseteq \mathcal{M}$.) The paper [HP12] shows that the category of partially labelled graphs $\mathcal{G}_\perp$ is a so-called $\mathcal{M},\mathcal{N}$-adhesive category and the following properties.

Fact A.1 ($\mathcal{G}_\perp$ is an $\mathcal{M},\mathcal{N}$-adhesive category)**.** The category $\mathcal{G}_\perp$ has the properties of $\mathcal{M},\mathcal{N}$-adhesive categories:

1. $\mathcal{M}$ and $\mathcal{N}$ contain all isomorphisms and are closed under composition and decomposition. Moreover, $\mathcal{N}$ is closed under $\mathcal{M}$-decomposition, that is, $A \to B \to C \in \mathcal{N}$ and $B \to C \in \mathcal{M}$ implies $A \to B \in \mathcal{N}$.

2. $\mathcal{G}_\perp$ has $\mathcal{M},\mathcal{N}$-pushouts and pullbacks along $\mathcal{M}$-morphisms.

3. $\mathcal{M}$ and $\mathcal{N}$ are stable under $\mathcal{M},\mathcal{N}$-pushouts and $\mathcal{M}$-pullbacks.

Remark 10. A morphism class X is closed under composition and decomposition if $A \to B, B \to C \in$ X implies $A \to B \to C \in$ X, and $A \to B \to C, B \to C \in X$ implies $A \to B \in$ X.

A pushout along an $\mathcal{M},\mathcal{N}$-morphism span, or $\mathcal{M},\mathcal{N}$-pushout, is a pushout where one of the given morphisms is in $\mathcal{M}$ and the other is in $\mathcal{N}$. A pullback along an $\mathcal{M}$-morphism, or $\mathcal{M}$-pullback, is a pullback where at least one of the given morphisms is in $\mathcal{M}$.

A class of morphisms X is stable under $\mathcal{M},\mathcal{N}$-pushouts if, given the $\mathcal{M},\mathcal{N}$-pushout (1) in Figure A.1a, $m \in \mathcal{X}$ implies $n \in \mathcal{X}$. Similarly, a class of morphisms X is stable under $\mathcal{M}$-pullbacks if, given the $\mathcal{M}$-pullback (1) in Figure A.1a, $n \in$ X implies $m \in$ X. □

Fact A.2 ($\mathcal{G}_\perp$ has HLR-properties)**.** The category $\mathcal{G}_\perp$ has the following HLR-properties:

1. $\mathcal{M},\mathcal{N}$-pushouts are pullbacks.

$$\begin{tikzcd}
A \arrow[r, "m"] \arrow[d] \arrow[dr, phantom, "(1)"] & C \arrow[d] \\
B \arrow[r, "n"'] & D
\end{tikzcd}$$

(a) Commutative square.

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & C \arrow[d] \\
B \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & D \arrow[d] \\
E \arrow[r] & F
\end{tikzcd}$$

(b) Commutative diagram.

Figure A.1: Commutative diagrams in $\mathcal{G}_\perp$.

2. $\mathcal{M},\mathcal{N}$-pushout-pullback decomposition: If (1+2) in the commutative diagram of Figure A.1b is a pushout, (2) is a pullback, $A \to C \in \mathcal{M}$ and $A \to B, D \to F \in \mathcal{N}$, then the squares (1) and (2) are pushouts and pullbacks.

3. Uniqueness of $\mathcal{M},\mathcal{N}$-pushout complements: Given morphisms $A \to B \in \mathcal{M}$ and $B \to D \in \mathcal{N}$ (Figure A.1a), there is, up to isomorphism, at most one object C with morphisms such that the square (1) is a pushout.

Fact A.3 ($\mathcal{G}_\perp$ has HLR$^+$-properties)**.** Let $\mathcal{E}$ be the class of surjective, undefinedness preserving morphisms and $\mathcal{E}'$ be the class of pairs of jointly surjective morphisms. The category $\mathcal{G}_\perp$ has the so-called HLR$^+$-properties:

1. $\mathcal{G}_\perp$ has binary coproducts: Given two graphs A and B, the binary coproduct is the disjoint union $A + B$ together with morphisms $(i_A : A \to A + B, i_B : B \to A + B)$ such that the following universal property holds: for all objects X with morphisms $f : A \to X$ and $g : B \to X$, there is a morphism $[f,g] : A + B \to X$ such that $[f,g] \circ i_A = f$ and $[f,g] \circ i_B = g$.

2. $\mathcal{G}_\perp$ has $\mathcal{E}$-$\mathcal{N}$ pair factorization: for each morphism $f : A \to C$, there exist an object K and morphisms $A \to K \in \mathcal{E}$ and $K \to C \in \mathcal{N}$ such that $A \to C = A \to K \to C$

3. $\mathcal{G}_\perp$ has $\mathcal{E}'$-$\mathcal{N}$ pair factorization: for each pair of morphism $(m_1 : A \to C, m_2 : B \to C)$, there exist an object K and unique morphisms $A \to K, B \to K, K \to C$ such that $A \to C = A \to K \to C$, $B \to C = B \to K \to C$ with $(A \to K, B \to K) \in \mathcal{E}'$ and $K \to C \in \mathcal{N}$

Remark 11. Binary coproducts can be seen as a generalization of the disjoin union of sets and graphs to a categorical framework. Property 3 is a direct corollary of properties 1,2 and Remark 5.26 of [EEPT06]. □

Appendix B

The SELECT Algorithm

Input: Unification problem $\langle s =^? t \rangle$ and a unifier δ
Output: A sequence of unification rules, associated with a branch of the unification tree

```
    $\bar{s}, \bar{t} \leftarrow$ Preprocess $s, t$ according to $\delta$
    initialize the position pointers: $m = 1, n = 1$
    initialize the sequence of rules: $B = ''''$
    if $m > length(\bar{s})$ and $n > length(\bar{t})$ then stop
    Case analysis on $L(\bar{s}, m)$ and $L(\bar{t}, n)$:
Case 1. $L(\bar{s}, m) < L(\bar{t}, n)$
    Since $L(\bar{t}, n) > 0$, $\overline{t_n}$ must be a list variable
    Do subcase analysis on type of $\overline{s_m}$ and look ahead information:
Case 1.1. $Type(\overline{s_m}) = ListVar$    /* $x : L = y : M$ */
    It cannot be the case that the $AtEnd(\overline{s_m})$ returns $false$ (i.e. $L \neq empty$) as $\overline{t_n}$ is not of type $EmptyListVar$ ($L(\overline{t_n}) > 0$)
        select Decomp2'    /* $x : L = y : M$ with $L \neq empty$ */
        $m = m + 1 + L(\overline{s_m})$, $n = n + 1 + L(\overline{s_m})$    /* move head to end of shorter substring */
Case 1.2. $Type(\overline{s_m}) = AtomVar$
    problem is of the form $a : L = y : M$
    select Orient3, swap input tapes and pointers
Case 1.3. $Type(\overline{s_m}) = AtomExpr$
    problem is of the form $a : L = y : M$
    select Orient1, swap input tapes and pointers
Case 1.4. $Type(\overline{s_m}) = EmptyListVar$
    problem is of the form $x_e : L = y : M$
    It cannot be the case that the $AtEnd(\overline{s_m})$ returns $false$ (i.e. $L \neq empty$) as $\overline{t_n}$ is not of type $EmptyListVar$ ($L(\overline{t_n}) > 0$)
        select Subst2, $m$++, no change for $n$
Case 1.5. $Type(\overline{s_m}) = empty$
    This cannot happen, as $\overline{t_n}$ is not of type $EmptyListVar$ ($L(\overline{t_n}) > 0$) and $\delta$ is a unifier
```

Case 2. $L(\bar{s}, m) > L(\bar{t}, n)$

Since $L(\bar{s}, m) > 0$, $\overline{s_m}$ must be a list variable

```
  if $AtEnd(\overline{s_m}) = false$
    then (Case 2.1)
      select Subst1    /* x = y : $M$ */
      move head to end of both strings: $m = m + 1 + L(\overline{s_m})$, $n = length(\bar{t}) + 1$
    else    /* $L \neq empty$ */
```

Case 2.2. $Type(\overline{t_n}) = ListVar$

x : L = y : M with $L \neq M$ and x should start with y

```
    select Decomp2
    move head to end of shorter substring: $m = m + 1 + L(\overline{t_n})$, $n = n + 1 + L(\overline{t_n})$
```

Case 2.3. $Type(\overline{t_n}) = AtomVar\ or\ AtomExpr$

x : L = a : M with $L \neq M$ and x should start with a

```
    select Subst3
    $m$++, $n$++
```

Case 2.4. $Type(\overline{t_n}) = EmptyListVar$

x : $L = y_e$: M with $L \neq M$ and x should start with y

```
    select Decomp2
    $m$++, $n$++
```

Case 2.5. $Type(\overline{t_n}) = empty$

This cannot happen, same reason as Case 1.5

Case 3. $L(\bar{s}, m) = L(\bar{t}, n)$

Case 3.1. $L(\bar{s}, m) = L(\bar{t}, n) > 0$

Both $\bar{s}_m$ and $\bar{t}_n$ must be list variables

```
if $AtEnd(\overline{s_m}) = false$ then
  | problem has the shape x = y : $M$
  | select Subst1
  | move head to end of both strings: $m = m + 1 + L(\overline{s_m})$, $n = length(\bar{t}) + 1$
else
  | we know $L \neq empty$, and the problem has the shape x : $L$ = y : $M$
  | select Decomp1
  | move head to next pair of unrelated symbols: $m = m + 1 + L(\overline{s_m})$,
  |   $n = n + 1 + L(\overline{s_m})$
end
```

Case 3.2. $L(\bar{s}, m) = L(\bar{t}, n) = 0$

This is the largest subcase because $\overline{s_m}$ and $\overline{t_n}$ can be of any of the possible types. Do subcase analysis on $Type(\bar{s}_m)$ and $Type(\bar{t}_n)$:

3.2.1. $(ListVar, ListVar)$ – same as 3.1

3.2.2. $(ListVar, AtomVar)$ – same as 3.1

3.2.3. $(ListVar, AtomExpr)$ – same as 3.1

3.2.4. $(ListVar, EmptyListVar)$ – same as 2.4

3.2.5. $(ListVar, empty)$ – cannot happen as δ is a unifier

3.2.6. $(AtomVar, ListVar)$ – Orient3 (like 1.2)

3.2.7. $(AtomVar, AtomVar)$ or $(AtomVar, AtomExpr)$

```
if $AtEnd(\overline{s_m}) = false$ then
    if $AtEnd(\overline{t_n}) = false$ then
        problem has the shape $\mathtt{a} = \mathtt{b}$
        select Subst1
        $m$++, $n$++
    else
        problem has the shape $\mathtt{a} = \mathtt{b} : \mathtt{y}$ with $\delta(\mathtt{y}) = \mathtt{empty}$
        select Decomp4
        move head to next pair of unrelated symbols: $m$++, $n$++
    end
else
    we know $L \neq empty$, and the problem has the shape $\mathtt{a} : L = \mathtt{b} : M$
    select Decomp1'
    move head to next pair of unrelated symbols: $m$++, $n$++
end
```

3.2.8. $(AtomVar, EmptyListVar)$ – same as 1.2

3.2.9. $(AtomVar, empty)$ – cannot happen due to failure lemmata

3.2.10. $(AtomExpr, ListVar)$ – same as 1.3

3.2.11. $(AtomExpr, AtomVar)$ – same as 1.3

3.2.12. $(AtomExpr, AtomExpr)$ –
/* must be the same expression due to not considering the subunification algorithm for String-Char */
select Decomp3, m++, n++

3.2.13. $(AtomExpr, EmptyListVar)$ – same as 1.3

3.2.14. $(AtomExpr, empty)$ – cannot happen due to failure lemmata

3.2.15. $(EmptyListVar, ListVar)$ or $(EmptyListVar, AtomVar)$ or $(EmptyListVar, AtomExpr)$
It must be the case that there is something following $\overline{s_m}$ for δ to be a unifier. Proceed like case 1.4
select Subst2, m++

3.2.16. $(EmptyListVar, EmptyListVar)$ – same as 3.1

3.2.17. $(EmptyListVar, empty)$ – select Subst1, m++, $n = length(\overline{t_n}) + 1$

3.2.18. $(empty, ListVar)$ – cannot happen as δ is a unifier

3.2.19. $(empty, AtomVar)$ or $(empty, AtomExpr)$ – cannot happen due to failure lemmata

3.2.20. $(empty, EmptyListVar)$ – select Orient4, swap tapes and pointers

3.2.21. $(empty, empty)$ – select Remove

Appendix C

Proofs

This appendix contains some additional technical proofs that are either too lengthy or too technical to be contained in the main text.

C.1 Proof of the SELECT Lemma

Lemma 15 (SELECT Lemma). *There exists an algorithm SELECT($\langle s =^? t\rangle, \delta$) that takes a unification problem $\langle s =^? t\rangle$ and a unifier δ as input and produces a branch of its unification tree represented as a sequence of rule selections $B = (b_1, \ldots, b_k)$ such that:*

1. *UNIFY($\langle s =^? t\rangle$) has a branch specified by B.*
2. *For the sequence of rules $b \in B$:*
   - *(a) the symbols examined by UNIFY at node b are parent symbols of the symbols SELECT sees at head position (m, n) for node b*
   - *(b) if σ is the substitution corresponding to b, then there exists an instantiation λ such that $\sigma \circ \lambda \leq \delta$.*

Proof. By induction on the number of the rule selections $(b_1, b_2, \ldots)$ of B:

Base Case The initial rule selection b_1 of SELECT is based on the first symbols of $\bar{s}$ and $\bar{t}$ as this is the initial head position (m and n are initialized to 1). The rules of UNIFY also examine the head of the initial problem. Also, SELECT does not throw away list variables that should be removed due to Subst2, so condition 2.a is trivially satisfied. For 2.b, $\sigma_1 = id$ by definition of initial unification problem, and let $\lambda_1 = id$. Therefore $\sigma_1 \circ \lambda_1 = id \leq \delta$ by the properties of the identity substitution, as required.

Hypothesis Assume the statements of the theorem are true for selection b_k with a corresponding node in the unification tree $\langle P_k, \sigma_k\rangle$ and let (m, n) be the head position of SELECT before outputting selection b_{k+1}.

2.a the symbols at the head of the problem P_k (x and y) are parent symbols of $\overline{s}_m, \overline{t}_n$

2.b exists substitution λ_k such that $\sigma_k \circ \lambda_k \leq \delta$

Inductive case There are several possible cases for the selection b_{k+1}. We have to show that from b_n,

1. UNIFY *can* apply the rule b_{k+1}, and
2. The symbols at the head of the next unification problem P_{k+1} (where $P_k \Rightarrow P_{k+1}$ using the rule b_{k+1}) are parent symbols of the new current symbols $\overline{s}_{m_{new}}$ and $\overline{t}_{n_{new}}$
3. exists instantiation λ_{k+1} such that $\sigma_{k+1} \circ \lambda_{k+1} \leq \delta$

For all Orient rules, we can notice that:

- they do not change the head of the unification problem nor they generate a supplementary substitution σ_{k+1}, so conditions 2 and 3 trivially hold
- they always swap left-hand side with right-hand side; so does SELECT for each Orient selection
- Therefore, we only need to prove condition 1 for each Orient selection.

We now have to consider all the cases for $P_k \Rightarrow P_{k+1}$ in SELECT using rule b_{k+1}.

Case 1. $L(\overline{s}, m) < L(\overline{t}, n)$

It must be the case that $\overline{t}_n$ is a list variable because $L(\overline{t}, n) > 0$ only for list variables.

Call x and y the parent symbols of $\overline{s_m}$ and $\overline{t_n}$ in s and t.

Case 1.1. $Type(\overline{s_m}) = ListVar$

By hypothesis, it follows that x and y are also list variables.

Suppose nothing follows x (i.e. $LookAhead(\overline{s}_m) = false$). Then the unification problem at node b_k must be of the form x = y : M. However, we have that $L(\overline{s}, m) < L(\overline{t}, n)$ which contradicts that δ is a unifier.

Therefore, there must be something following x. Then the current unification problem must be of the form x : L = y : M. Then rule Decomp2' is applicable (condition 1 satisfied). Because x has the shorter expansion by δ, $\overline{t}_{n+L(\overline{s}_m)+1}$ is a child variable of y and condition 2 holds.

For 3, let $\lambda'_k = (\overline{s_n} \ldots \overline{s_{m+L(\overline{s}_m)}} \mapsto T)$ where $T = \delta(\overline{s_m} \ldots \overline{s_{m+L(\overline{s}_m)}})$. For δ to be a unifier, it must contain term(s) $(\overline{t_n} \ldots \overline{t_{n+L(\overline{s}_m)}} \mapsto T)$:

$$\begin{array}{rcl}
\delta & \geq & \sigma_k \circ \lambda_k \cup (\overline{t_n} \ldots \overline{t_{n+L(\overline{s}_m)}} \mapsto T) \\
& = & \sigma_k \circ \lambda_k \cup ((\overline{t_n} \ldots \overline{t_{n+L(\overline{s}_m)}} \mapsto \overline{s_m} \ldots \overline{s_{m+L(\overline{s}_m)}}) \circ \lambda'_k) \text{ (same as saying y starts with x} \\
& = & \sigma_k \circ \lambda_k \cup ((\mathrm{y} \mapsto \mathrm{x} : \mathrm{y}') \circ \lambda'_k) \\
& = & \sigma_k \circ (\mathrm{y} \mapsto \mathrm{x} : \mathrm{y}') \circ \lambda_k \circ \lambda'_k \text{ (since no repeated list variables)} \\
& = & \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k)
\end{array}$$

Case 1.2. $Type(\overline{s_m}) = AtomVar$

It holds by hypothesis that x and y are atom and list variables so rule Orient3 is applicable.

Case 1.3. $Type(\overline{s_m}) = AtomExpr$

The argument is the same as above except that x is an atom expression and Orient1 becomes applicable.

Case 1.4. $Type(\overline{s_m}) = EmptyListVar$

Because $L(\bar{t}_n) > L(\bar{s}_m) = 0$, there must be something after $\overline{s_{m+L(\bar{s}_m)}}$ (i.e. $LookAhead(\bar{s}_m) = true$) for δ to be a unifier. Also, by hypothesis x must be a list variable since $\mathtt{x}\delta = \mathtt{empty}$. Therefore rule Subst2 is applicable (condition 1 holds). The resulting unification problem is $(\{L = M\}\{\mathtt{x} \mapsto \mathtt{empty}\};\ \sigma_k \circ \{\mathtt{x} \mapsto \mathtt{empty}\})$. SELECT moves the position m by 1 and keeps n, therefore condition 2 also holds.

For condition 3:

$$\begin{aligned}
\delta &\geq \sigma_k \circ \lambda_k \cup (\mathtt{x} \mapsto \mathtt{empty}) \\
&= \sigma_k \circ (\mathtt{x} \mapsto \mathtt{empty}) \circ \lambda_k \text{ (since no repeated list variables)} \\
&= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k)
\end{aligned}$$

Case 1.5. $Type(\overline{s_m}) = \mathtt{empty}$

As explained, this subcase cannot occur if δ is a unifier.

Case 2. $L(\bar{s}, m) > L(\bar{t}, n)$

As in the previous case, it follows that $\bar{s}_m$ is a list variable with $|\delta(x)| > 0$ for its parent symbol x. By hypothesis, x is also a list variable.

Case 2.1. Let $LookAhead(\bar{s}_m) = false$, i.e. nothing follows x

The unification problem must be of the form $\mathtt{x} = L$ so rule Subst1 is applicable (condition 1 satisfied). Since the resulting unification problem is $\langle \varnothing, \sigma_k \circ \{\mathtt{x} \mapsto L\}\rangle$ and SELECT moves the pointers to the end of the tapes, condition 2 is satisfied.

For 3, let $\lambda'_k = (L \mapsto T)$ where $T = \delta(L)$. For δ to be a unifier, it must contain term(s) $(\mathtt{x} \mapsto T)$

$$\begin{aligned}
\delta &\geq \sigma_k \circ \lambda_k \cup (x \mapsto T) \\
&= \sigma_k \circ \lambda_k \cup ((x \mapsto L) \circ \lambda'_k) \\
&= \sigma_k \circ (x \mapsto L) \circ \lambda_k \circ \lambda'_k \text{ (since no repeated list variables)} \\
&= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k)
\end{aligned}$$

For the rest of the cases, let $LookAhead(\bar{s}_m) = true$ and the problem is of the form $\mathtt{x} : L = M$ with $L \neq empty$.

Case 2.2. $Type(\overline{t_n}) = ListVar$

By hypothesis, y must be a list variable. The problem is of the form $\mathtt{x} : L = \mathtt{y} : M$. Then rule Decomp2 is applicable (condition 1 satisfied). Because y has the shorter expansion by δ, $\overline{s_{m+L(\bar{t}_n)+1}}$ is a child variable of x and condition 2 holds.

For 3, let $\lambda'_k = (\overline{s_n} \ldots \overline{s_{m+L(\bar{t}_n)}} \mapsto T)$ where $T = \delta(\overline{s_m} \ldots \overline{s_{m+L(\bar{t}_n)}})$, and the argument becomes identical to Case 1.1 (x starts with y, so $\sigma_{k+1} = \sigma_k \circ \{\mathtt{x} \mapsto \mathtt{y} : \mathtt{x}'\}$)

Case 2.3. $Type(\overline{t_n}) = AtomVar$ or $Type(\overline{t_n}) = AtomExpr$

By hypothesis, y is an atom variable or atom expression. Then the problem is of the form $\mathtt{x} : L = \mathtt{a} : M$ with $L \neq \mathtt{empty}$. So rule Subst3 is applicable (condition 1 satisfied). Positions are incremented by 1, so condition 2 holds as $\bar{s}_{m+1}$ and $\mathtt{x}'$ (in Orient3) must have the same parent symbol.

For 3, let $\lambda'_k = (\bar{t}_n \mapsto T)$ where $T = \delta(\bar{t}_n)$. It follows that δ must contain term $(\bar{s}_m \mapsto T)$ to be a unifier.

$$\begin{aligned}
\delta &\geq \sigma_k \circ \lambda_k \cup (\bar{s}_m \mapsto T) \\
&= \sigma_k \circ \lambda_k \cup ((\bar{s}_m \mapsto \bar{t}_n) \circ \lambda'_k) \\
&= \sigma_k \circ \lambda_k \cup ((\mathtt{x} \mapsto \mathtt{a} : \mathtt{x}') \circ \lambda'_k) \\
&= \sigma_k \circ (\mathtt{x} \mapsto \mathtt{a} : \mathtt{x}') \circ \lambda_k \circ \lambda'_k \text{ (since no repeated list variables)} \\
&= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k)
\end{aligned}$$

Case 2.4. $Type(\overline{t_n}) = EmptyListVar$

We have that $\bar{t}_n$ is of the form $\mathtt{y}_e$ where $\mathtt{y}$ is a list variable with $\delta(\mathtt{y}) = \mathtt{empty}$. The unification problem is then of the form $\mathtt{x} : \mathtt{L} = \mathtt{y} : \mathtt{M}$. Rule Decomp2 becomes applicable

Let $\lambda'_k = (\mathtt{y} \mapsto \mathtt{empty})$

$$\begin{aligned}
\delta &\geq \sigma_k \circ \lambda_k \cup (\mathtt{y} \mapsto \mathtt{empty}) \\
&= \sigma_k \circ \lambda_k \cup ((\mathtt{x} \mapsto \mathtt{y} : \mathtt{x}') \circ \lambda'_k) \\
&= \sigma_k \circ (\mathtt{x} \mapsto \mathtt{y} : \mathtt{x}') \circ \lambda_k \circ \lambda'_k \text{ (because no repeated list variables)} \\
&= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k)
\end{aligned}$$

Case 3. $L(\bar{s}, m) = L(\bar{t}, n)$

Case 3.1. $L(\bar{s}, m) = L(\bar{t}, n) > 0$

It must be the case that $\bar{s}_m$ and $\bar{t}_n$ are list variables because $L(\bar{s}_m) > 0$ and $L(\bar{t}, n) > 0$ only for list variables.

By hypothesis, it follows that $\mathtt{x}$ and $\mathtt{y}$ are also list variables.

If there is nothing after the $\mathtt{x}$ (i.e. $LookAhead(\bar{s}_m) = false$). Then Subst1 is applicable and the argument is the same as Case 2.1.

Otherwise, it must be that $L \neq \mathtt{empty}$ so rule Decomp1 is applicable. Here the problem is of the form $\mathtt{x} : \mathtt{L} = \mathtt{y} : \mathtt{M}$ and the generated substitution is $\sigma_{k+1} = \sigma_k \circ (\mathtt{x} \mapsto \mathtt{y})$.

For 2, the argument is similar to Case 2.2 except that now both $\bar{s}_{m+L(\overline{s_m})+1}$ and L, and $\bar{t}_{n+L(\overline{s_m})+1}$ and M must have the same pairs of parent symbols.

Let $\lambda'_k = (\mathtt{y} \mapsto \delta(y))$. It must be the case that δ also contains term $(\mathtt{x} \mapsto \delta(y))$ to be a unifier.

$$\begin{aligned}
\delta &\geq \sigma_k \circ \lambda_k \cup (\mathtt{x} \mapsto \delta(y)) \\
&= \sigma_k \circ \lambda_k \cup ((\mathtt{x} \mapsto \mathtt{y}) \circ \lambda'_k) \\
&= \sigma_k \circ (\mathtt{x} \mapsto \mathtt{y}) \circ \lambda_k \circ \lambda'_k \text{ (because no repeated list variables)} \\
&= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k)
\end{aligned}$$

Case 3.2. $L(\bar{s}, m) = L(\bar{t}, n) = 0$

This is the largest subcase because $\overline{s_m}$ and $\overline{t_n}$ can be of any of the possible types.

3.2.1-3 $(ListVar, ListVar)$ or $(ListVar, AtomVar)$ or $(ListVar, AtomExpr)$–
Same as **3.1** because the parent symbol of $\bar{t}_n$ can be matched by s in Decomp1.

3.2.4. $(ListVar, EmptyListVar)$ – same as Case **2.4**

3.2.5. $(ListVar, empty)$ – cannot happen as δ is a unifier

3.2.6. $(AtomVar, ListVar)$ – same as Case **1.2**

3.2.7. $(AtomVar, AtomVar)$ or $(AtomVar, AtomExpr)$

- $LookAhead(\bar{s}_m) = true$, i.e. $L \neq$ `empty` - then the problem is of the form `a` $: L =$ `b` $: M$ (`a` and `b` are also atom variables by hypothesis) and rule Decomp1' is applicable. UNIFY generates $\sigma_{k+1} = \sigma_k \circ \{\texttt{a} \mapsto \texttt{b}\}$.

  Let $\lambda'_k = (\bar{s}_m \mapsto \delta(\bar{s}_m)) = (\texttt{a} \mapsto \text{¡}(\texttt{a}))$. Then δ must also contain term $(\bar{t}_n \mapsto \delta(\bar{s}_m))$ to be a unifier:

  $$\begin{aligned} \delta &\geq \sigma_k \circ \lambda_k \cup (\bar{t}_n \mapsto \delta(\bar{s}_m)) \\ &= \sigma_k \circ \lambda_k \cup ((\bar{t}_n \mapsto \bar{s}_m) \circ \lambda'_k) \\ &= \sigma_k \circ \lambda_k \cup ((\texttt{a} \mapsto \texttt{b}) \circ \lambda'_k) \\ &= \sigma_k \circ (\texttt{a} \mapsto \texttt{b}) \circ \lambda_k \circ \lambda'_k \\ &= \sigma_{k+1} \circ \lambda_{k+1} \text{ as required (set } \lambda_{k+1} = \lambda_k \circ \lambda'_k) \end{aligned}$$

- $LookAhead(\bar{s}_m) = false$, i.e. the problem is `a` $=$ `b` $:$ `M`.

  - If $LookAhead(\bar{t}_n) = true$, then M can only be a single list variable for δ to be a unifier with $\delta(\texttt{y}) =$ `empty`. This makes rule Decomp4 applicable. For 3, the argument is the same as in the previous item (with $\sigma_{k+1} = \sigma_k \circ \{\texttt{a} \mapsto \texttt{b}\}$).
  - If $LookAhead(\bar{t}_n) = false$, then the problem is of the form `a` $=$ `b`, which makes Subst1 is applicable and the argument is the same as Case 2.1

3.2.9. $(AtomVar, EmptyListVar)$ – same as Case **1.2**
3.2.10. $(AtomVar, empty)$ – cannot happen due to failure lemmata
3.2.11. $(AtomExpr, ListVar)$ – same as Case **1.3**
3.2.12. $(AtomExpr, AtomVar)$ – same as Case **1.3**
3.2.13. $(AtomExpr, AtomExpr)$ – Both symbols must be the same atom expression due to not considering the subunification algorithm for String-Char. So rule Decomp3 is applicable. Conditions 2 and 3 hold trivially.
3.2.14. $(AtomExpr, EmptyListVar)$ – same as **1.3**
3.2.16-18. $(EmptyListVar, ListVar)$ or $(EmptyListVar, AtomVar)$ or $(EmptyListVar, AtomExpr)$

It must be the case that there is something following $\overline{s_m}$ for δ to be a unifier. Proceed proof like Case **1.4**

3.2.19. $(EmptyListVar, EmptyListVar)$ – same as **3.1**
3.2.20. $(EmptyListVar, empty)$

It follows that `x` (the parent of $\bar{s}_m$) is a list variable with $\delta(\texttt{x}) =$ `empty`. Also, since there are no repeated list variables, for δ to be a unifier there must be nothing following `x`. So the problem is of the form `x` $=$ `empty` and rule Subst1 is applicable.

For condition 3, proceed as Case **1.4**

3.2.21. $(empty, ListVar)$ – cannot happen because δ is a unifier
3.2.22 & 3.2.23 $(empty, AtomVar)$ or $(empty, AtomExpr)$ – cannot happen due to failure lemmata
3.2.24. $(empty, EmptyListVar)$

It follows that y (the parent of $\bar{t}_n$) is a list variable with $\delta(\texttt{y}) = \texttt{empty}$. So the problem has the form $\texttt{empty} = \texttt{x} : L$ and rule Orient4 is applicable.

3.2.25. $(empty, empty)$

It follows that rule Remove is applicable. Conditions 2 and 3 hold trivially.

□

C.2 Equivalence between Joinability Definitions

In this section, we explore the different definitions of strong joinability appearing in literature, more specifically the track formulation of [Plu05] and the commutative-squares formulation of [EEPT06], an earlier version of which appeared in [EHPP04]. We have used the commutative-squares definition in Section 6.3 and mainly in the proof of the Local Confluence Theorem for rule schemata (Theorem 6.1).

Definition C.1 (Track morphism, Persist graph [Plu05]). Given a direction derivation $G \Rightarrow H$ with span $G \leftarrow^{c} D \rightarrow^{c'} H$, the *track morphism* $track_{G \Rightarrow H} : G \rightarrow H$ is the graph morphism between a subgraph of G and H defined as

$$track_{G\Rightarrow H} = c'(c^{-1}(x) \textit{ unless } x \notin c(D)$$

The track morphism of a derivation $t : G_0 \Rightarrow^* G_n$ is defined inductively as $track_t = id_{G_0}$ if $n = 0$, and $track_t = track_{G_1 \Rightarrow^* G_n} \circ track_{G_0 \Rightarrow G_1}$ otherwise.

The graph *Persist* is the domain of $track_{G \Rightarrow^* H}$. □

A graph morphism between a subgraph of G and H is called a *partial graph morphism*.

Definition C.2 (Strong Joinability [Plu05]). Let $cp : T_1 \Leftarrow S \Rightarrow T_2$ be a critical pair and define $Persist_{cp}$ be the intersection of graphs $Persist_{S \Rightarrow T_1} \cap Persist_{S \Rightarrow T_2}$. Then the critical pair is *stronly joinable* if it is joinable, i.e. exist derivations $T_1 \Rightarrow^* X_1$ and $T_2 \Rightarrow^* X_2$ together with an isomorphism $f : X_1 \rightarrow X_2$ such that for each item $x \in Persist_{cp}$:

1. $track_{S \Rightarrow T_1 \Rightarrow^* X_1}(x)$ and $track_{S \Rightarrow T_1 \Rightarrow^* X_2}(x)$ are defined;

2. $f(track_{S \Rightarrow T_1 \Rightarrow^* X_1}(x)) = track_{S \Rightarrow T_1 \Rightarrow^* X_2}(x)$

The definition communicates the idea that each item that is preserved by both derivations of the critical pair must be preserved by the joining derivations, and its descendants in X_1 and X_2 have to be related by the isomorphism $f : X_1 \rightarrow X_2$. Note we have used the formulation of [Plu05] with the modification of using graph items rather than only nodes.

We now give a commutative version of the track formulation.

Definition C.3 (Commutative formulation of single-step *track*). Given a direction derivation $G \Rightarrow H$ with span $G \leftarrow^{c} D \rightarrow^{c'} H$, computing its track function amounts to constructing the diagram in Figure C.2 as follows:

$$\begin{tikzcd}
G & D \arrow[l, "c"'] \arrow[r, "c'"] \arrow[d, "i"'] & H \\
(1) & c(D) \arrow[ul] \arrow[ur] \arrow[u, dashed, bend right, "i^{-1}"'] & (2)
\end{tikzcd}$$

Figure C.1: Commutative-squares formulation of a track morphism for direct derivations.

$$\begin{tikzcd}
G_0 & D_0 \arrow[l, "c_0"'] \arrow[r, "c_0'"] \arrow[d] & G_1 & & G_n \\
& c_0(D_0) \arrow[ul, "\supseteq"] \arrow[ur] & \text{PB} & Persist_{G_1 \Rightarrow^* G_n} \arrow[ul, "\supseteq"'] \arrow[ur] & \\
& & Persist_{G_0 \Rightarrow^* G_n} \arrow[ul, "\supseteq"] \arrow[ur] \arrow[uull, dashed, bend left] \arrow[uurr, dashed, bend right] & &
\end{tikzcd}$$

Figure C.2: Commutative-squares formulation of a track morphism for multi-step derivations.

- $c(D)$ is the subgraph of G, constructed as the image of D under morphism c
- morphism $i : D \to c(D)$ has the same node/edge mappings as c
- morphism $c(D) \to G$ is an inclusion
- i^{-1} is a premoprhism that is the inverse of i – it is injective, surjective and total because i is, but it is not label-preserving as $c(D)$ is totally labelled whereas D may be partially labelled.
- the premorphism $c(D) \to H$ is the composition $c(D) \to H = c(D) \to^{i^{-1}} D \to^{c'} H$

We have the triangles (1) and (2) commute, the track morphism is the arrow $c(D) \to H$, and the graph $c(D)$ is the graph $Persist_{G \Rightarrow H}$ of Definition C.1.

□

Definition C.4 (Commutative formulation of multi-step *track*)**.** Given a zero-length derivation $G \Rightarrow G$ with span $G \leftarrow^{c} G \to^{c'} G$ of zero length, then its track is the identity id_G (as in Definition C.1). Given an derivation $G \Rightarrow^* H$ with span $G \leftarrow^c D \rightarrow^{c'} H$ of at least length 2, computing its track function (as a diagram) amounts to constructing the diagram in Figure C.2 as the combination of the commutative diagram of $track_{G_0 \Rightarrow G_1}$ and the track diagram of of $G_1 \Rightarrow^* G_n$:

- the pullback of $c(D) \rightarrow G_1 \leftarrow Persist_{G_1 \Rightarrow^* G_n}$ leads to a pair of morphisms $c(D) \leftarrow Persist_{G_0 \Rightarrow^* G_n} \rightarrow Persist_{G_1 \Rightarrow^* G_n}$, and this construction is always possible (Fact A.1.2) since the morphism $Persist_{G_1 \Rightarrow^* G_n} \rightarrow G_1$ is an inclusion and hence in $\mathcal{M}$, similar to the fact that $c_0(D_0) \rightarrow G_0$ is an inclusion.

- the arrows $G_0 \leftarrow Persist_{G_0 \Rightarrow^* G_n} \rightarrow G_n$ are defined as

$$\begin{aligned} Persist_{G_0 \Rightarrow^* G_n} \rightarrow G_0 &= Persist_{G_0 \Rightarrow^* G_n} \rightarrow c_0(D_0) \rightarrow G_0 \\ Persist_{G_0 \Rightarrow^* G_n} \rightarrow G_n &= Persist_{G_0 \Rightarrow^* G_n} \rightarrow Persist_{G_1 \Rightarrow^* G_n} \rightarrow G_n \end{aligned}$$

□

Lemma 16. *If a critical pair $cp : T_1 \Leftarrow S \Rightarrow T_2$ is strongly joinable according to Definition C.2, then it is strongly joinable according Definition 6.4.*

Proof. Let N be the pullback of $O_1 \rightarrow S \leftarrow O_2$ as in Definition 6.4. Let P_1 and P_2 be the graphs $Persist_{S \Rightarrow T_1}$ and $Persist_{S \Rightarrow T_2}$ of the critical pair's derivations. By the commutative formulation of track Definition C.3, we have that $P_1 \subseteq S$ and $P_2 \subseteq S$ are inclusions. Next, define morphisms

$$\begin{aligned} N \rightarrow P_1 &= N \rightarrow O_1 \rightarrow P_1 \\ N \rightarrow P_2 &= N \rightarrow O_2 \rightarrow P_2 \end{aligned}$$

Due to the definition of $Persist_{cp}$ for a critical pair as an intersection (Definition C.2), we have inclusions $Persist_{cp} \subseteq P_1$ and $Persist_{cp} \subseteq P2$ such that $Persist_{cp} \rightarrow P1 \rightarrow S = Persist_{cp} \rightarrow P2 \rightarrow S$.

Now, we get that

$$\begin{aligned} N \rightarrow P_1 \rightarrow S &= N \rightarrow O_1 \rightarrow P_1 \rightarrow S && \text{(definition of } N \rightarrow P_1\text{)} \\ &= N \rightarrow O_1 \rightarrow S && \text{(commutativity of } O_1P_1S\text{)} \\ &= N \rightarrow O_2 \rightarrow S && \text{(commutativity of pullback} N\text{)} \\ &= N \rightarrow O_1 \rightarrow P_2 \rightarrow S && \text{(commutativity of } O_1P_2S\text{)} \\ &= N \rightarrow P_2 \rightarrow S && \text{(definition of } N \rightarrow P_2\text{)} \end{aligned}$$

Now, since $Persist_{cp}$ is intersection over inclusions, it must also be a pullback. Hence we get a unique morphism $N \rightarrow Persist_{cp}$ such that :

$$N \rightarrow P_1 = N \rightarrow Persist_{cp} \rightarrow P_1 \tag{C.1}$$
$$N \rightarrow P_2 = N \rightarrow Persist_{cp} \rightarrow P_2 \tag{C.2}$$

Let P'_1 be the graph $Persist_{T_1 \Rightarrow^* X_1}$ of the joining derivation, and P'_2 defined analogously. By the commutative definition of track Definition C.4, $Persist_{S \Rightarrow^* X_1}$ is the *pullback* of $P_1 \rightarrow T_1$ and $Persist_{S \Rightarrow^* X_1} \rightarrow S = Persist_{S \Rightarrow^* X_1} \rightarrow P_1 \rightarrow S$.

$$\begin{tikzcd}
S & O_1 \arrow[l] \arrow[r] \arrow[dd] & T_1 & \Rightarrow^* & X_1 \\
 & & & P_1' \arrow[ul] \arrow[ur] & \\
 & P_1 \arrow[uul] \arrow[uur] & \text{PB} & & \\
 & & Persist_{S\Rightarrow^* X_1} \arrow[uuull, bend left] \arrow[ul] \arrow[uur] \arrow[uuurr, bend right] & & \\
 & & Persist_{cp} \arrow[u] \arrow[uul, bend left] \arrow[uuur, bend right] & N \arrow[l] &
\end{tikzcd}$$

Next, define morphisms

$$N \to P_1' = N \to Persist_{S\Rightarrow^* X_1} \to P_1$$
$$N \to Persist_{S\Rightarrow^* X_1} = N \to Persist_{cp} \to Persist_{S\Rightarrow^* X_1}$$

where we get the inclusion $Persist_{cp} \to Persist_{S\Rightarrow^* X_1}$ by the requirement of Definition C.2 that all items of $Persist_{cp}$ are defined in $track_{S\Rightarrow^* X_1}$. We get the same for $N \to Persist_{S\Rightarrow^* X_2}$.

Next, we get the equalities

$$\begin{aligned}
N \to P_1 \to T_1 &= N \to Persist_{cp} \to P_1 \to T_1 && \textit{Equation C.1}\\
&= N \to Persist_{cp} \to Persist_{S\Rightarrow^* X_1} \to P_1 \to T_1 && (Persist_{S\Rightarrow^* X_1}\text{uniqueness})\\
&= N \to Persist_{cp} \to Persist_{S\Rightarrow^* X_1} \to P_1' \to T_1 && (\text{comm of } Persist_{S\Rightarrow^* X_1})\\
&= N \to Persist_{cp} \to P_1' \to T_1 && (\text{def of } Persist_{cp} \to P_1')
\end{aligned}$$

Define the morphisms

$$N \to P_1' = N \to Persist_{cp} \to P_1'$$
$$N \to P_2' = N \to Persist_{cp} \to P_2'$$

We get that $N \to O_1 \to T_1 = N \to Persist_{S\Rightarrow^* X_1} \to P_1 \to T_1 = N \to Persist_{S\Rightarrow^* X_1} \to P_1' \to T_1 = N \to P_1' \to T_1$. Since we are ignoring labels, $P_1{}'$ can be considered as the *derived span* of $T_1 \Rightarrow^* X_1$. Therefore, the square $NO_1O_3T_1$ of Definition 6.4 commutes. Analogously for the square $NO_2O_4T_2$.

Finally, we show the commutativity of $N \to O_3 \to X_1 \to X_2 = N \to O_4 \to X_2$. Given that all tracks are equal on all items of $Persist_{cp}$ (Definition C.2), we have that $Persist_{cp} \to Persist_{S\Rightarrow^* X_1} \to X_1 \to X_2 = Persist_{cp} \to Persist_{S\Rightarrow^* X_2} \to X_2$. Define the morphisms

$$N \to O_3 = N \to Persist_{cp} \to O_3$$
$$N \to O_4 = N \to Persist_{cp} \to O_4$$

Note $N \to O_4 \to X_2 = N \to \mathit{Persist}_{cp} \to O_4 \to X_2 = N \to \mathit{Persist}_{cp} \to P_2' \to X_2$
Now, for the final step:

$$\begin{aligned}
N \to O_3 \to X_1 \to X_2 &= N \to \mathit{Persist}_{cp} \to O_3 \to X_1 \to X_2 && \text{(def of } N \to O_3\text{)}\\
&= N \to \mathit{Persist}_{cp} \to P_1' \to X_1 \to X_2 && (O_3 \text{ as Persist graph})\\
&= N \to \mathit{Persist}_{cp} \to \mathit{Persist}_{S \Rightarrow^* X_1} \to X_1 \to X_2 && \text{(PB uniqueness)}\\
&= N \to \mathit{Persist}_{cp} \to \mathit{Persist}_{S \Rightarrow^* X_2} \to X_2 && \text{(tracks are equal)}\\
&= N \to \mathit{Persist}_{cp} \to P_2' \to X_2 && \text{(PB uniqueness)}\\
&= N \to O_4 \to X_2
\end{aligned}$$

which completes the point that all squares commute.

The final consideration are the labels: this follows directly from the requirement that $X_1 \to X_2$ is an isomorphism and hence is label- and undefinedness-preserving. This completes the proof of equivalence. □

List of References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place EMF model transformations. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proc. International Conference on Model Driven Engineering Languages and Systems (MODELS 2010), Part I*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. doi:10.1007/978-3-642-16145-2_9.

[ACPS93] Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, 1993. doi:10.1145/174147.169807.

[AGG] AGG: The attributed graph grammar system. `http://user.cs.tu-berlin.de/~gragra/agg/`.

[Aßm96] Uwe Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proc. International Conference on Compiler Construction (CC '96)*, pages 121–135, 1996. doi:10.1007/3-540-61053-7_57.

[BAHT15] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. Analyzing conflicts and dependencies of rule-based transformations in Henshin. In Alexander Egyed and Ina Schaefer, editors, *Proc. Fundamental Approaches to Software Engineering (FASE 2015), Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015*, volume 9033 of *Lecture Notes in Computer Science*, pages 165–168. Springer, 2015. doi:10.1007/978-3-662-46675-9_11.

[Bak16] Christopher Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, The University of York, 2016. `http://etheses.whiterose.ac.uk/id/eprint/12586`.

[BG02] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs - theory, algorithms and applications*. Springer, 2002.

[BGK+17] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. Confluence of graph rewriting with interfaces. In Hongseok Yang, editor, *Proc. European Symposium on Programming Languages and Systems (ESOP 2017), part of ETAPS*, volume 10201 of *Lecture Notes in Computer Science*, pages 141–169. Springer, 2017. doi:10.1007/978-3-662-54434-1_6.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[BP12] Christopher Bak and Detlef Plump. Rooted graph programs. In *Proc. International Workshop on Graph Based Tools (GraBaTs 2012)*, volume 54 of *Electronic Communications of the EASST*, 2012.

[BP16] Christopher Bak and Detlef Plump. Compiling graph programs to c. In *Proc. International Conference on Graph Transformation (ICGT 2016)*, volume 9761 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2016. doi:10.1007/978-3-319-40530-8_7.

[BPR04] Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. In *Int. Workshop Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 2004.

[BS94] Franz Baader and Jörg H. Siekmann. Unification theory. In Dov M. Gabbay, Christopher J. Hogger, J. A. Robinson, and Jörg H. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Deduction Methodologies*, volume 2, pages 41–126. Oxford University Press, 1994.

[BS01] Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001. `http://www.cs.bu.edu/~snyder/publications/UnifChapter.pdf`.

[BTS00] Paolo Bottoni, Gabriele Taentzer, and Andy Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proc. IEEE International Symposium on Visual Languages (VL 2000)*, pages 59–60, 2000.

[Cor16] Andrea Corradini. On the definition of parallel independence in the algebraic approaches to graph transformation. In Milazzo et al. [MVW16], pages 101–111. doi:10.1007/978-3-319-50230-4_8.

[CR36] Alonzo Church and J Barkley Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. doi:10.1090/S0002-9947-1936-1501858-0.

[Dec17] Frederik Deckwerth. *Static Verification Techniques for Attributed Graph Transformations*. PhD thesis, Technische Universität Darmstadt, May 2017. `http://tuprints.ulb.tu-darmstadt.de/6150`.

[DKL$^+$16] Frederik Deckwerth, Géza Kulcsár, Malte Lochau, Gergely Varró, and Andy Schürr. Conflict detection for edits on extended feature models using symbolic graph transformation. In Julia Rubin and Thomas

Thüm, editors, *Proc. International Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE@ETAPS 2016)*, volume 206 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–31, 2016. doi:10.4204/EPTCS.206.3.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08), part of European Conferences on Theory and Practice of Software (ETAPS '08)*, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.

[Duf65] Richard J Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10(2):303–318, 1965. doi:10.1016/0022-247X(65)90125-3.

[EE08] Hartmut Ehrig and Claudia Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In *Proc. 4th International Conference on Graph Transformation (ICGT 2008)*, Lecture Notes in Computer Science, pages 194–210, 2008. doi:10.1007/978-3-540-87405-8_14.

[EEdL$^+$05] Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In *Proc. Fundamental Approaches to Software Engineering (FASE 2005)*, pages 49–63, 2005. doi:10.1007/978-3-540-31984-9_5.

[EEGH15] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015. doi:10.1007/978-3-662-47980-3.

[EEHP09] Hartmut Ehrig, Claudia Ermel, Frank Hermann, and Ulrike Prange. On-the-fly construction, correctness and completeness of model transformations based on triple graph grammars. In *Proc. 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, pages 241–255, 2009. doi:10.1007/978-3-642-04425-0_18.

[EEKR99] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations - Volume 2: Applications, Languages and Tools*. World Scientific, 1999. doi:10.1142/9789812815149.

[EEPR04] Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors. *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*. Springer, 2004.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. doi:10.1007/3-540-31188-2.

[EET11] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2011), part of ETAPS 2011*, volume 6603 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2011. doi:10.1007/978-3-642-19811-3_15.

[EGH$^+$12] Hartmut Ehrig, Ulrike Golas, Annegret Habel, Leen Lambers, and Fernando Orejas. $\mathcal{M}$-adhesive transformation systems with nested application conditions: Part 2: Embedding, Critical Pairs and Local Confluence. *Fundamenta Informaticae*, 118(1-2):35–63, 2012. doi:10.3233/FI-2012-705.

[EH86] Hartmut Ehrig and Annagret Habel. Graph grammars with application conditions. In *The Book of L*, pages 87–100. Springer Berlin Heidelberg, 1986. doi:10.1007/978-3-642-95486-3_7.

[EHK$^+$97] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part II: single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars and Computing by Graph Transformations - Volume 1: Foundations*, pages 247–312, 1997.

[EHKP91] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1(3):361–404, 1991. doi:10.1017/S0960129500001353.

[EHL$^+$10] Hartmut Ehrig, Annegret Habel, Leen Lambers, Fernando Orejas, and Ulrike Golas. Local confluence for rules with nested application conditions. In Hartmut Ehrig, Arend Rensink, Grzegorz Rozenberg, and Andy Schürr, editors, *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2010. doi:10.1007/978-3-642-15928-2_22.

[EHPP04] Hartmut Ehrig, Annegret Habel, Julia Padberg, and Ulrike Prange. Adhesive high-level replacement categories and systems. In Ehrig et al. [EEPR04], pages 144–160. doi:10.1007/978-3-540-30203-2_12.

[EK76] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In Antoni W.

Mazurkiewicz, editor, *Proc. Mathematical Foundations of Computer Science (MFCS 76)*, volume 45 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 1976. doi:10.1007/3-540-07854-1_188.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *Proc. Symposium on Switching and Automata Theory (SWAT '08)*, pages 167–180. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.11.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Ehrig et al. [EEPR04], pages 161–177. doi:10.1007/978-3-540-30203-2_13.

[GLEO12] Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. *Theoretical Computer Science*, 424:46–68, 2012. doi:10.1016/j.tcs.2012.01.032.

[GPdBG94] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994. doi:10.1109/69.298174.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996. doi:10.3233/FI-1996-263404.

[HHT02] Jan Hendrik Hausmann, Reiko Heckel, and Gabriele Taentzer. Detection of conflicting functional requirements in a use case-driven approach: A static analysis technique based on graph transformation. In *Proc. International Conference on Software Engineering (ICSE 2002)*, pages 105–115, 2002.

[HKT02] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, pages 161–176, 2002. doi:10.1007/3-540-45832-8_14.

[HP01] Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.

[HP02] Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.

[HP09] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009. doi:10.1017/S0960129508007202.

[HP12] Annegret Habel and Detlef Plump. $\mathcal{M},\mathcal{N}$-adhesive transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2012. doi:10.1007/978-3-642-33654-6_15.

[HP15] Ivaylo Hristakiev and Detlef Plump. A unification algorithm for GP 2. In *Graph Computation Models (GCM 2014), Revised Selected Papers*, volume 71 of *Electronic Communications of the EASST*, 2015. doi:10.14279/tuj.eceasst.71.1002, long version available at: `https://arxiv.org/abs/1705.02171`.

[HP16a] Ivaylo Hristakiev and Detlef Plump. Attributed graph transformation via rule schemata: Church-Rosser theorem. In Milazzo et al. [MVW16], pages 145–160. doi:10.1007/978-3-319-50230-4_11, long version available at: `http://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump16.Full.pdf`.

[HP16b] Ivaylo Hristakiev and Detlef Plump. Towards critical pair analysis for the graph programming language GP 2. In Phillip James and Markus Roggenbach, editors, *Recent Trends in Algebraic Development Techniques (WADT 2016), Revised Selected Papers*, volume 10644 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2016. doi:10.1007/978-3-319-72044-9_11, online version available at: `http://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.WADT16.pdf`, long version available at: `https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.WADT16.Long.pdf`.

[HP17a] Ivaylo Hristakiev and Detlef Plump. Checking graph programs for confluence. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations — STAF 2017 Collocated Workshops, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2017. doi:10.1007/978-3-319-74730-9_8, online version available at: `http://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.GCM17.pdf`, long version available at: `https://www.cs.york.ac.uk/plasma/publications/pdf/HristakievPlump.GCM17.Long.pdf`.

[HP17b] Ivaylo Hristakiev and Detlef Plump. A unification algorithm for GP 2 (Long Version). *ArXiv e-prints*, arXiv:1705.02171, 2017. `https://arxiv.org/abs/1705.02171`.

[HT06] Brent Hailpern and Peri L. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–462, 2006. doi:10.1147/sj.453.0451.

[Hue80] Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[Hue81] Gérard P. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 23(1):11–21, 1981.

[HW95] Reiko Heckel and Annika Wagner. Ensuring consistency of conditional graph rewriting — a constructive approach. *Electronic Notes in Theoretical Computer Science*, 2:118–126, 1995. doi:10.1016/S1571-0661(05)80188-4.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.

[KB70] Donald E. Knuth and Peter Bendix. Simple word problems in universal algebras. In John Leech, editor, *Proc. Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KB83] Donald E. Knuth and Peter Bendix. Simple word problems in universal algebras. In *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, Symbolic Computation, pages 342–376. Springer Berlin Heidelberg, 1983.

[KDL$^+$15] Géza Kulcsár, Frederik Deckwerth, Malte Lochau, Gergely Varró, and Andy Schürr. Improved conflict detection for graph transformation with attributes. In *Proc. Graphs as Models (GaM 2015)*, pages 97–112, 2015. doi:10.4204/EPTCS.181.7.

[KMP05] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. Graph-based specification of access control policies. *JCSS*, 71(1):1–33, 2005. doi:10.1016/j.jcss.2004.11.002.

[Lam09] Leen Lambers. *Certifying Rule-based Models using Graph Transformation*. PhD thesis, Berlin Institute of Technology, 2009.

[LE06] Leen Lambers and Hartmut Ehrig. Efficient detection of conflicts in graph-based model transformation. *Electronic Notes in Theoretical Computer Science*, 152:97–109, 2006. doi:10.1016/j.entcs.2006.01.017.

[LEO06] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict detection for graph transformation with negative application conditions. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2006. doi:10.1007/11841883_6.

[LEO08] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient conflict detection in graph transformation systems by essential critical pairs. *Electronic Notes in Theoretical Computer Science*, 211:17–26, 2008. doi:10.1016/j.entcs.2008.04.026.

[LS05] Stephen Lack and Pawel Sobocinski. Adhesive and quasiadhesive categories. *Informatique Théorique et Applications*, 39(3):511–545, 2005. doi:10.1051/ita:2005028.

[Mak77] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Matematiceskiĭ Sbornik*, 103:147–236, 1977. In Russian. English translation in *Math. USSR Sbornik*, 32:129–198, 1977.

[Men05] Tom Mens. On the use of graph transformations for model refactoring. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, pages 219–257, 2005. doi:10.1007/11877028_7.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817.

[MTR07] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software and System Modeling*, 6(3):269–285, 2007. doi:10.1007/s10270-006-0044-6.

[MVW16] Paolo Milazzo, Dániel Varró, and Manuel Wimmer, editors. *Revised Selected Papers of Software Technologies: Applications and Foundations (STAF 2016) Collocated Workshops*, volume 9946 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-50230-4.

[OL12] Fernando Orejas and Leen Lambers. Lazy graph transformation. *Fundamenta Informaticae*, 118(1-2):65–96, 2012. doi:10.3233/FI-2012-706.

[Pla99] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. In *Symposium on Foundations of Computer Science (FOCS 1999)*, pages 495–500. IEEE Computer Society, 1999.

[Plo72] Gordon Plotkin. Building-in equational theories. *Machine intelligence*, 7(4):73–90, 1972.

[Plo04] Gordon D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004. doi:10.1016/j.jlap.2004.03.009.

[Plu93] Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. John Wiley, 1993.

[Plu94] Detlef Plump. Critical pairs in term graph rewriting. In *Proc. Mathematical Foundations of Computer Science (MFCS'94)*, volume 841 of *Lecture Notes in Computer Science*, pages 556–566. Springer-Verlag, 1994. doi:10.1007/3-540-58338-6_102.

[Plu98] Detlef Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae*, 33(2):201–209, 1998. doi:10.3233/FI-1998-33204.

[Plu05] Detlef Plump. Confluence of graph transformation revisited. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer-Verlag, 2005. doi:10.1007/11601548_16.

[Plu09] Detlef Plump. The graph programming language GP. In *Proc. International Conference on Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009. doi:10.1007/978-3-642-03564-7_6.

[Plu10] Detlef Plump. Checking graph-transformation systems for confluence. In *Manipulation of Graphs, Algebras and Pictures: Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday*, volume 26 of *Electronic Communications of the EASST*, 2010.

[Plu12] Detlef Plump. The design of GP 2. In *Proc. International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, volume 82 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–16, 2012. doi:10.4204/EPTCS.82.1.

[Plu16] Detlef Plump. Reasoning about graph programs. In *Proc. International Workshop on Computing with Terms and Graphs (TERMGRAPH 2016)*, Electronic Proceedings in Theoretical Computer Science, pages 35–44, 2016. doi:10.4204/EPTCS.225.6.

[PP14] Christopher M. Poskitt and Detlef Plump. Verifying monadic second-order properties of graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2014)*, volume 8571 of *Lecture Notes in Computer Science*, pages 33–48. Springer-Verlag, 2014. doi:10.1007/978-3-319-09108-2_3.

[PS04] Detlef Plump and Sandra Steinert. Towards graph programs for graph algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 128–143. Springer-Verlag, 2004. doi:10.1007/978-3-540-30203-2_11.

[RET11] Olga Runge, Claudia Ermel, and Gabriele Taentzer. AGG 2.0 - new features for specifying and analyzing algebraic graph transformations. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *International Symposium on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2011), Revised Selected and Invited Papers*, volume 7233 of *Lecture Notes in Computer Science*, pages 81–88. Springer, 2011. doi:10.1007/978-3-642-34176-2_8.

[RLP$^+$14] Shekoufeh Kolahdouz Rahimi, Kevin Lano, Suresh Pillay, Javier Troya, and Pieter Van Gorp. Evaluation of model transformation approaches for model refactoring. *Science of Computer Programming*, 85:5–40, 2014. doi:10.1016/j.scico.2013.07.013.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations - Volume 1: Foundations*. World Scientific, 1997.

[SBG$^+$17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A usability-focused framework for EMF model transformation development. In Juan de Lara and Detlef Plump, editors, *Proc. International Conference on Graph Transformation (ICGT 2017)*, volume 10373 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2017. doi:10.1007/978-3-319-61470-0_12.

[Sch92] Klaus U. Schulz. Makanin's algorithm for word equations: Two improvements and a generalization. In *Proc. Word Equations and Related Topics (IWWERT '90)*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer-Verlag, 1992.

[Sie78] Jörg Siekmann. *Unification and Matching Problems*. PhD thesis, University of Essex, 1978.

[TGM98] Gabriele Taentzer, Michael Goedicke, and Torsten Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. International Workshop on Theory and Application of Graph Transformations (TAGT '98), Selected Papers*, pages 179–193, 1998. doi:10.1007/978-3-540-46464-8_13.

[Wel14] Ruud Welling. *Conflict Detection and Analysis for Single-Pushout High-Level Replacement*. Master thesis, University of Twente, 2014.