Verification of Graph Programs with Monadic Second-Order Logic

Gia Septiana Wulandari

PhD

University of York
Computer Science

January 2021

Abstract

In this thesis, we consider Hoare-style verification for the graph programming language GP 2. In literature, Hoare-style verification for graph programs has been studied by using extensions of nested conditions called E-conditions and M-conditions as assertions. However, E-conditions are only able to express first-order properties of GP 2 graphs, while M-conditions can only express properties of a non-attributed graph. Hence, there is still no logic that can express monadic second-order properties of GP 2 graphs. Moreover, both E-conditions and M-conditions may not be easy to comprehend by programmers used to formal specifications expressed in standard first-order logic.

Here, we present an approach to verify GP 2 graph programs with a standard monadic second-order logic. We show how to construct a strongest liberal postcondition with respect to a rule schema and a precondition. We then extend this construction to obtain a strongest liberal postcondition for arbitrary loop-free programs. Also, we show how to construct a precondition expressing successful execution of a loop-free program, and failing execution of a so-called iteration command. These constructions allow us to define a partial proof calculus that can handle a larger class of graph programs than what can be verified by the calculus that uses E-conditions and M-conditions as assertions.

Other than partial proof calculus whose assertions are monadic second-order logic, we also define semantic partial proof calculus. Similar calculus has been introduced in literature, but here we update the calculus by considering a GP 2 command that was not considered in existing work.

Contents

List of Tables

List of Figures

Acknowledgements

First of all, I am grateful to God for the opportunity that leads me to complete and present this thesis. I would also like to thank several people for their help and support during the development of this thesis.

This thesis would not have been completed without the continuous encouragement of my supervisor, Dr Detlef Plump, who provide guidance, encouragement, and suggestions throughout my PhD years. His support really helps me enhance my knowledge, especially in problem-solving. I would also like to thank my examiner, Dr Radu Calinescu, for his precious feedback during the viva and TAP meetings and Prof Reiko Heckel for his feedback and insight during the viva.

I am grateful for all love, patience, and support that has been given to me by my family. I really appreciate Nungki who gives his understanding, support, and encouragement, Arka and Declan who inspire me to be cheerful and always able to remove my stress; also mamah, who always support me. Each of them motivates me to reach the finish line of this journey. I would like to dedicate this thesis to my family, especially my mom, who always support me as best as she could. May she rest in peace.

I would also like to thank all of the friends I met at the University of York and in the city of York to make my life very pleasant during my time in York. I am very grateful for the kindness of the people in room CSE/215. I would like to say thank you to Robert for helping me review the thesis. Also, to my friends and family in Indonesia for the chats that can relieve my stress away.

Finally, I would like to acknowledge the Indonesia Endowment Fund for Education (LPDP) funding that makes my PhD journey possible, also Telkom University that gives me the opportunity to continue my study.

Declaration

I, Gia Septiana Wulandari, declare that this thesis titled, 'Verification of Graph Programs with Monadic Second-Order Logic' and the work presented in it are my own. This work was done wholly while in candidature for a research degree at this University and has not previously been presented or submitted for a degree or any other qualification at this University or any other institution. Where I have consulted or quoted the work of others, this is always clearly attributed as References.
Some parts of this thesis have been published within the following publications:

- Gia S. Wulandari and Detlef Plump. Verifying a copying garbage collector in GP 2. In *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, pages 479–494, 2018. doi:10.1007/978-3-030-04771-9.

- Gia S. Wulandari and Detlef Plump. Verifying graph programs with first-order logic. In *Proceedings of the Eleventh International Workshop on Graph Computation Models*, volume 330 of *Electronic Proceedings in Theoretical Computer Science*, pages 181–200, 2020. doi: 10.4204/EPTCS.330.

- Gia S. Wulandari and Detlef Plump. Verifying graph programs with first-order logic (extended version). *ArXiv e-prints*, arXiV:2010.14549 [cs.LO], 2020.

- Gia S. Wulandari and Detlef Plump. Verifying graph programs with monadic second-order logic. In *Proceeding of the 13th International Conference on Graph Transformation (ICGT 2020)*, volume 12150 of *Lecture Notes in Computer Science*, pages 257–275, 2020. doi: 10.1007/978-3-030-51372-6

Chapter 1

Introduction

We open this thesis by giving the motivation of our research and summarising our contributions. Then, we describe its structure, giving an overview of the chapters.

1.1 Motivation

Graphs are a natural way to represent complex situations. Graphs can model structures and their relations in a simple way so that they are relevant for many practical problems. In computer science, we can find the application of graphs in many areas. We may see them in data and control flow diagrams, entity-relationship and UML diagrams, Petri nets, pointer structures, and visualisation of the hardware architecture [3]. The use of mathematical graph theory in solving practical problems then motivated researchers to study algorithms that work on graphs, including graph transformation. Unlike studies in graph theory where in general, the structure of graphs is not changed, graph transformation is an approach for structural modifications of graphs by applying transformation rules [4]. Intuitively, graph transformation is a study about how to manipulate graphs with local changes expressed by rules.

Some programming languages have been built to facilitate the application of graph transformations [5–11]. In his thesis, we focus on the graph programming language GP 2 [11]. GP 2 is a development made from the minimal and computationally complete core language that was introduced by Habel and Plump [12]. The design of GP 2 was published in [2] and the language has implemented by Christopher Bak as documented in this PhD thesis [11]. One novel aspect of GP 2, called root nodes, can be used to obtain constant time graph matching. Another new feature is so-called marks which help to control the rule schema applications. The power of GP 2 lies in its syntax that is simple, yet able to program every computable graph function. This implies that graph programs also facilitate formal reasoning about programs, which has been studied since 2010 by Poskitt and Plump [1, 13–16].

They use Hoare Logic for reasoning and define so-called E-conditions and M-conditions to express graph properties which are used for pre- and postconditions.

Both E- and M-conditions extend nested graph conditions [17] with support for expressions. The conditions are constructed based on graph morphisms, which may be difficult to understand by average programmers. For those who are familiar with standard logic, E- and M-conditions may not be easy to comprehend due to their notation. For example, if we want to express that all nodes are labelled with an integer, we can use the E-condition $\forall($  $, \exists($ $\mid$ `int(a)`$)) \wedge \forall($ $, \exists($ $\mid$ `int(a)`$)) \wedge \forall($ $, \exists($ $\mid$ `int(a)`$)) \wedge \forall($ $, \exists($ $\mid$ `int(a)`$)) \wedge \forall($ $, \exists($ $\mid$ `int(a)`$))$. The existence of two quantifiers to express a universal property of nodes may appear unnatural from the perspective of standard logic. In standard logic, the use of one quantifier would suffice. For example, in the logic we introduce in this thesis, the above condition can be simply written as $\forall_v x(\mathsf{int}(\mathsf{l}_v(x)))$. Moreover, E-conditions are limited to the expression of first-order properties of GP 2 graphs. M-conditions are able to express monadic second-order properties of plain graphs, but not of GP 2 graphs because they do not contain expressions, marks, or roots.

Another limitation of the Hoare-style verification method introduced in [1] is that it can only handle programs where every loop-body and every condition of an `if/try` branching command is a rule set call. Hence, the approach cannot be used to verify programs with nested loops, e.g. most of graph programs in [11, 18, 19]. The subset of graph programs that has no nested loops actually already computationally complete [20]. This means that there always exists an equivalent graph program without nested loops for any graph programs with nested loops. However, it is not clear how to get the equivalent program. In addition, many programs are constructed naturally by using nested loops, e.g. programs that use depth-first search approach [11] to obtain linear running time [19]. There may not exist an equivalent graph program without nested loops that is as efficient as the one with nested loops because it is impossible to encode the DFS strategy without nested loops.

In this thesis, we present the use of standard logic to express monadic second-order properties of graph programs. We choose standard logic since it may is easier to comprehend by programmers who does not familiar with morphisms. Moreover, standard logic may be useful for future work because of the large range of theorem proving environments such as Isabelle [21, 22], Coq [23], or Z3 [24]. for standard logic and the logic's vast literature. Using monadic second-order logic, we can also verify graph programs with stronger properties, such as the existence of a path or connectedness. The limitation of programs that the proof calculi in [1] can verify also motivated us to extend the calculi so that we can verify some nested loops, which are common used in practice, especially for efficient graph programs.

1.2 Thesis aims

Based on the motivations described above, we derive our main objective of this study, to use monadic second-order logic in reasoning about graph programs. We then elaborate the aim into three sub-aims, that are:

1. To define monadic second-order formulas based on standard logic that can express monadic second-order properties of GP 2 graphs
2. To use monadic second-order formulas as assertions in the Hoare-style verification of the partial correctness of graph programs
3. To extend the proof calculus of [1] so that it can handle a larger class of programs, in particular programs with nested loops

1.3 Thesis contributions

This thesis makes contributions regarding the use of monadic second-order logic in the verification of graph programs. These contributions are described below.

1. *Monadic second-order formulas for graph programs.*
   In this thesis, we define monadic second-order formulas that can be used to express the counting monadic second-order properties of GP 2 graphs (e.g. connectedness of a graph, 2-colourability of a graph). To the best of our knowledge, monadic second-order formulas have never been used as assertions to verify GP 2 programs. We have reported an earlier version of this work in [25].
2. *Monadic second-order formulas as assertions for graph program verification.*
   This thesis shows how to construct a strongest liberal postcondition with respect to a given GP 2 conditional rule schema and a precondition in the form of a monadic second-order formula. Moreover, we show how to construct the following monadic second-order formulas:
   - a strongest liberal postcondition of a loop-free program
   - a precondition of a loop-free program that must be satisfied by a graph to have a successful execution (i.e. produce a graph)
   - a precondition of a so-called iteration command that must be satisfied by a graph to have a failing execution

For the construction of a strongest liberal postcondition, we mainly use a similar approach as the construction of a weakest liberal precondition in [1, 26] and extend it to MSO logic. We take the main idea of generating a left and right-application condition with respect to the given assertion and rule schema. However, since we work with standard logic instead of nested conditions, we use a different technique to generate the conditions. In addition, here we handle more attributes of graphs such as `any`-mark and rootedness.

3. *Extension of Hoare calculi for graph programs.*
   This thesis presents an extension of the partial correctness proof calculus of [1]. The partial correctness proof calculus presented here can be used to verify programs whose loop bodies are iteration commands, and the condition of branching commands are loop-free programs. With this calculus, we are able to verify certain graph programs with nested loops such as `connectedness` [19], which has been proven that it is run in linear time on bounded degree input graph, can not be verified by the calculus. Here, we show that we are able to verify graph programs that exist in literature [11, 18, 19], which can not be verified by the framework of [1].

1.4 Thesis structure

The rest of the thesis is divided into several chapters, as follows:

Chapter 2 gives the context of this thesis. This chapter examines related work that is used as a basis theory for this thesis. This chapter presents preliminaries about graph transformations and the graph programming language GP 2 to give the basic information about graph programs we verify in this thesis. In addition, it describes Hoare-style verification of graph programs that have been done in the literature to give an idea of how we can verify a graph program with respect to the given assertions. Finally, this chapter presents how standard logic can be used to express properties of a graph.

Chapter 3 presents monadic second-order formulas that can be used to express properties of GP 2 graphs. Moreover, we show how we can use the formulas to express properties of GP 2 graphs that depend on morphisms. Here, we also show properties of monadic second-order formulas by identifying some lemmas and provide proofs of the lemmas.

Chapter 4 describes the intuition on calculating a strongest liberal postcondition. It first gives us the definition of a strongest liberal postcondition and how can we obtain one over a given rule schema and precondition (in first-order or monadic second-order formulas). Here,

we present properties that should hold within the construction to obtain a strongest liberal postcondition.

Chapter 5 presents a construction of a strongest liberal postcondition with respect to a given precondition and a rule schema. Here, we limit the pre- and postconditions to closed first-order so that we can focus on the main idea of transforming the conditions. This chapter also shows proofs that the construction is sound. Moreover, it shows how we can use the construction to obtain a weakest liberal precondition over a postcondition and a rule schema.

Chapter 6 gives the complete construction of a strongest liberal postcondition with respect to a monadic second-order formula (as a precondition) and a rule schema. It describes how we can extend the construction from the previous chapter to monadic second-order formulas, knowing the general ideas of the construction. Here, we start from an example to illustrate how we can extend the construction before we define the construction formally. In addition, this chapter also discusses the size of the obtained strongest liberal postcondition.

Chapter 7 presents proof calculi for graph program verification. This chapter shows how we can obtain a strongest liberal postcondition from a given precondition and a loop-free program. Moreover, we show how we can express properties that must be satisfied by a graph so that the execution of a loop-free program may yield a graph as a result. Here, we show how to obtain a monadic second-order formula to express properties of graphs where the execution of a graph program (of a subclass of programs) on the graphs may yield failure. This chapter also presents semantic proof calculus and syntactical proof calculus, in the sense of partial correctness. Semantic proof calculus allows arbitrary assertions as pre- and postconditions. For syntactic proof calculus, we limit the pre- and postconditions to closed monadic second-order formulas.

Chapter 8 demonstrates the use of our syntactic proof calculus by proving some graph programs with various specifications. Graph programs we use as examples in this chapter are: `vertex-colouring`, `transitive-closure`, `2-colouring`, and `connectedness`. Three from the four programs contain nested loops, and the program `connectedness` contains a rooted node. Moreover, for the verification of programs `transitive-closure`, `2-colouring`, and `connectedness`, we use specifications in monadic second-order formula which cannot be expressed in first-order logic.

Chapter 9 discusses the soundness and completeness of the partial correctness proof calculi provided in Chapter 6. It shows that both semantic and syntactic proof calculus is sound. The semantic proof calculus is proved to be relatively complete, but the question whether the syntactic proof calculus is relative complete remains open. However, we give a conjecture regarding the relative completeness of the syntactic proof calculus in this chapter.

Chapter 10 summarises the findings of this thesis. Here, we also propose some future work in several areas, including the development of a theorem prover to support our syntactic proof calculus, an extension to total correctness proof calculus, and monadic second-order transductions for expressing morphisms between initial and final graphs.

Chapter 2

Context

This chapter introduces fundamental theories about graph programs, verification of graph programs, and monadic second-order logic. This chapter is a summary of the existing literature to support the main result of this thesis.

2.1 Graph programming

This section introduces GP 2 programming language, which is based on graph transformation systems. A graph transformation is a rule-based modification of graphs [3]. It is a technique to obtaining a new graph from a given graph by a system that consists of rules. A single-step transformation is done by considering the morphisms between the given rule and graph. With its ability to change the structure, graph transformation has been applied in many areas, such as model transformation [4]. Graph transformation has also been used for describing biological or chemical processes [27, 28].

2.1.1 Graphs and graph morphisms

A graph is a flexible structure in representing objects and relations between them. There are three main components in a graph: nodes, edges, and labels. In applications, nodes are usually used to represent objects, edges to represent relations between objects, and labels to store information that is needed about the objects or the relations. This study uses graphs with labelled nodes and with directed and labelled edges, where parallel edges and loops are acceptable. However, we sometimes use the blank label ($\square$), which is usually not written on the graph.

In mathematics, a graph is commonly defined as pair of vertices V and edges $E \subseteq V \times V$. However, it is not expressive enough to define parallel edges and labels. Moreover, in GP 2 we also need to define nodes and edges' labels, also rootedness of nodes in a graph.

Definition 2.1 (Label and marks)**.** *A label and mark alphabet* $\mathcal{C} = \langle \mathcal{C}_V^L, \mathcal{C}_E^L, \mathcal{C}_V^M, \mathcal{C}_E^M \rangle$ is a set comprising a set $\mathcal{C}_V^L$ of node labels, a set $\mathcal{C}_E^L$ of edge labels, a set $\mathcal{C}_V^M$ of node marks, and a set $\mathcal{C}_E^M$ of edge marks. □

In literature, such as in [1, 29], marks are included in labels, so that a graph consists of a set of nodes, a set of edges, source and target functions, and labelling functions. Here, for convenience, we separate labels and marks. As in [29], we include root function in a graph to define the rootedness of nodes in the graph.

Definition 2.2 (Graph over label alphabet)**.** *A graph over label and mark alphabet* $\mathcal{C}$ is a system $G = \langle V_G, E_G, s_G, t_G, l_G^V, l_G^E, m_G^V, l_G^E, p_G \rangle$ comprising a finite set V_G of nodes, a finite set E_G of edges, source and target functions $s_G, t_G : E_G \to V_G$, a partial node labelling function $l_G^V : V_G \to \mathcal{C}_V^L$, a partial edge labelling function $l_G^E : E_G \to \mathcal{C}_E^L$, a partial node marking function $m_G^V : V_G \to \mathcal{C}_V^M$, a partial edge marking function $m_G^E : E_G \to \mathcal{C}_E^M$, and a partial rootedness function $p_G : V_G \to \{0, 1\}$. For a node (or edge) i in G, $l_G^V = \perp$ iff $m_G^V = \perp$ (or $l_G^E = \perp$ iff $m_G^E = \perp$), where $\perp$ represents undefined function. A *totally labelled graph* is a graph whose functions are total. □

One may think that a natural way to express rootedness of a graph is by using a set of rooted nodes instead of a partial function. However, undefined rootedness is useful in graph programs [29]. Graphically, we use circles to represent nodes and arrows for edges where the arrow's head is attached to the edge's target while the tail is attached to the source. Labels of nodes are written inside the circle, and labels of edges are written next to the arrow. Node identifiers are written outside the circle or not written at all, while edge identifiers are not written.

Example 2.1 (A graph)**.** Let consider a graph $G = \langle \{1, 2, 3, 4\}, \{a, b, c, d\}, \{a \mapsto 1, b \mapsto 1, c \mapsto 2, d \mapsto 2\}, \{a \mapsto 1, b \mapsto 2, c \mapsto 3, d \mapsto 3\}, \{1 \mapsto \alpha, 2 \mapsto \beta, 3 \mapsto \alpha, 4 \mapsto \gamma\}, \{a \mapsto \square, b \mapsto \square, c \mapsto \square, d \mapsto \square\}, \{1 \mapsto \mathit{none}, 2 \mapsto \mathit{none}, 3 \mapsto \mathit{none}, 4 \mapsto \mathit{none}\}, \{a \mapsto \mathit{none}, b \mapsto \mathit{dashed}, c \mapsto \mathit{none}, d \mapsto \mathit{none}\}, \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 0, 4 \mapsto 0\} \rangle$ over the label alphabet $\mathcal{C} = \langle \{\alpha, \beta, \gamma\}, \{\square\}, \{\mathit{none}\}, \{\mathit{none}, \mathit{dashed}\} \rangle$. We can represent the graph as an abstract graph as can be seen in Figure 2.1. In this example, we do not write node and edge identifiers as well as blank label.

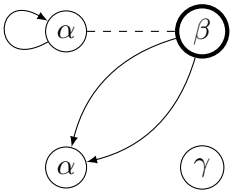
There are two notations for the class of graphs over a label alphabet: the class of all graphs over the label alphabet (including graphs that are not totally labelled), and the class of all totally labelled graphs over the label alphabet.

$V_G = \{1, 2, 3, 4\}$
$E_G = \{a, b, c, d\}$

	a	b	c	d
s_G	1	1	2	2
t_G	1	2	3	3
l_G^E	□	□	□	□
m_G^E	none	dashed	none	none

	1	2	3	4
l_G^V	α	β	α	γ
l_G^E	none	none	none	none

(a) Concrete graph



(b) Abstract graph

FIGURE 2.1: Graph G from Example 2.1

Definition 2.3 (Classes of graphs [1])**.** Let $\mathcal{C}$ be a label alphabet. We denote by $\mathcal{G}(\mathcal{C}_\perp)$ the class of all graphs over $\mathcal{C}$, while $\mathcal{G}(\mathcal{C})$ denotes the class of all totally labelled graphs over $\mathcal{C}$. □

To show a relation between two graphs, we often use graph morphisms to express structure-preserving mappings between graphs. In graph morphisms, structures, labels, and rootedness are preserved in the relation [29]. In GP 2, in addition to graph morphism, we also have graph premorphisms which is similar to graph morphisms but not considering node and edge labels. Formally, graph morphism is defined in Definition 2.4. The definition is similar to the definition of graph morphisms defined in [29], but here, again, we separate labels and marks.

Definition 2.4 (Graph morphisms)**.** Let G and H be graphs. A *graph morphism* $g : G \to H$ is a pair of mapping $g = \langle g_V : V_G \to V_H, g_E : E_G \to E_H \rangle$ such that for all nodes and edges in G, sources, targets, labels, marks, and rootedness are preserved. That is:

1. $g_V \circ s_G = s_H \circ g_E$,
2. $g_V \circ t_G = t_H \circ g_E$,
3. $l_H^V(g_V(x)) = l_G^V(x)$ for all $x \in V_G$ such that $l_G^V(x) \neq \perp$,
4. $l_H^E(g_E(x)) = l_G^E(x)$ for all $x \in E_G$ such that $l_G^E(x) \neq \perp$,
5. $m_H^V(g_V(x)) = m_G^V(x)$ for all $x \in V_G$ such that $m_G^V(x) \neq \perp$,
6. $m_H^E(g_E(x)) = m_G^E(x)$ for all $x \in E_G$ such that $m_G^E(x) \neq \perp$,
7. $p_H(g_V(v)) = p_G(v)$ for all $v \in V_G$, such that $p_G(v) \neq \perp$,

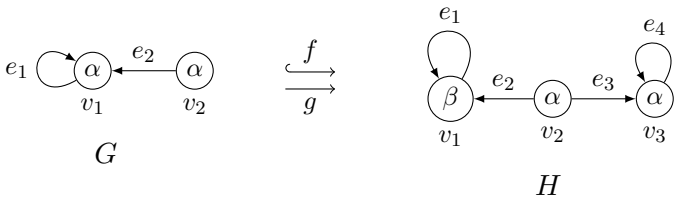
where $\circ$ denotes function composition and $\perp$ denotes undefined value. A graph morphism g is injective (surjective) if both g_V and g_E are injective (surjective). A graph morphism

$g : G \to H$ is an *isomorphism* if g is both injective and surjective, also satisfies $l_H^V(g_V(v)) = \perp$ for all nodes v with $l_G^V(v) = \perp$ and $p_H(g_V(v)) = \perp$ for all nodes v with $p_G(v) = \perp$. Furthermore. we call a morphism g as an *inclusion* if $g(x) = x$ for all x in G. □

Definition 2.5 (Premorphisms [1])**.** Let us consider graphs L and G. A *premorphism* $g : L \to G$ consists of two injective functions $g_V : V_L \to V_G$ and $g_E : E_L \to E_G$ that preserves sources, targets, and rootedness. □

Intuitively, a premorphism $g : L \to G$ is a structure-preserving map that ignores the labels and rootedness of L and G. On the other hand, a morphism $g : L \to G$ is a structure-preserving map that requires the preservation of labels and rootedness if defined in L.

Example 2.2 (Graph morphism)**.** Let us consider graphs in Figure 2.2. There are two graph morphisms f and g from graph G to graph H where f is an injective graph morphism while g is a non-injective graph morphism. We can see from the example that both f and g preserve sources, targets, and labels.



$$f = \left\langle f_V : \begin{Bmatrix} v_1 \mapsto v_3 \\ v_2 \mapsto v_2 \end{Bmatrix}, f_E : \begin{Bmatrix} e_1 \mapsto e_4 \\ e_2 \mapsto e_3 \end{Bmatrix} \right\rangle$$

$$g = \left\langle g_V : \begin{Bmatrix} v_1 \mapsto v_3 \\ v_2 \mapsto v_3 \end{Bmatrix}, g_E : \begin{Bmatrix} e_1 \mapsto e_4 \\ e_2 \mapsto e_4 \end{Bmatrix} \right\rangle$$

FIGURE 2.2: An injective graph morphism f and a non-injective graph morphism g

Example 2.3 (Injective graph premorphism)**.** Considering the graphs and morphism g of Figure 2.2. The morphism g and

$$h = \left\langle h_V : \begin{Bmatrix} v_1 \mapsto v_1 \\ v_2 \mapsto v_2 \end{Bmatrix}, h_E : \begin{Bmatrix} e_1 \mapsto e_1 \\ e_2 \mapsto e_2 \end{Bmatrix} \right\rangle$$

are injective premorphisms.

2.1.2 Graph transformation systems

Graph transformation is a study about how to manipulate graphs with rules to change the structure without erasing important elements that are needed for solving a problem. Graph transformations derive a graph based on a set of rules.

2.1.2.1 Rules and direct derivation

The idea of graph transformation is to manipulate graphs by rules $L \Rightarrow R$ where both L and R are graphs. The transformation is done by applying rule $L \Rightarrow R$ to graph G, that is, by replacing a subgraph of G that matches with L with R. There are two major approaches in graph transformation, that are single pushout (SPO) and double pushout (DPO) approach [3]. The latter is more widely used due to the extra requirement of the dangling condition, that is, the constraint that prevents the creation dangling edges by a rule application (see Definition 2.8). In this thesis as well, we are focus only on the DPO approach with injective matching.

Definition 2.6 (Pushout [1])**.** Let us consider graph morphisms $A \to B$ and $A \to C$, the pushout of these morphisms is formed by the graph D and graph morphisms $B \to D$ and $C \to D$ as in Figure 2.3 if the following properties are satisfied:

1. Commutativity. $A \to B \to D = A \to C \to D$
2. Universal Property. For all graph morphisms $B \to D'$ and $C \to D'$ such that $A \to B \to D' = A \to C \to D'$, there is a unique graph morphism $D \to D'$ such that $B \to D \to D' = B \to D'$ and $C \to D \to D' = C \to D'$. □

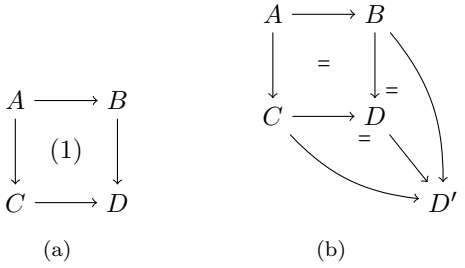


FIGURE 2.3: A pushout and the universal property of pushouts [1]

Intuitively, D can be obtained by gluing B and C in a common part A. The commutativity means that whenever the graphs B and C has common items (nodes or edges) in A, these items are identified in D as well. On the other hand, the universal property asserts that no other items that are not in B or C exist in D [30].

Example 2.4 (Pushout)**.** Figure 2.4 shows a simple pushout where all morphisms are injective. All nodes and edges are labelled with the blank label.

Habel, Müler, and Plump in [31] show that DPO with injective matching makes the double-pushout approach more expressive than the non-injective matching. Injective matching makes the double-pushout approach more expressive because we can have finer control on transformation than in the traditional framework.

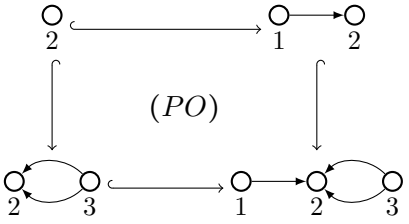
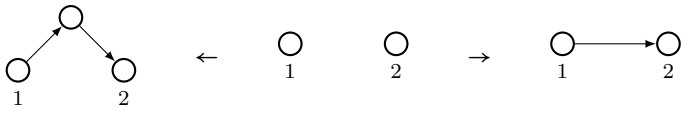


FIGURE 2.4: A pushout

Definition 2.7 (Rules [1])**.** A rule $r : \langle L \leftarrow K \rightarrow R \rangle$ over a label alphabet $\mathcal{C}$ comprises totally labelled graphs $L, K, R \in \mathcal{G}(\mathcal{C})$ and inclusions $K \rightarrow L$ and $K \rightarrow R$. We call L the left-hand graph, R the right-hand graph, while K is the interface of r. □

A rule r in the DPO approach is described by three components: left-hand graph L, interface K, and right-hand graph R of r. However, rules are sometimes written as $r : L \Rightarrow R$, without showing the interface K. In such cases, nodes that correspond in L and R are numbered. These nodes must be matched and must be preserved. We then establish that K consists of these nodes. Figure 2.5 shows an example of a rule, with and without interface K written in the rule. We then establish that K consists of nodes that are preserved, such that $E_K = \varnothing$.

(a) Interface K is not written



(b) Interface K is written

FIGURE 2.5: A rule $r : L \leftarrow K \rightarrow R$

Intuitively, a rule $r : L \leftarrow K \rightarrow R$ transforms a graph G by finding a subgraph in G that is isomorphic to L, let say by isomorphism g, and change elements $g(L - K)$ to $R - K$. This means deletion of a node may exist. We need to ensure that deletion of a node must be accompanied with deletion of edge(s) incident to it so that we can not have a dangling edge after rule applications.

Definition 2.8 (Dangling condition; match [1])**.** Let $r : L \leftarrow K \rightarrow R$ be a match, G be a totally labelled graph, and $g : L \rightarrow G$ be an injective graph morphism. The *dangling condition* requires that no edge in $G - g(L)$ is incident to any node in $g(L - K)$. When the dangling condition is satisfied by g, we say that g is a *match* for r. □

Definition 2.9 (Application of a rule [11])**.** The application of a rule $r : \langle L \leftarrow K \rightarrow R \rangle$ to a graph G with a match $g : L \rightarrow R$ yields a graph M, written $G \Rightarrow_{r,g} M$, if $M \cong H$, where H is constructed from G by applying these steps:

1. Delete all nodes and edges in $g(L - K)$, to produce an intermediate graph D.
2. Add all nodes and edges from $R - K$ with retaining their labels to D, to produce a graph H.
3. The source of a new edge $e \in E_R - E_K$, $s_H(e)$, is defined as $s_R(e)$ if $s_R(e) \in V_R - V_K$. Otherwise, $s_H(e) = g_V(s_R(e))$.
4. Target functions are defined analogously. □

The above definition of application condition is important in practical use. However, it may be hard to use it in reasoning at the abstract level. Hence, we also need the definition of rule application in the DPO approach.

Definition 2.10 (Direct derivation; comatch [1])**.** Let us consider a rule $r : \langle L \leftarrow K \rightarrow R \rangle$, a graph G, and an injective match $g : L \rightarrow G$. A direct derivation from G to H, written $G \Rightarrow_{r,g} H$ or more commonly $G \Rightarrow_r H$, is a pair of natural pushouts, or a double pushout represented in Figure 2.6. The morphism $R \rightarrow H$ is then called the *comatch* of r.

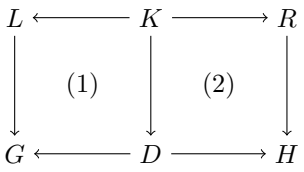


FIGURE 2.6: A direct derivation

□

When no ambiguity arises, direct derivation is also denoted as $G \Rightarrow_r H$, $G \Rightarrow H$, and $G \Rightarrow_{\mathcal{R}} H$ if r belongs to a set of rules $\mathcal{R}$. If $G \cong H$ or we have a sequence of direct derivation

$$G = G_0 \Rightarrow_{r_1} G_1 \Rightarrow_{r_2} \Rightarrow \ldots \Rightarrow_{r_n} G_n = H$$

with $r_1, r_2, \ldots, r_n \in \mathcal{R}$, we write $G \Rightarrow^*_{\mathcal{R}} H$ or $G \Rightarrow^* H$; denoting that G derives H in zero or more direct derivations.

2.1.2.2 Rules with relabelling

The transformation we discussed until now requires L, K, and R be totally labelled graphs. Relabelling of an edge can easily be done by deleting and recreating the edge with a new label.

However, relabelling a node is not as easy because of the dangling condition requirements. One approach to handle relabelling problem this is presented in [32, 33]. The modification allows K to be a partially labelled graph such that unlabelled nodes in the interface can have different labels in left and right-hand graph of the rule.

Definition 2.11 (Rule with relabelling [1])**.** A rule (with relabelling) $r : \langle L \leftarrow K \rightarrow R \rangle$ over a label alphabet $\mathcal{C}$ comprises totally labelled graphs $L, R \in \mathcal{G}(\mathcal{C})$, a partially labelled graph $K \in \mathcal{G}(\mathcal{C}\bot)$, and inclusions $K \rightarrow L$ and $K \rightarrow R$. We call L the left-hand graph, R the right-hand graph, while K the interface of r. □

As before, we sometimes do not write the interface in the rule. In this case, we establish the interface comprises exactly the numbered nodes in left and right-hand graph with no edge in the interface. Moreover, all nodes in the interface are unlabelled.

The application of a rule with relabelling is similar to the application of a rule defined in Definition 2.12 , but with additional treatment for unlabelled nodes [11].

Definition 2.12 (The application of a rule with relabelling [11])**.** The application of a rule (with relabelling) $r : \langle L \leftarrow K \rightarrow R \rangle$ to a graph G with a match $g : L \rightarrow R$ yields a graph M, written $G \Rightarrow_{r,g} M$, if $M \cong H$, where H is constructed from G by applying these steps:

1. Delete all nodes and edges in $g(L - K)$ and remove the labels of the images of the unlabelled nodes in K, to produce an intermediate graph D.
2. Add all nodes and edges from $R - K$ with retaining their labels to D, to produce a graph H.
3. The source of a new edge $e \in E_R - E_K$, $s_H(e)$, is defined as $s_R(e)$ if $s_R(e) \in V_R - V_K$. Otherwise, $s_H(e) = g_V(s_R(e))$.
4. Target functions are defined analogously.
5. For each unlabelled node $v \in K$, $l^V_H(g_V(v))$ is defined as $l^V_R(v)$. □

In the traditional approach, a direct derivation requires two squares in Figure 2.6 to be pushouts. For rule with relabelling, a direct derivation requires these squares to be pullbacks as well.

Definition 2.13 (Pullbacks [1])**.** Let us consider graph morphisms $B \rightarrow D$ and $C \rightarrow D$, the pullback of these morphisms is formed by the graph A and graph morphisms $A \rightarrow B$ and $A \rightarrow C$ as in Figure 2.7 if the following properties are satisfied:

1. Commutativity. $A \rightarrow B \rightarrow D = A \rightarrow C \rightarrow D$
2. Universal Property. For all graph morphisms $A' \rightarrow B$ and $A' \rightarrow C$ such that $A' \rightarrow B \rightarrow D = A' \rightarrow C \rightarrow D$, there is a unique graph morphism $A' \rightarrow A$ such that $A' \rightarrow A \rightarrow B = A' \rightarrow B$ and $A' \rightarrow A \rightarrow C = A' \rightarrow C$. □

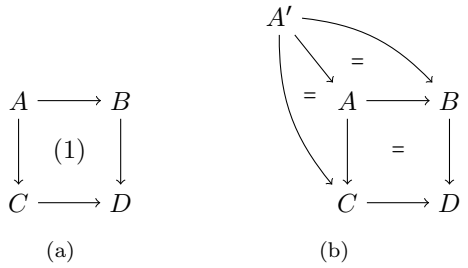


FIGURE 2.7: A pullback and the universal property of pullbacks [1]

Definition 2.14 (Natural pushouts [34])**.** A diagram (1) as in Figure 2.7 is a natural pushout if it is both a pushout and a pullback. □

The natural double-pushout construction such that we have natural double-pushout is described in [11, 29], that are:

1. To obtain D, remove all nodes and edges in $g(L - K)$ from G. For all $v \in V_K$ with $l_K^V(v) = \perp$ and $m_K^V(v) = \perp$, define $l_D^V(g_V(v)) = \perp$ and $m_D^V(g_V(v)) = \perp$. Also, define $p_D(g_V(v)) = \perp$ for all $v \in V_K$ where $p_K(v) = \perp$.
2. Add all nodes and edges, with their labels, marks, and rootedness, from $R - K$ to D. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(E) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.
3. For all $v \in V_K$ with $l_K^V(v) = \perp$ and $m_K^V(v) = \perp$, define $l_H^V(g_V(v)) = l_R^V(v)$ and $m_H^V(g_V(v)) = m_R^V(v)$. Also, for the injective morphism $R \to H$ and $v \in V_K$ where $p_K(v) = \perp$, define $p_H(g_V^*(v)) = p_R(v)$. The resulting graph is H.

Direct derivations transform a host graph via a rule whose the left and right-hand graph are totally labelled host graphs. However, a conditional rule schema contains a condition, and its left or right-hand graph may not be a host graph. Hence, we need some additional requirements for the application of a conditional rule schema on a host graph.

2.1.3 The GP 2 programming language

Graph programs (GP 2) is a graph programming language based on graph transformation and was developed from the minimal and computationally complete core language introduced by Habel and Plump [12]. The idea to base the language on rule schemata was introduced in [35], in order to allow computations on labels. Further refinements of the language are described in [33, 34]. The power of GP lies in its simple syntax and complete formal semantics that facilitates formal reasoning. The implementation of GP is described in [36]. GP then was developed into GP 2 whose initial design is presented in [2]. The up to date documentation

of GP 2 and a description of its implementation can be found in the thesis [11]. A new concept introduced in [11] is so-called root node which drastically reduce the search space for rule matches.

2.1.3.1 Graphs in GP 2

There are two kinds of graphs in GP 2: host graphs and rule graphs. A label in a host graph is a pair of list and mark, while a label in a rule graph is a pair of expression and mark. Input and output of graph programs are host graphs, while graphs in GP 2 rules are rule graphs.

We differentiate labels for host graphs and rule graphs because we only want constants as labels of items in the input and output graphs of a program. However, labels in a rule graph are expressions that may contain variables. By using expressions as labels, a rule can match with varying input graphs with different node and edge labels.

In GP 2 graphs, nodes and edges can be marked. Marks are useful in algorithms because they can let us track which nodes/edges we have visited. Items in GP 2 graphs can be marked with red, blue, green, grey, and dashed. Grey is reserved for nodes, while dashed is reserved for edges. Dashed is used instead of grey to make the difference with unmarked edges be obvious. In a rule graph, we may also have a node/edge marked `any` as a wildcard to be able to find a match with any mark other than `none`.

In literature, such as in [11], the set of marks does not include the mark `none`. Here, we include the mark `none` to differ an unmarked node and node with undefined mark.

Definition 2.15 (GP 2 host graph marks and labels)**.** A set of node marks, denoted by $\mathbb{M}_V$, is the set {`none`, `red`, `blue`, `green`, `grey`}. The set of edge marks, denoted by $\mathbb{M}_E$, is the set {`none`, `red`, `blue`, `green`, `dashed`}. The set of lists, denoted by $\mathbb{L}$, consists of all integers and strings that can be derived from the following abstract syntax:

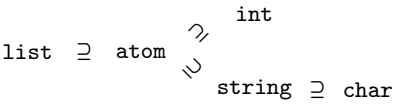
$\mathbb{L}$	::=	`empty` | GraphExp | $\mathbb{L}$ ‘:’ $\mathbb{L}$
GraphExp	::=	[‘-’] Digit {Digit} | GraphStr
GraphStr	::=	‘ “ ’ {Character} ’ ” ’ | GraphStr ‘.’ GraphStr

where Character is the set of all printable characters except ‘”’ (i.e. ASCII characters 32, 33, and 35-126), while Digit is the digit set $\{0, \ldots, 9\}$.

The *GP 2 label and mark alphabet* is defined as the tuple $\langle \mathbb{L}, \mathbb{L}, \mathbb{M}_V, \mathbb{M}_E \rangle$, which is denoted by $\mathcal{L}$. □

The colon operator ':' is used to concatenate list expressions while the dot operator '.' is used to concatenate strings. The empty list is signified by the keyword `empty`, where it is displayed as a blank label graphically.

Basically, in a host graph, a list consists of (list of) integers and strings which are typed according to hierarchical type system as below:



where the domain for `list`, `atom`, `int`, `string`, and `char` is $\mathbb{Z} \cup \text{Char}^*)^*, \mathbb{Z} \cup \text{Char}^*, \mathbb{Z}$, $\{\text{Char}\}^*$, and Char respectively.

Definition 2.16 (Labels of rules in GP 2)**.** Let $\mathbb{E}$ be the set of all expressions that can be derived from the syntactic class List in the following grammar:

$\mathbb{E}$	::=	List
List	::=	`empty` | Atom | List ':' List | ListVar
Atom	::=	Integer | String | AtomVar
Integer	::=	['-'] Digit {Digit} | '('Integer')' | IntVar
		| Integer ('+' | '-' | '*' | '/') Integer
		| (`indeg` | `outdeg`) '('NodeId')'
		| `length` '('AtomVar | StringVar | ListVar')'
String	::=	Char | String '.' String | StringVar
Char	::=	' " '{Character}' " ' | CharVar

where ListVar, AtomVar, IntVar, StringVar, and CharVar represent variables of type `list`, `atom`, `int`, `string`, and `char` respectively. Also, NodeId represents node identifiers.

Label alphabet for left and right-hand graphs of a GP 2 rule, denoted by $\mathcal{S}$, is defined as the tuple $\mathbb{E}, \mathbb{E}, (\mathbb{M}_V \cup \{\texttt{any}\}), (\mathbb{M}_E \cup \{\texttt{any}\})$. □

Definition 2.17 (GP 2 host graphs and rule graphs)**.** A graph is a *rule graph* if it is in $\mathcal{G}(\mathcal{S})$, and it is a *host graph* if it is in $\mathcal{G}(\mathcal{L})$. □

If we compare the grammars of Definition 2.15 and Definition 2.16, it is obvious that $\mathcal{L}$ is part of expressions that can be derived in the latter grammar. Hence, $\mathcal{L} \subset \mathcal{S}$, which means we can consider host graphs as special cases of rule graphs. From here, we may refer 'rule graphs' simply as 'graphs', which also means host graphs are included.

Syntactically, a graph in GP 2 is written based on the following syntax:

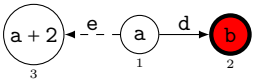
Graph	::=	[Position] '|' Nodes '|' Edges
Nodes	::=	'(' NodeId ['(R)'] ',' Label [',' Position] ')'
Edges	::=	'(' EdgeId ['(B)']',' NodeId ',' NodeId ',' Label ')'

where Position is a set of floating-point cartesian coordinates to store layout information for graphical editors, NodeId and EdgeId are sets of node and edge identifiers, and Label is set of labels as defined in Definition 2.15 and Definition 2.16. Also, (R) in Nodes is used for rooted nodes while (B) in Edges is used for bidirectional edges. Bidirectional edges may exist in rule graphs but not in host graphs. In GP 2, we only need to define graphs in $\mathcal{G}(\mathcal{L})$ (including $\mathcal{G}(\mathcal{S})$), such that labels are always defined and nodes without additional information'(B)' represent unrooted nodes.

Example 2.5 (A GP 2 graph)**.** Let G be a graph with $V_G = \{1, 2, 3\}, E_G = \{e1, e2\}, s_G = \{e1 \mapsto 1, e2 \mapsto 1\}, t_G = \{e1 \mapsto 2, e2 \mapsto 3\}, l_G^V = \{1 \mapsto a, 2 \mapsto b, 3 \mapsto a + 2\}, l_G^E = \{e1 \mapsto d, e2 \mapsto e\}$, $m_G^V = \{1 \mapsto \texttt{none}, 2 \mapsto \texttt{red}, 3 \mapsto \texttt{none}\}, m_G^E = \{e1 \mapsto \texttt{none}, e2 \mapsto \texttt{dashed}\}$, and $p_G = \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 0\}$.

The definition of G tells us that G has three nodes named 1, 2, and 3, it has two edges named $e1$ and $e2$, where $e1$ is an edge from node 1 to node 2, while $e2$ is an edge from node 1 to node 3. Respectively, node 1, 2, and 3 are labelled with a, b, and $a + 2$, where node 2 is red while the other two nodes has no colour. Edge $e1$ is labelled with d and has no mark, while $e2$ is a dashed edge labelled with e. Node 2 is a rooted node, while the others are unrooted.

Graphically, G can be seen as the following graph:



Syntactically in GP 2, G is written as follows:

$$\texttt{| (1, a) (2[R], b\#red) (3, a + 2) | (e1, 1, 2, d) (e2, 1, 3, e\#dashed)}$$

2.1.3.2 Conditional rule schemata

Like traditional rules in graph transformation that use double-pushout approach, rules in GP 2 (called rule schemata) consists of a left-hand graph, an interface graph, and a right-hand graph. GP 2 also allows a condition for the left-hand graph. When a condition exists, the rule is called a conditional rule schema.

Definition 2.18 (Rule schemata [11])**.** A *rule schema* $r = \langle L \leftarrow K \rightarrow R \rangle$ comprises totally labelled rule graphs L and R, a graph K containing only unlabelled nodes with undefined

rootedness, and inclusions $K \to L$ and $K \to R$. All list expressions in L are simple (i.e. no arithmetic operators, contains at most one occurrence of a list variable, and each occurrence of a string sub-expression contains at most one occurrence of a string variable). Moreover, all variables in R must also occur in L, and every node and edge in R whose mark is `any` has a preserved counterpart item in L.

An *unrestricted rule schema* is a rule schema without restriction on expressions and marks in its left and right-hand graph. □

Remark 2.19. Note that the left and right-hand graph of a rule schema can be rule graphs or host graphs since a host graph is a special case of rule graphs. In GP 2, we only consider rule schemata (with restrictions). In this thesis, we use unrestricted rule schemata to be able to express the properties of the inverse of a rule schema.

In GP 2, a condition can be added to a rule schema. This condition expresses properties that must be satisfied by a match of the rule schema. The conditions may express structural graph conditions (e.g. the existence of an edge between two nodes), and also attribute conditions (e.g. labels of a node in the graph). The variables occur in a rule schema condition must also occur in the left-hand graph of the rule schema.

Definition 2.20 (Conditional rule schemata [11])**.** A conditional rule schema is a pair $\langle r, \Gamma \rangle$ with r a rule schema and Γ a condition that can be derived from Condition in the grammar below:

Condition	::=	(`int` | `char` | `string` | `atom`) '('Var')'
		| List ('=' | '`!=`') List
		| Integer ('`>`' | '`>=`' | '`<`' | '`<=`') Integer
		| `edge` '(' NodeId ',' NodeId [',' List [Mark]] ')'
		| `not` Condition
		| Condition (`and` | `or`) Condition
		| '(' Condition ')'
Var	::=	ListVar | AtomVar | IntVar | StringVar | CharVar
Mark	::=	`red` | `green` | `blue` | `dashed` | `any`

such that all variables that occur in Γ also occur in the left-hand graph of r. □

The left-hand graph of a rule schema consists of a rule graph, while a morphism is a mapping function from a host graph. To obtain a host graph from a rule graph, we can assign constants for variables in the rule graph. For this, here we define assignment for labels.

In literature, such as [1], label assignment is defined as a mapping from label variables in a rule graph to a list. Here, we also consider the mark `any` as a mark variable, which can be mapped to a node/edge mark.

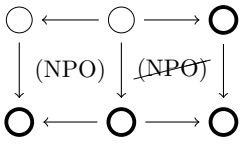


FIGURE 2.8: Non-natural double-pushout

Definition 2.21 (Label assignment)**.** Consider a rule graph L and the set X of all variables occurring in L. For each $x \in X$, let $\mathrm{dom}(x)$ denote the domain of x associated with the type of x. A *label assignment* for L is a triple $\alpha_L = \langle \alpha_{\mathbb{L}}, \mu_V, \mu_E \rangle$ where $\alpha_{\mathbb{L}} : X \to \mathbb{L}$ is a function such that for each $x \in X$, $\alpha_{\mathbb{L}}(x) \in \mathrm{dom}(x)$, and $\mu_V : V_L \to \mathbb{M}_V \backslash \{\texttt{none}\}$ and $\mu_E : E_L \to \mathbb{M}_E \backslash \{\texttt{none}\}$ are partial functions assigning a mark to each node and edge marked with `any`. □

For a conditional rule schema $\langle L \leftarrow K \rightarrow R,\ \Gamma \rangle$ with the set X of all list variables, set Y (or Z) of all nodes (or edges) whose mark is `any`, and label assignment α_L, we denote by L^{α} the graph L after the replacement of every $x \in X$ with $\alpha_{\mathbb{L}}(x)$, every $m_L^V(i)$ for $i \in Y$ with $\mu_{\mathbb{M}_V}(i)$, and every $m_L^E(i)$ for $i \in Z$ with $\mu_{\mathbb{M}_E}(i)$. Then for an injective graph morphism $g : L^{\alpha} \to G$ for some host graph G, we denote by $\Gamma^{g,\alpha}$ the condition that is obtained from Γ by substituting $\alpha_{\mathbb{L}}(x)$ for every variable x, $g(v)$ for every $v \in V_L$, and $g(e)$ for every $e \in E_L$.

The truth value of $\Gamma^{g,\alpha}$ is required for the application of a conditional rule schema. In addition, the application also depends on the dangling condition. Since a rule schema has an unlabelled graph as its interface, a natural pushout, i.e. a pushout that is also a pullback, is required in a rule schema application. This approach is introduced in [37] for unrooted graph programming.

Note that we require natural double-pushout in direct derivation. We use a natural pushout to have a unique pushout complement up to isomorphism in relabelling graph transformation [37, 38]. In [11], a graph morphism preserves rooted nodes while here we require a morphism to preserve unrooted nodes as well. We require the preservation of unrooted nodes to prevent a non-natural pushout as can be seen in Figure 2.8 [29]. In addition, we need a natural double-pushout because we want to have invertible direct derivations.

The natural double-pushout construction such that we have natural double-pushout is described in [11, 29], that are:

1. To obtain D, remove all nodes and edges in $g(L - K)$ from G. For all $v \in V_K$ with $l_K^V(v) = \perp$ and $m_K^V(v) = \perp$, define $l_D^V(g_V(v)) = \perp$ and $m_D^V(g_V(v)) = \perp$. Also, define $p_D(g_V(v)) = \perp$ for all $v \in V_K$ where $p_K(v) = \perp$.
2. Add all nodes and edges, with their labels and rootedness, from $R - K$ to D. For $e \in E_R - E_K$, $s_H(e) = s_R(e)$ if $s_R(E) \in V_R - V_K$, otherwise $s_H(e) = g_V(s_R(e))$. Targets are defined analogously.

3. For all $v \in V_K$ with $l_K^V(v) = \perp$ anf $m_K^V(v) = \perp$, define $l_H^V(g_V(v)) = l_R^V(v)$ and $m_H^V(g_V(v)) = m_R^V(v)$. Also, for the injective morphism $R \to H$ and $v \in V_K$ where $p_K(v) = \perp$, define $p_H(g_V^*(v)) = p_R(v)$. The resulting graph is H.

Direct derivations transform a host graph via a rule whose the left and right-hand graph are totally labelled host graphs. However, a conditional rule schema contains a condition, and its left or right-hand graph may not be a host graph. Hence, we need some additional requirements for the application of a conditional rule schema on a host graph.

Definition 2.22 (Conditional rule schema application)**.** Let us consider a conditional rule schema $r = \langle L \leftarrow K \rightarrow R,\ \Gamma \rangle$, and host graphs G, H. G directly derives r, denoted by $G \Rightarrow_{r,g} H$ (or $G \Rightarrow_r H$), if there exists a premorphism $g : L \to G$ and a label assignment α_L such that:

(i) $g : L^\alpha \to G$ is an injective morphism,
(ii) $\Gamma^{g,\alpha}$ is true,
(iii) $G \Rightarrow_{r^{g,\alpha},g} H$. □

A rule schema r (without condition) can be considered as a conditional rule schema $\langle r, \texttt{true} \rangle$, which means in its application, the point (ii) in the definition above is a valid statement for every unconditional rule schema r.

Syntactically, a conditional rule schema in GP 2 is written as follows:

RuleDecl	::=	RuleId '(' [VarList {':' VarList}] ';' ')'
		Graphs Interface [`where` Condition]
VarList	::=	Variable {',' Variable} ':' Type
Graphs	::=	'[' Graph ']' '=>' '[' Graph ']'
Interface	::=	`interface` '=' '{' [NodeId {',' NodeId}]'}'
Type	::=	`int` | `char` | `string` | `atom` | `list`

where Condition is the set of GP 2 rule conditions as defined in Definition 2.20 and Variable represents variables of all types. Graph represent rule graphs, where bidirectional edges may exist. Bidirectional edges and `any`-marks are allowed in the right-hand graph if there exist preserved counterpart item in the left-hand graph.

A rule schema with bidirectional edges can be considered as a set of rules with all possible direction of the edges. For example, a rule schema with one bidirectional edge between node u and v can be considered as two rule schemata, where one rule schema has an edge from u to v while the other has an edge from v to u.

2.1.3.3 Syntax and operational semantics of graph programs

A GP 2 graph program consists of a list of three declaration types: rule declaration, main procedure declaration, and other procedure declaration. A main declaration is where the program starts from so that there is only one main declaration allowed in the program, and it consists of a sequence of commands. For more details on GP 2 programs' abstract syntax, see Figure 2.9, where RuleId and ProcId are identifiers that start with lower case and upper case respectively.

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

FIGURE 2.9: Abstract syntax of GP 2 programs

When executed, rule schemata that exist in the program are applied to the input graph. If a rule schema can not be applied to the graph, it yields failure. A program can also execute some commands sequentially by using ';'. There also exist `if` and `try` as branching commands, where the program will execute command after `then` when the condition is satisfied or `else` if the condition is not satisfied. However, as we can see in the syntax of GP 2 in Figure 2.9, we have command sequence as the condition of branching commands instead of a Boolean expression. Here, we say that the condition is satisfied when the execution of the condition on the initial graph terminates with a result graph (that is, it neither diverges nor fails) and it is not satisfied if the execution yields failure.

The difference between `if` and `try` lies in the host graph that is used after the evaluation of conditions. For `if`, the program will use the host graph that is used before the examination of the condition. Otherwise for `try`, if the condition is satisfied, then the program will execute the graph obtained from applying the condition or the previous graph if the condition is not satisfied. Other than branching commands, there is also a loop command '!' (read as "as long as possible"). It executes the loop-body as long as the command does not yield failure. Like a loop in other programming languages, a !-construct can result in non-termination of a program.

Configurations in GP 2 represents a program state of program execution in any stage. Configurations are given by $(\texttt{ComSeq} \times \mathcal{G}(\mathbb{L})) \cup \mathcal{G}(\mathbb{L}) \cup (\texttt{fail}))$, where $\mathcal{G}(\mathbb{L})$ consists of all host graphs. This means that a configuration consists either of unfinished computations, represented by command sequence together with current graph; only a graph, which means all commands have been executed; or the special element `fail` that represents a failure state. A small step transition relation $\rightarrow$ on configuration is inductively defined by inference rules shown in Figure 2.10 and Figure 2.11 where $\mathcal{R}$ is a rule set call; C, P, P', and Q are command sequences; and G and H are host graphs.

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

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

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

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

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

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

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

FIGURE 2.10: Inference rules for core commands [2]

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

$$[\text{Skip}_1]\ \langle \texttt{skip}, G\rangle \rightarrow G \qquad [\text{Fail}]\ \langle \texttt{fail}, G\rangle \rightarrow \texttt{fail}$$

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

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

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

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

FIGURE 2.11: Inference rules for derived commands [2]

The semantics of programs is given by the semantic function $[\![\_]\!]$ that maps an input graph G to the set of all possible results of executing a program P on G. The application of $[\![P]\!]$ to G is written $[\![P]\!]G$. The result set may contain proper results in the form of graphs or the special values *fail* and $\perp$. The value `fail` indicates a failed program run while $\perp$ indicates a run that does not terminate or gets stuck. Program P can diverge from G if there is an

infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$. Also, P can get stuck from G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

Definition 2.23 (Semantic function [2])**.** The semantic function $[\![\_]\!]$: ComSeq $\to (\mathcal{G}(\mathbb{L}) \to 2^{\mathcal{G}(\mathbb{L})\cup\{fail,\perp\}})$ is defined by

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

A program C can get stuck only in two situations, that is either P contains a command `if` A `then` P `else` Q or `try` A `then` P `else` Q such that A can diverge from a host graph G, or P contains a loop $B!$ whose body B can diverge from a host graph G. The evaluation of such commands gets stuck because none of the inference rules for if-then-else, try-then-else or looping is applicable. Getting stuck always signals some form of divergence.

We sometimes need to prove that a property holds for all graph programs. For this, we use structural induction on graph programs by having a general version of graph programs. That is, ignoring the context condition of the command `break` such that it can appear outside a loop. However, when `break` occur outside the context condition, we treat it as a `skip`.

Definition 2.24 (Structural induction on graph programs)**.** Proving that a property *Prop* holds for all graph programs by induction, is done by:

Base case.

Show that *Prop* holds for $\mathcal{R} = \{r_1, \ldots, r_n\}$, where $n \geq 0$

Induction case.

Assuming *Prop* holds for graph programs C, P, and Q, show that *Prop* also holds for:

1. $P; Q$,
2. `if` C `then` P `else` Q,
3. `try` C `then` P `else` Q, and
4. $P!$. $\square$

The commands `fail` and `skip` can be considered (respectively) as a call of the ruleset $\mathcal{R} = \{\}$ and a call of the rule schema where the left and right-hand graphs are the empty graphs. Also, the command P `or` Q can be replaced with the program
`if` (`Delete`!; {`nothing`, `add`}; `zero`) `then` P `else` Q
, where `Delete` is a set of rule schemata that deletes nodes and edges, including loops. `nothing` is the rule schema where the left and right-hand graphs are the empty graphs, `add` is the rule schema where the left-hand graph is the empty graph and the right- hand graph is a single 0-labelled unmarked and unrooted node, and `zero` is a rule schema that matches with a 0-labelled unmarked and unrooted node.

As mentioned before, a graph program's execution may yield a proper graph, failure, or diverge/get stuck. The last outcome only may happen when a loop exists in the program. In

some cases, we may want to disregard the possibility of diverging or getting stuck such that we only consider loop-free graph programs. To show that a property holds for a loop-free program, we also introduce structural induction on loop-free programs.

Definition 2.25 (Structural induction on loop-free programs)**.** Proving that a property *Prop* holds for all loop-free programs by induction, is done by:

Base case.

Show that *Prop* holds for $\mathcal{R} = \{r_1, \ldots, r_n\}$, where $n \geq 0$

Induction case.

Assuming *Prop* holds for loop-free programs C, P, and Q, show that *Prop* also holds for:

1. P `or` Q,
2. $P; Q$,
3. `if` C `then` P `else` Q, and
4. `try` C `then` P `else` Q. □

2.2 Verification of graph programs

Poskitt and Plump have studied reasoning about graph programs since 2010 [13–16]. They use Hoare logic style verification to reason about the programs. They introduced E-condition and M-condition to express pre- and post-condition a graph in first-order logic and monadic second-order logic. While E-condition can express first-order properties of GP 2 graphs, M-condition can express monadic second-order properties of graphs without attribute (not GP 2 graphs). Apart from reasoning about graph programs, Hoare logic verification has also been used in verification of conventional languages such as Java [39–41].

2.2.1 Verification with Hoare logic

Hoare logic is one approach to the verification of the correctness of programs. Central of Hoare logic are Hoare triples $\{pre\}\ P\ \{post\}$ that explain the behaviour of program P in executing programs on a state satisfying a precondition *pre*. In this case, the result of the execution must satisfy the postcondition *post*. For expressing pre- and postconditions, we can use assertions languages such as arbitrary mathematical languages or English.

Axioms and inference rules are included in Hoare logic. Axioms serve as a starting point of the verification for further reasoning. For example, let consider the program `skip`, that is a program that terminates immediately without altering program states. The following axiom

would be an obvious axiom for the program:

$$\frac{}{\{pre\}\ \texttt{skip}\ \{pre\}}$$

It is obvious because when we execute `skip` to a state satisfying *pre*, the program terminates without doing anything to the state so that *pre* still holds after the execution.

Unlike axioms, inference rules can not stand on one triple alone. We need to proof other inference rules and axioms to derive a triple from inference rule. For example, we want to have inference rule about $P;Q$, that is a program that executes P first and then executes Q. We might need to use other inference rules, axioms, or complete proofs about P and Q that are individually relative to some midpoint as follows:

$$\frac{\{pre\}\ P\ \{mid\} \qquad \{mid\}\ Q\ \{post\}}{\{pre\}\ P;Q\ \{post\}}$$

Hoare triples are used with proof trees as defined in Definition 2.26. Basically, a proof tree consists of axioms and inference rules, collectively referred to as "proof rules".

Definition 2.26. Proof tree [1]] If $\{c\}\ P\ \{d\}$ is an instance of an axiom X then

$$X\ \frac{}{\{c\}\ P\ \{d\}}$$

is a proof tree. If $\{c\}\ P\ \{d\}$ can be instantiated from the conclusion of an inference rule X, and there are proof trees $T_1, \ldots, T_n$ with conclusions that are instances of the n premises of X, then

$$X\ \frac{T_1 \quad \ldots \quad T_n}{\{c\}\ P\ \{d\}}$$

is a proof tree.

To define what it means for Hoare triples $\{pre\}\ P\ \{post\}$ to be correct in reasoning about graph programs, Poskitt in his thesis [1] defines three notions of correctness, that are: partial correctness, weak total correctness, and total correctness.

Partial correctness is only used to consider the graphs that might result without considering that the program might diverge, get stuck, or fail so no graphs might occur as a result. So with this correctness, we can not guarantee the absence of divergence and executions that get stuck and the absence of failure. Weak total correctness then covers the absence of divergence and getting stuck, and total correctness completes it with the absence of failure.

Definition 2.27 (Partial correctness [1])**.** Let c and d be assertions, P be a graph program, and $\mathbb{L}$ be the set of lists (see Definition 2.17). P is *partially correct* with respect to precondition c and postcondition d if for every graph $G \in \mathcal{G}(\mathbb{L})$, G satisfies c implies H satisfies d for every H in $[\![P]\!]G$. □

Definition 2.28 (Weak total correctness [1])**.** Let c and d be assertions and P be a graph program. P is *weakly totally correct* with respect to a precondition c and postcondition d if P is partially correct with respect to precondition c and postcondition d and if for every graph $G \in \mathcal{G}(\mathbb{L})$ such that G satisfying c, there is no infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_1, G_1\rangle \to \ldots$ (divergence), and there is no terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$ (getting stuck). □

Definition 2.29 (total correctness [1])**.** Let c and d be assertions and P be a graph program. P is *totally correct* with respect to a precondition c and postcondition d if P is weakly totally correct with respect to precondition c and postcondition d and if for every graph $G \in \mathcal{G}(\mathbb{L})$ such that G satisfying c, there is no derivation $\langle P, G\rangle \to^*$ fail. □

2.2.2 Assertions for graph programs

In his thesis, Poskitt expresses pre- and postcondition using nested conditions with expression (E-Condition). It is used to reason about the structural properties of graphs. The following is the simplest form of E-condition:

$$\exists(C)$$

where C is a graph labelled with expressions. Graph G in $\mathcal{G}(\mathbb{L})$ satisfies such an E-condition when there exists an assignment α from variables in C such that there exists injective morphism from C^α to G. We can also use Boolean negation $\neg$ to express that no such morphism exists. So if we want to express "the graph is loop-free", we can use the following E-condition:

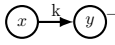
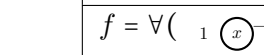
$\neg\exists($  $)$

Definition 2.30 (E-condition [14])**.** An E-condition c over a graph P is of the form true or $\exists(a|\gamma, c')$, where $a : P \to C$ is an injective graph morphism with $P, C \in \mathcal{G}(\text{Exp})$, γ is an assignment constraint, and c' is an E-condition over C. Boolean formulae over E-conditions over P yield E-conditions over P, that is, $\neg c$ and $c_1 \wedge c_2$ are E-conditions over P if c, c_1, c_2 are E-conditions over P. □

More examples of E-condition can be seen in Table 2.1. In the examples, when the domain of morphism $a : P \to C$ can unambiguously be inferred, only the codomain C is written. An

E-condition over a graph morphism whose domain is the empty graph is referred to as an E-constraint.

TABLE 2.1: E-condition examples

E-condition	is read
$c = \exists($ $)$	there exists at least one non-looping edge
$d = \exists($  $\text{type}(x,y)=\text{int} \wedge x < y)$	there exists at least one pair of adjacent integer-labelled nodes, of which the label of the target node is larger than that of the source node
$e = \neg\exists($  $)$	there does not exist a node that is attached to more than one loop
$f = \forall($  $\text{type}(x)=\text{int}, \neg\exists($ $))$	no integer-labelled node has an outgoing edge to another node (with any label)

2.2.3 Hoare calculus for graph programs

Proof rules for partial correctness used by Poskitt and Plump are written in Figure 2.12. Here, $Pre[r, c]$ is an operation to get the weakest liberal precondition relative to r and c, that is the weakest condition on a graph such that application of r to the graph will result in a graph that satisfies c. The use of liberal precondition instead of precondition is used for partial correctness because the existence of a result graph is not proven in the calculus. The transformation itself is done by:

1. forming disjunction of E-conditions over the right-hand graph of rule r accounting for the possible ways in which assertion c and comatches of r might overlap,
2. shift this e-condition over to the left-hand graph of r, and
3. nest the obtained e-condition to obtain an E-constraint such that it universally quantified over all possible matches of r [1].

In addition, we also have $App(\mathcal{R})$ which formalises the applicability of $\mathcal{R}$. It can take a set of conditional rule schemata $\mathcal{R}$ as input and transform it into an E-constraint expressing that at least one rule schema in the set is applicable. In other words, graph G satisfying $App(\mathcal{R})$ if there exists direct derivation $G \Rightarrow_{\mathcal{R}} H$ for graph H, and G satisfying $\neg App(\mathcal{R})$ if there is no such direct derivation (i.e. applying $\mathcal{R}$ to G will lead failure).

Example 2.6. $(Pre[r, c])$.
Let consider the following rule schema:

$$[\text{ruleapp}]_{wlp}\ \frac{}{\{Pre[r,c] \vee \neg App(\{r\})\}\ r\ \{c\}} \qquad [\text{ruleapp}]\ \frac{}{\{Pre[r,c]\}\ r\ \{c\}}$$

$$[\text{nonapp}]\ \frac{}{\{\neg App(\{r\})\}\ r\ \{\texttt{false}\}} \qquad [\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

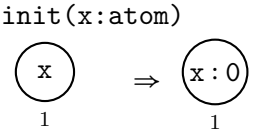
$$[\text{comp}]\frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P;Q\ \{d\}} \qquad [!]\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg App(\mathcal{R})\}}$$

$$[\text{cons}]\ \frac{c \text{ implies } c' \quad \{c'\}\ P\ \{d'\} \quad d' \text{ implies } d}{\{c\}\ P\ \{d\}}$$

$$[\text{if}]\ \frac{\{c \wedge App(\mathcal{R})\}\ P\ \{d\} \quad \{c \wedge \neg App(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge App(\mathcal{R})\}\ \mathcal{R};P\ \{d\} \quad \{c \wedge \neg App(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

FIGURE 2.12: Partial correctness rules with E-constraints for core commands[1]



and let c denote the E-constraint

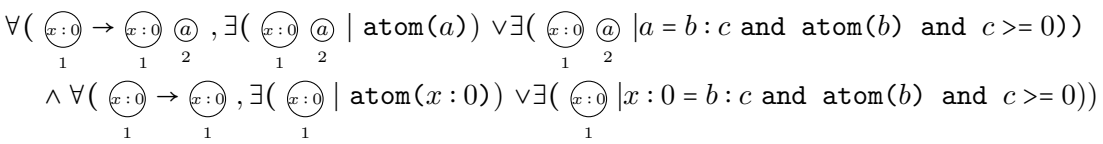
$$\forall(\underset{1}{@}, \exists(\underset{1}{@} \mid \texttt{atom}\ (a)) \vee \exists(\underset{1}{@} \mid a = b : c\ \texttt{and}\ \texttt{atom}(b)\ \texttt{and}\ c >= 0))$$

specifying that "every (unmarked) node is labelled by either an atom or a list comprising an atom followed by a natural number".

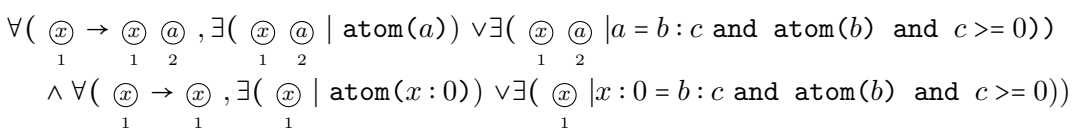
Recall the first step in transforming $Pre[r,c]$. From this step, we get condition that expresses possible ways in which assertion c and comatches of r might overlap, that is:

1. every node that is not in the image of the right-hand graph under the the morphism is either labelled by an atom or an atom followed by a natural number, and
2. the node in the image of the right-hand graph under the morphism is either labelled by an atom or an atom followed by a natural number.

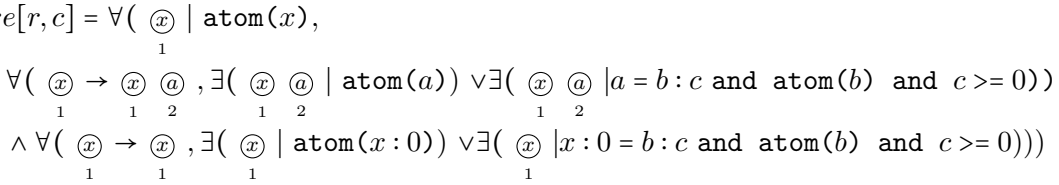
Or in E-condition:



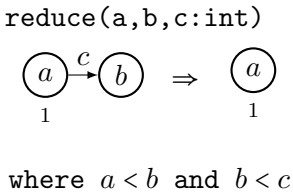
Then by shifting the conditions over to the left hand graph of the rule, we get:



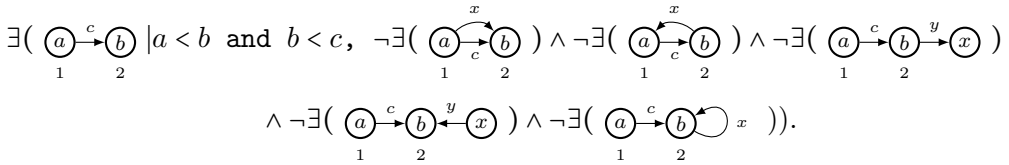
Finally, by doing the last step of the transformation, we get:



Example 2.7. ($App(\mathcal{R})$).
Let consider the following rule schema:



For the rule `reduce` above, the rule is applicable to a graph when there exists two adjacent nodes and the target node is not incident to any other nodes (otherwise, we will get a dangling condition because of the removal of the target node). Therefore $App(\{\texttt{reduce}\})$ is true if the constraint is satisfied. We can also express this in the following E-constraint:



2.3 Monadic second-order logic for graphs

Nested conditions [26] (also E-conditions as its extension), can also be translated to first-order logic [17]. In first-order logic, we may have quantification over a single element. Monadic

second-order formulas are often considered as an extension of first-order logic. It is a a fragment of second-order logic, where the second-order quantification is limited to quantification over sets.

In standard logic, a formula consists of vocabulary. A vocabulary σ is a collection of individual/constant symbols, variable symbols, relation/predicate symbols, function symbols, connectives, the quantifier symbols, and auxiliary symbols, where every relation and function symbol has an associated arity [42, 43]. When associated with a structure (e.g. graphs), formulas the three classes of individual, relation, and functional symbols should be one-one correspondence should be correspondence with the designated individuals, relations, and functions of the structure [43].

Let us consider a vocabulary σ, term and formulas of the monadic second-order predicate calculus over σ is defined inductively as follow [42]:

- Every (first-order and second-order) variable x is a term
- Every constant symbol c is a term
- If $t_1, \ldots, t_k$ are terms and f is a k-ary function symbol, $f(t_1, \ldots, t_k)$ is a term
- If t_1 and t_2 are terms, $t_1 = t_2$ is an (atomic) formula
- If $t_1, \ldots, t_k$ are terms and f is a k-ary relation symbol, $f(t_1, \ldots, t_k)$ is an (atomic) formula
- If ψ_1 and ψ_2 are formulas, $\psi_1 \wedge \psi_2, \psi_1 \vee \psi_2$, and $\neg\psi_1$ are formulas
- If ψ is a formula and x is a (first-order or second-order) variable, $\exists x(\psi)$ and $\forall x(\psi)$ are formulas

For formulas ϕ and ψ, the shorthand notation $\phi \Rightarrow \psi$ and $\phi \Leftrightarrow \psi$ are often used for $\neg\phi \vee \psi$ and $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$ respectively. A variable can be free or bounded. A variable is bounded if there is a quantifier bind the variable. Otherwise, it is a free variable.

Definition 2.31 (Free variables [44])**.** Let each t, t_1, and t_2 be a term (i.e. a variable, constant, or function). The set $Free(t)$ of free variables of t is defined by recursion as follows:

1. $Free(t) = \{t\}$, if t is a variable;
2. $Free(t) = \{\}$, if t is a constant;
3. $Free(t) = Free(t_1, \ldots, t_n)$, if t is a function with argument $t_1, \ldots, t_n$;

4. $Free(t) = Free(t_1) \cup Free(t_2)$, if t is in the form $t_1 \oplus t_2$ for $\oplus \in \{+, -, *, /, .,:\}$

Then, let c, c_1, and c_2 be formulas. The set $Free(c)$ of free variables of c is defined by recursion as follows:

1. $Free(c) = Free(t_1) \cup \ldots \cup Free(t_n)$, if c is a predicate with arguments $t_1, \ldots, t_n$;
2. $Free(c) = Free(c_1) \cup Free(c_2)$, if c is in the form $c_1 \ominus c_2$ for a connective symbol $\ominus$;
3. $Free(\neg c) = Free(c)$;
4. $Free(c) = Free(c_1) - \{x\}$, for a variable x, if c is in the form $\exists x(c_1)$ or $\forall x(c_1)$;

A formula c is *closed* if $Free(c) = \varnothing$. A closed formula is also called a *sentence*. A formula without quantifiers is called *open*. □

Definition 2.32 (Bound Variables [44])**.** Let c, c_1, and c_2 be formulas. The set $Bound(c)$ of bound variables in c is given by:

1. $Bound(c) = \varnothing$, if c is a predicate;
2. $Bound(c) = Bound(c_1) \cup Bound(c_2)$, if c is in the form $c_1 \otimes c_2$ for a connective $\otimes$;
3. $Bound(\neg c) = Bound(c)$;
4. $Bound(c) = Bound(c_1) \cup \{x\}$, for a variable x, if c is in the form $\exists x(c_1)$ or $\forall x(c_1)$;

The sets $Free(c)$ and $Bound(c)$ of a formula c are not always be disjoint. They may also have the same variable as an element of the sets. See the following examples for more intuition about free and bound variable.

Next, we define the substitution of a term for a free variable in a term or a formula. The definition is similar to [44]

Definition 2.33 (Substitutions [44])**.** Let x be a variable and s, s_1, s_2 and t be terms where t is in domain of x based on Table 3.1. The result of substituting t in s for a variable x, denoted by $s^{[x \mapsto t]}$ is defined recursively as follows:

1. $s^{[x \mapsto t]} = if\ y \neq x\ then\ y\ else\ t$, when s is a variable y;
2. $s^{[x \mapsto t]} = c$, when s is a constant c;
3. $s^{[x \mapsto t]} = f(t)$, when s is a function f with argument x;

4. $s^{[x\mapsto t]} = s_1^{[x\mapsto t]} \oplus s_2^{[x\mapsto t]}$, when s is in the form $s_1 \oplus s_2$ for $\oplus \in \{+, -, *, /, .., :\}$.

Then, let c, c_1, and c_2 be formulas. The set $c^{[x\mapsto t]}$ is defined recursively as follows:

1. $c^{[x\mapsto t]} = c$, if c is true or false;
2. $c^{[x\mapsto t]} = p(s_1^{[x\mapsto t]}, \ldots, s_n^{[x\mapsto t]})$, if c is a predicate p with argument $s_1, \ldots, s_n$;
3. $c^{[x\mapsto t]} = c_1^{[x\mapsto t]} \otimes c_2^{[x\mapsto t]}$, if c is in the form $c_1 \otimes c_2$ for a connective $\otimes$;
4. $(\neg c)^{[x\mapsto t]} = \neg c^{[x\mapsto t]}$;
5. $c^{[x\mapsto t]} = if\ x \neq y\ then\ \exists y(c_1^{[x\mapsto t]})\ else\ \exists y(c_1)$, for a variable y, if c is in the form $\exists y(c_1)$;
6. $c^{[x\mapsto t]} = if\ x \neq y\ then\ \forall y(c_1^{[x\mapsto t]})\ else\ \forall y(c_1)$, for a variable y, if c is in the form $\forall y(c_1)$;

□

Courcelle and Engelfriet in [45] define some classes of monadic second-order formulas. Some of them are MS_1, MS_2, C_2MS. MS_1 is the class of monadic second-order formulas with quantification over nodes, while MS_2 has quantification over edges. On the other hand, C_2MS has an additional cardinality predicate, known as counting monadic second-order formulas.

Both first-order logic and monadic second-order logic are known to be undecidable [42, 45] in general. However, there are studies to answer the satisfiability problem of monadic second-order logic in some classes [45–47]. Courcelle shows that if the property of interest can be expressed in MS_2 logic, then parameterising by the combination of the treewidth of G and the size of the formula ϕ, it can be determined in linear time whether the graph has the property [45, 48]. Moreover, Seese in [46] shows the converse, that if a set of finite, simple, undirected graphs has a decidable MS_2 then it has bounded tree-width. Also, Courcelle in [47] shows that if the class has a decidable C_2MS-satisfiability problem, it has bounded clique-width.

2.4 Summary

In this chapter, we have discussed some literature that becomes a basic theory of this thesis. We presented the theory about graph transformation systems and introduce the graph programming language GP 2. Verification of GP 2 graph programs has been studied by Poskitt and Plump since 2010 [14–16, 49]. They introduce E-conditions, which is an extension of nested conditions [17, 26], to express first-order properties of GP 2 graphs and use it in graph program verification with Hoare calculus. They also define construction to obtain weakest liberal precondition from a given postcondition and rule schema, which can be used to provide an axiom in Hoare-style verification.

However, there are some drawback from approach:

- E-conditions are limited to express first-order properties of GP 2 graphs; they can not express the existence of a path or connectedness
- The construction of weakest liberal precondition does not consider rootedness, wild-cards (the `any`-marks), and the command `break`
- The proof calculi are limited to graphs programs whose every loop body and the condition of every branching command is a rule-set call; so that it can not handle nested loops

As a first step to tackle the first problem, Poskitt and Plump [16] introduce M-conditions to express monadic second-order properties of graphs without attributes. However, since GP 2 graphs are attributed, the problem of expressing monadic second-order properties of GP 2 graphs is still an open problem in the literature.

This chapter also discusses the monadic second-order logic. Here, we choose to use standard logic because it may be useful for programmers who are not familiar with morphisms. In addition, the vast literature on standard logic may give us an advantage in the future because we may be able adopt the theories to GP 2 environment.

Chapter 3

Monadic second-order logic for graph programs

This chapter defines monadic second-order formulas that are used as assertions in verification about graph programs later in Chapter 7. The formulas can be used to express properties about graphs that are used in GP 2. We can show that a graph has properties expressed by a formula by showing satisfaction of the formula in the graph.

3.1 Monadic second-order formulas

In this section, we define monadic second-order (MSO) formulas by presenting a syntax. Our MSO formulas are in the form of standard logic, unlike in [1, 16, 26] where assertions for graph program verification are in the form of nested conditions. Similar to other languages in standard logic, our monadic second-order language has logical connectives, variables, constants, auxiliaries, predicates, and functions. However, here we consider GP 2 attributes and syntax when we define the language so that we can express every property of a GP 2 graph that are essential in a rule application.

Here, we use logical connectives that are commonly used in standard logic, that are $\wedge$ (and), $\vee$ (or), and $\neg$ (not), with logical constants true and false, equality symbols $=, \neq, >, \geq, <, \leq, \subset, \subseteq$, and quantifiers $\exists_{\mathsf{v}}, \exists_{\mathsf{e}}, \exists_{\mathsf{l}}, \exists_{\mathsf{V}}, \exists_{\mathsf{E}}$, which are reserved for node, edge, label, node set, and edge set variables respectively. We also use some constants, which are label constants (all elements in $\mathbb{L}$ and label `empty`), also mark constants $\mathsf{none}, \mathsf{red}, \mathsf{green}, \mathsf{blue}, \mathsf{green}, \mathsf{dashed}, \mathsf{grey}$, and any. Variables are typed as defined in Definition 3.1. For predicates, we use $\mathsf{int}, \mathsf{char}, \mathsf{string}, \mathsf{atom}$, $\mathsf{edge}, \mathsf{path}, \mathsf{root}$ as described in Definition 3.4. We also have functions s (source), t (target), $\mathsf{l_v}$ (node label), $\mathsf{l_e}$ (edge label), $\mathsf{m_v}$ (node mark), $\mathsf{m_e}$ (edge mark), indeg, outdeg, length, card, integer operators $+, -, *, /$, label operator : (list concatenation), and string operator . (concatenation) as described in Definition 3.2.

Definition 3.1 (Variables)**.** A *variable* is a symbol used to represent an arbitrary element of a set. The set X is called the *domain* of a variable x, denoted by $\mathrm{dom}(x)$, if x representing an element of X. *Monadic second-order variables* are divided into nine categories, based on their domains, as can be seen in Table 3.1. The node, edge, node-set, edge-set, and list variables are pairwise distinct, while list, atom, integer, string, and char variables follow the hierarchy based on their domain, as in GP 2 labels.

Variables in categories SetNodeVar and SetEdgeVar are denoted by a single uppercase letter that may be followed by an integer as a subscript. On the other hand, we use a single lowercase letter, which may also be followed by a subscript, to denote variables in other categories. □

TABLE 3.1: Categories of variables and their domain on a graph G

kind of variables	**domain**
NodeVar	V_G
SetNodeVar	2^{V_G}
EdgeVar	E_G
SetEdgeVar	2^{E_G}
ListVar	$(\mathbb{Z} \cup (\mathrm{Char})^*)^*$
AtomVar	$\mathbb{Z} \cup \mathrm{Char}^*$
IntVar	$\mathbb{Z}$
StringVar	Char^*
CharVar	Char

We differentiate variables in nine categories for our formulas, which are first-order variables for nodes, edges, and labels (where labels are typed as in GP 2 labels), and second-order variables (set variables) for nodes and edges. We do not have set variables for labels since we have not find its significant effect in expressing graph properties. Moreover, graphs in GP 2 have a finite set of nodes and edges, but the set of labels is infinite.

A node, edge, or label of a graph can be represented by a variable or a function. In our formulas, functions we use are functions that are used in GP 2 rule labels, rule schema conditions, and the definition of graphs. In addition, we use a cardinality function to be able to express more properties of graphs. In [50], counting monadic second-order formulas was introduced. The formulas have a predicate to express the cardinality (i.e. the number of elements) of a set in modulo. Here, we use more general expression to express the cardinality of a set. That is, by using the function $\mathsf{card(X)}$ for a node or edge-set variable X. The function returns the number of elements in the set represented by X.

Definition 3.2 (Functions)**.** A *function* in monadic second-order formula may represent a node, a label, or a list. The following are functions used in the formulas:

1. Source and target functions $\mathsf{s(x)}$ and $\mathsf{t(x)}$ representing nodes, where x is an edge variable

2. Node and edge label functions $\mathsf{l_v(y)}$ and $\mathsf{l_e(x)}$ representing lists, where x is an edge variable and y is a node variable or a source/target function
3. Node and edge mark functions $\mathsf{m_v(y)}$ and $\mathsf{m_e(x)}$ representing marks, where x is an edge variable and y is a node variable or a source/target function
4. Degree functions $\mathsf{indeg(y)}$ and $\mathsf{outdeg(y)}$ representing integers, where y is a node variable or a source/target function
5. Length function $\mathsf{length(z)}$ representing integer, where z is a list variable (or an atom, integer, string, or char variable)
6. Cardinality function $\mathsf{card(X)}$ representing integer, where X is a node-set or edge-set variable
7. Integer operators $\mathsf{z_1 + z_2}$, $\mathsf{z_1 - z_2}$, $\mathsf{z_1 * z_2}$, and $\mathsf{z_1/z_2}$ representing integers, where each $\mathsf{z_1}$ and $\mathsf{z_2}$ can be an integer variable or a function representing an integer
8. String concatenation operator $\mathsf{z_1 + z_2}$ representing string, where each $\mathsf{z_1}$ and $\mathsf{z_2}$ can be a string variable or a function representing a string
9. List concatenation operator $\mathsf{z_1 : z_2}$ representing list, where each $\mathsf{z_1}$ and $\mathsf{z_2}$ can be a list variables or a function representing a list □

In standard logic, as described in Chapter 2, other than the formula true and false, predicates are usually used for atom formulas. Here, we use predicates that are used in conditional rule schemata. In addition, we use a rootedness predicate to express rootedness of nodes, path predicate to express the existence of a directed path (see Definition 3.3), and an element predicate to express the connection between a variable or function representing a node (or edge) with a node (or edge) set variable.

Definition 3.3 (Directed path [51])**.** A *directed path* (with length $n \geq 0$ from node a to node b in a directed graph G is a sequence of edges $e_1, \ldots, e_n$ in G, where $s_G(e_1) = a$, $s_G(e_n) = b$, and for every $i = 1, \ldots, n-1$, $t_G(e_i) = s_G(e_{i+1})$. The empty set defines a path from a to a, and a path of length $n \geq 1$ from a to a is called a *cycle*. □

Definition 3.4 (Predicates)**.** A *predicate* can represent a Boolean value (true or false). The predicates used in monadic second-order formulas are:

1. Character predicate $\mathsf{char(x)}$, where x is a list variable
2. String predicate $\mathsf{string(x)}$, where x is a list variable
3. Integer predicate $\mathsf{int(x)}$, where x is a list variable

4. Atom predicate $\mathsf{atom}(\mathsf{x})$, where x is a list variable
5. Rootedness predicate $\mathsf{root}(\mathsf{y})$, where y is a node variable or a source/target function
6. Edge predicate $\mathsf{edge}(\mathsf{y}_1, \mathsf{y}_2), \mathsf{edge}((\mathsf{y}_1, \mathsf{y}_2, \mathsf{l}), \mathsf{edge}((\mathsf{y}_1, \mathsf{y}_2, \mathsf{m}),$ or $\mathsf{edge}((\mathsf{y}_1, \mathsf{y}_2, \mathsf{l}, \mathsf{m}),$ where y_1 and y_2 are node variables or source/target functions, l is a list constant, list variable, or function representing a list, while m is a mark constant
7. Path predicate $\mathsf{path}(\mathsf{y}_1, \mathsf{y}_2)$ or $\mathsf{path}(\mathsf{y}_1, \mathsf{y}_2, \mathsf{Z})$, where y_1 and y_2 are node variables or source/target functions, and Z is an edge-set variable
8. Element predicates $\mathsf{y} \in \mathsf{Y}$ and $\mathsf{z} \in \mathsf{Z}$, where y is a node variable or a source/target function, z is an edge variable, Y is a node-set variable, and Z is an edge-set variable. □

From the logical connectives, variables, constants, auxiliaries, predicates, and functions we have, we then define monadic second-order formulas as an abstract syntax as can be seen in Figure 3.1. In the syntax, Digit and Character are defined as in Definition 2.15: Character is the set of all printable characters except '"' (i.e. ASCII characters 32, 33, and 35-126), and Digit is the digit set $\{0, \ldots, 9\}$.

For brevity, we write $c \Rightarrow d$ for $\neg c \vee d$, $c \Leftrightarrow d$ for $(c \Rightarrow d) \wedge (d \Rightarrow c)$, $\forall_{\mathsf{V}}\mathsf{x}(c)$ for $\neg\exists_{\mathsf{v}}\mathsf{x}(\neg c)$, and similarly with $\forall_{\mathsf{e}}\mathsf{x}(c), \forall_{\mathsf{l}}\mathsf{x}(c), \forall_{\mathsf{V}}\mathsf{X}(c)$, and $\forall_{\mathsf{Y}}\mathsf{x}(c)$. We also sometimes write $\exists_{\mathsf{v}} x_1, \ldots, x_n(c)$ for $\exists_{\mathsf{v}} x_1(\exists_{\mathsf{v}} x_2(\ldots \exists_{\mathsf{v}} x_n(c) \ldots))$ (also for other quantifiers). Also, we define 'term' as the set of variables, constants, and functions in MSO formulas.

Definition 3.5 (Terms)**.** A *term* is a component of a monadic second-order formula that represents a node, an edge, a mark, or a list. In monadic second-order formulas, terms are defined as below:

1. every variable is a term representing an element of its domain
2. every constant is a term representing itself
3. source and target functions are terms representing nodes
4. label functions and list concatenation operator are terms representing lists
5. mark functions are terms representing marks
6. degree functions, length function, cardinality functions, and integer operators are terms representing integers
7. String concatenation operator is a term representing a string □

Formula	::=	true | false | Elem | Cond | Equal
		| Formula (‘∧’ | ‘∨’) Formula | ‘¬’Formula | ‘(’Formula‘)’
		| ‘$\exists_{\mathsf{v}}$’ NodeVar ‘(’Formula‘)’
		| ‘$\exists_{\mathsf{e}}$’ EdgeVar ‘(’Formula‘)’
		| ‘$\exists_{\mathsf{l}}$’ (ListVar) ‘(’Formula‘)’
		| ‘$\exists_{\mathsf{V}}$’ SetNodeVar ‘(’Formula‘)’
		| ‘$\exists_{\mathsf{E}}$’ SetEdgeVar ‘(’Formula‘)’
Number	::=	Digit {Digit}
Elem	::=	Node (‘∈’ | ‘∉’) SetNodeVar | EdgeVar (‘∈’ | ‘∉’) SetEdgeVar
Cond	::=	(int | char | string | atom) ‘(’Var‘)’
		| Lst (‘=’ | ‘≠’) Lst
		| Int (‘>’ | ‘>=’ | ‘<’ | ‘<=’) Int
		| edge ‘(’ Node ‘,’ Node [‘,’ Label] [‘,’ EMark] ‘)’
		| path ‘(’ Node ‘,’ Node [‘,’ SetEdgeVar] ‘)’
		| root ‘(’ Node ‘)’
Var	::=	ListVar | AtomVar | IntVar | StringVar | CharVar
Lst	::=	empty | Atm | Lst ‘:’ Lst | ListVar | $\mathsf{l_v}$ ‘(’Node‘)’ | $\mathsf{l_e}$ ‘(’EdgeVar‘)’
Atm	::=	Int | String | AtomVar
Int	::=	[‘-’] Number | ‘(’Int‘)’ | IntVar
		| Int (‘+’ | ‘-’ | ‘*’ | ‘/’) Int
		| (indeg | outdeg) ‘(’Node‘)’
		| length ‘(’AtomVar | StringVar | ListVar‘)’
		| card‘(’(SetNodeVar | SetEdgeVar)‘)’
String	::=	‘ “ ’ Character ‘ ” ’ | CharVar | StringVar | String ‘.’ String
Node	::=	NodeVar | (s | t) ‘(’ EdgeVar‘)’
EMark	::=	none | red | green | blue | dashed | any | $\mathsf{m_e}$‘($'$EdgeVar‘)$'$
VMark	::=	none | red | blue | green | grey | any | $\mathsf{m_v}$‘($'$Node‘)
Equal	::=	Node (’=’ | ‘≠’) Node | EdgeVar (’=’ | ‘≠’) EdgeVar
		| Lst (’=’ | ‘≠’) Lst | VMark (’=’ | ‘≠’) VMark
		| EMark (’=’ | ‘≠’) EMark

FIGURE 3.1: Abstract syntax of monadic second-order formulas

Remark 3.6. Due to the hierarchical system, a variable x representing a list may represent a character, string, integer, or atom as well. For practical reason, we consider x as a list variable unless the type char, string, int, or atom is stated in the formula. For example, x in the formula $\exists_{\mathsf{l}}\mathsf{x}(\mathsf{m_v}(\mathsf{y}) = \mathsf{x})$ is a list variable, while x in $\exists_{\mathsf{l}}\mathsf{x}(\mathsf{m_v}(\mathsf{y}) = \mathsf{x} \wedge \mathsf{int}(\mathsf{x}))$ is an integer variable.

In [50], a cardinality predicate $p(m, n, X)$ with $n > m \geq 0$ and $n \geq 2$ expresses that the number of elements in X is m in modulo n, e.g. $p(0, 2, X)$ expressing the number of elements in a set X is even. In our setting, this predicate can be expressed by $\mathsf{card(X) = k * n + m}$ for some fresh variable k or $\mathsf{(card(X) - m)/n \neq (card(X) - m - 1)/n}$. As an example, we can express $p(0, 2, X)$ by $\mathsf{card(X) = 2 * k}$ or $\mathsf{card(X)/n \neq (card(X) - 1)/n}$.

In some cases, we may use first-order formulas instead of monadic second-order formulas. To

have a first-order formula, we only need to omit second-order variables, also logical connectives, functions and predicates that are connected with second-order variables, i.e. symbols $\in$, $\subset, \subseteq$, function card, and predicate path.

Definition 3.7 (First-order formulas)**.** A first-order formula is an MSO formula without any second-order variable, and does not contain symbols $\in, \subset, \subseteq$, also predicate path and function card. □

Example 3.1 (Monadic second-order formulas)**.**

1. $\forall_{\mathsf{V}} \mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none})$ is a first-order formula expressing "all nodes are unmarked".
2. $\exists_{\mathsf{V}} \mathsf{X}(\forall_{\mathsf{V}} \mathsf{x}(\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$
   is a monadic second-order formula expressing "there exists at least two unmarked nodes". Alternatively, we can express it by the first-order formula
   $\exists_{\mathsf{V}} \mathsf{x}, \mathsf{y}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{V}}(\mathsf{y}) = \mathsf{none} \wedge \mathsf{x} \neq \mathsf{y})$.
3. $\exists_{\mathsf{V}} \mathsf{X}(\forall_{\mathsf{V}} \mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \exists_{\mathsf{I}} \mathsf{n}(\mathsf{card}(\mathsf{X}) = 2 * \mathsf{n}))$
   is a monadic second-order formula expresses "The number of grey nodes is even".

If we check the grammar of our formula, we have MSO over the graphs (i.e. we have set quantifiers for nodes and edges), but not for the labels. However, note that GP 2 graphs have attributes. This allows us to express MSO properties of the attributes indirectly. For example, we can consider an edge-less graph G representing a finite multi-set of lists (if we have two nodes with the same label) or a finite set of lists (if we do not have two nodes with the same label) such that every node in G representing a list. By this representation, we can also express MSO properties of graph attributes.

3.2 Satisfaction of a monadic second-order formula

The satisfaction of a monadic second-order formula c in a graph G relies on assignments. An assignment of c on G is defined in Definition 3.8. Informally, an assignment is a function that maps free variables to their domain.

Definition 3.8 (Assignments)**.** Let c be a monadic second-order formula, A, B, C, D, and E be the set of free node, edge, list, node-set, and edge-set variables in c (respectively). For a free variable x, $\mathrm{dom}(\mathsf{x})$ denotes the domain of variable's kind associated with x as in Table 3.1. A *formula assignment* of c on a host graph G is a tuple $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ of functions $\alpha_G = \langle \alpha_V : A \to V_G, \alpha_E : B \to E_G, \alpha_{2V} : D \to 2^{V_G}, \alpha_{2E} : E \to 2^{E_G} \rangle$, and $\alpha_{\mathbb{L}} = C \to \mathbb{L}$ such that for each free variable x, $\alpha(\mathsf{x}) \in \mathrm{dom}(\mathsf{x})$. We then denote by c^α the MSO formula c after the replacement of each term y to y^α where y^α is defined inductively:

1. If y is a free variable, $y^\alpha = \alpha(y)$;
2. If y is a constant, $y^\alpha = y$;
3. If $y = \mathsf{length}(\mathsf{x})$ for some list variable x, y^α equals to the number of characters in x^α if x is a string variable, 1 if x is an integer variable, or the number of atoms in x^α if x is a list variable;
4. If $y = \mathsf{card}(\mathsf{X})$ for some node-set or edge-set variable X, y^α is the number of elements in X^α;
5. If y is the functions $\mathsf{s}(\mathsf{x}), \mathsf{t}(\mathsf{x}), \mathsf{l_E}(\mathsf{x}), \mathsf{m_E}(\mathsf{x}), \mathsf{l_V}(\mathsf{x}), \mathsf{m_V}(\mathsf{x}), \mathsf{indeg}(\mathsf{x})$, or $\mathsf{outdeg}(\mathsf{x})$, y^α is $s_G(\mathsf{x}^\alpha)$, $t_G(\mathsf{x}^\alpha)$, $l^E_G(\mathsf{x}^\alpha), m^E_G(\mathsf{x}^\alpha), l^V_G(\mathsf{x}^\alpha), m^V_G(\mathsf{x}^\alpha)$, indegree of x^α in G , or outdegree of x^α in G, respectively;
6. If $y = \mathsf{x}_1 \oplus \mathsf{x}_2$ for $\oplus \in \{+, -, *, /\}$ and integers ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, $y^\alpha = \mathsf{x}_1 \oplus_\mathbb{Z} \mathsf{x}_2$;
7. If $y = \mathsf{x}_1.\mathsf{x}_2$ for some terms ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, y^α is string concatenation x_1 and x_2;
8. If $y = \mathsf{x}_1 : \mathsf{x}_2$ for some lists ${\mathsf{x}_1}^\alpha, {\mathsf{x}_2}^\alpha$, y^α is list concatenation x_1 and x_2 □

The satisfaction of a formula c in a graph G can be valuated by checking the existence of an assignment α for c on G such that c^α is true in G. A formula's satisfaction on a graph is defined in Definition 3.9.

Definition 3.9 (Satisfaction)**.** Let G be a graph and c be a monadic second-order formula. G satisfies c, written $G \vDash c$, if there exists an assignment α such that c^α is true in G (denotes by $G \vDash^\alpha c$). The condition where c^α is true is inductively defined:

1. If $c^\alpha = \mathsf{true}$ (or $c^\alpha = \mathsf{false}$), then c^α is true (or false);
2. If $c^\alpha = \mathsf{int}(\mathsf{x}), \mathsf{char}(\mathsf{x}), \mathsf{string}(\mathsf{x}), \mathsf{atom}(\mathsf{x})$, or $\mathsf{root}(\mathsf{x})$, c^α is true ifff $x^\alpha \in \mathbb{Z}, x^\alpha \in \mathrm{Char}, x^\alpha \in \mathrm{Char}^*, x^\alpha \in \mathbb{Z} \cup \mathrm{Char}^*$, or $p_G(x^\alpha) = 1$ respectively.
3. If c^α is in the form $\mathsf{edge}(\mathsf{x}_1, \mathsf{x}_2)$ for some $x_1, x_2 \in V_G$, then c^α is true iff there exists an edge $e \in E_G$ where $s_G(e) = x_1$ and $t_G(e) = x_2$. If there is an additional argument l (or m) for some $l \in \mathbb{L}$ (or $m \in \mathbb{M}$), then in addition to the existence of e with such source and target, the label (or mark) of e is equal to l (or m).
4. If c^α is in the form $\mathsf{path}(\mathsf{x}_1, \mathsf{x}_2)$ for some $x_1, x_2 \in V_G$, then c^α is true iff there exists a directed path from x_1 to x_2 (see Definition 3.3). If there is an extra argument E for $E \in 2^{E_G}$, then there is no $e \in E$ in the edge sequence of the path.
5. If c^α has the form $t_1 \otimes t_2$ where $\otimes \in \{>, >=, <, <=\}$ and $t_1, t_2 \in \mathbb{Z}$, b^α is true if and only if $t_1 \otimes_\mathbb{Z} t_2$ where $\otimes_\mathbb{Z}$ is the integer relation on $\mathbb{Z}$ represented by $\otimes$

6. If c^α has the form $t_1 \ominus t_2$ where $\ominus \in \{=, \neq\}$ and $t_1, t_2 \in V_G$, $t_1, t_2 \in E_G$, or $t_1, t_2 \in \cup\mathbb{L}\cup\mathbb{M}\{\mathsf{any}\}$, c^α is true if and only if $t_1 \ominus_{\mathbb{B}} t_2$ where $\ominus_{\mathbb{B}}$ is the Boolean relation represented by $\ominus$. Then for $t_1 = \mathsf{any}$, c^α is true if and only if $\mathtt{blue} \ominus_{\mathbb{B}} t_2 \vee \mathtt{red} \ominus_{\mathbb{B}} t_2 \vee \mathtt{green} \ominus_{\mathbb{B}} t_2 \vee \mathtt{grey} \ominus_{\mathbb{B}} t_2 \vee \mathtt{dashed} \ominus_{\mathbb{B}} t_2$ is true (and analogously for $t_2 = \mathsf{any}$).

7. If c^α has the form $t_1 \ominus t_2$ where $\ominus \in \{=, \neq, \subset, \subseteq\}$ and $t_1, t_2 \in 2^{V_G}$ or $t_1, t_2 \in 2^{E_G}$, c^α is true if and only if $t_1 \ominus_{\mathbb{B}} t_2$ where $\ominus_{\mathbb{B}}$ is the Boolean relation represented by $\ominus$. Also, if c^α has the form $t_1 \in t_2$ where $t_1 \in V_G$ and $t_2 \in 2^{V_G}$ or $t_1 \in E_G$ and $t_2 \in 2^{E_G}$, c^α is true if and only if t_1 is an element of t_2;

8. If c^α has the form $b_1 \oslash b_2$ where $\oslash \in \{\vee, \wedge\}$ and b_1, b_2 are Boolean expressions, c^α is true if and only if $b_1 \oslash_{\mathbb{B}} b_2$ where $\oslash_{\mathbb{B}}$ is the Boolean operation on $\mathbb{B}$ represented by $\oslash$.

9. If the form of c^α is $\neg b_1$ where b_1 is a Boolean expression, c^α is true if and only if b_1 is false.

10. If c^α has the form $\exists_{\mathsf{v}}\mathsf{x}(\mathsf{b})$ where x is a first-order node variable and b is a Boolean expression, c^α is true if and only if there exists $v \in V_G$ such that $b^{[x \mapsto v]}$ is true.

11. If c^α has the form $\exists_{\mathsf{e}}\mathsf{x}(\mathsf{b})$ where x is a first-order edge variable and b is a Boolean expression, c^α is true if and only if there exists $e \in E_G$ such that $b^{[x \mapsto e]}$ is true.

12. If c^α is in the form $\exists_{\mathsf{l}}\mathsf{l}(\mathsf{b})$ where x is a first-order list variable and b is a Boolean expression, c^α is true if and only if there exists $l \in \mathbb{L}$ such that $b^{[x \mapsto l]}$ is true.

13. If c^α has the form $\exists_{\mathsf{V}}\mathsf{X}(\mathsf{b})$ where x is a node set variable and b is a Boolean expression, c^α is true if and only if there exists $V \in 2^{V_G}$ such that $b^{[X \mapsto V]}$ is true.

14. If c^α has the form $\exists_{\mathsf{E}}\mathsf{X}(\mathsf{b})$ where x is an edge set variable and b is a Boolean expression, c^α is true if and only if there exists $E \in E_G$ such that $b^{[X \mapsto E]}$ is true.

where $b^{[x \mapsto i]}$ for a (set) variable x, a constant i, and a Boolean expression b is obtained from b by changing every occurence of x to i. □

The predicate path checks the existence of a (directed) path between two nodes. The predicate $\mathsf{path}(\mathsf{x}, \mathsf{y})$ for some terms x, y representing nodes can also be expressed by monadic second-order formulas without using the predicate path, as can be seen in Lemma 3.10.

Lemma 3.10 (The existence of a directed path)**.**

$$\begin{aligned}\mathsf{path}(\mathsf{x},\mathsf{y}) \equiv \mathsf{x} = \mathsf{y} \vee \exists_{\mathsf{E}}\mathsf{X}(&\exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{u}) = \mathsf{x}) \wedge \exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{u}) = \mathsf{y})\\ &\wedge \neg\exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge (\mathsf{s}(\mathsf{u}) = \mathsf{y} \vee \mathsf{t}(\mathsf{u}) = \mathsf{x}))\\ &\wedge \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{u}) \neq \mathsf{y} \Rightarrow \exists_{\mathsf{e}}\mathsf{v}(\mathsf{v} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{v}) = \mathsf{t}(\mathsf{u})))\\ &\wedge \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \Rightarrow \neg\exists_{\mathsf{e}}\mathsf{v}(\mathsf{v} \neq \mathsf{u} \wedge \mathsf{v} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{v}) = \mathsf{t}(\mathsf{u}))))\end{aligned}$$

Proof. From Definition 3.9, recall that $\mathsf{path}(\mathsf{x},\mathsf{y})$ is true in a graph G if and only if there exists an assignment α such that there exists a directed path from x^α to y^α. From the definition of directed path (see Definition 3.3), $\mathsf{path}(\mathsf{x},\mathsf{y})$ is true if and only if for some assignment α, $\mathsf{x}^\alpha = \mathsf{y}^\alpha$ or there exists edges $e_1,\ldots,e_n \in E_G$ for some n where $s_G(e_1) = \mathsf{x}^\alpha$, $t_G(e_n) = \mathsf{y}^\alpha$, and for all $i = 1,\ldots,n-1$, $t_G(e_i) = s_G(e_{i+1})$.

Note that if there is an edge e_i for $2 \le i \le n$ where $s_G(e_i) = \mathsf{y}^\alpha$, then $t_G(e_{i-1}) = \mathsf{y}^\alpha$ such that the edge sequence $e_1,\ldots,e_{i-1}$ also defines directed path from x^α to y^α. Similarly, if there is an edge e_i for $1 \le i \le n-1$ where $t_G(e_i) = \mathsf{x}^\alpha$, then $s_G(e_{i+1}) = \mathsf{x}^\alpha$ such that the edge sequence $e_{i+1},\ldots,e_n$ defines directed path from x^α to y^α. Also, let us consider the case where there exist e_i and e_j for $1 \le i < j \le n$ where $t_G(i) = t_G(j)$. If $t_G(e_i) = y$, we can define the directed path from $e_1,\ldots,e_i$. Otherwise, we can define the directed path from $e_1,\ldots,e_i,e_j+1,\ldots,e_n$. Hence, we can always assume that there is no edge in the sequence whose source is y^α, or whose target is x^α, or whose target is the same with the target of another edge in the sequence.

With that assumption, let us consider the set of edges $X = \{e_1,\ldots,e_n\}$. Since $s_G(e_1) = \mathsf{x}^\alpha$, $t_G(e_n) = \mathsf{y}^\alpha$, there is no $x \in X$ where $s_G(x) = \mathsf{y}^\alpha$ or $t_G(x) = \mathsf{x}^\alpha$, and for all e_i, $t_G(e_i) = \mathsf{y}^\alpha$ or $t_G(e_i) = s_G(e_{i+1})$. With support of our assumption, the following formula must be hold:

$$
\begin{aligned}
\mathsf{x} = \mathsf{y} \vee \exists_{\mathsf{E}}\mathsf{X}(&\exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{u}) = \mathsf{x}) \wedge \exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{u}) = \mathsf{y}) \\
&\wedge \neg\exists_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge (\mathsf{s}(\mathsf{u}) = \mathsf{y} \vee \mathsf{t}(\mathsf{u}) = \mathsf{x})) \\
&\wedge \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{u}) \neq \mathsf{y} \Rightarrow \exists_{\mathsf{e}}\mathsf{v}(\mathsf{v} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{v}) = \mathsf{t}(\mathsf{u}))) \\
&\wedge \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \Rightarrow \neg\exists_{\mathsf{e}}\mathsf{v}(\mathsf{v} \neq \mathsf{u} \wedge \mathsf{v} \in \mathsf{X} \wedge \mathsf{t}(\mathsf{v}) = \mathsf{t}(\mathsf{u}))))
\end{aligned}
$$

From the other hand, if there exists a set of node X such that the above formula is true, then there exists an edge $e_1 \in X$ whose source is x^α and e_n whose target is y^α. If $t_G(e_1) \neq y$, then there must exists edge $e_2 \in X$ where $s_G(e_2) = t_G(e_1)$, $t_G(e_2) \neq \mathsf{x}^\alpha$, and $t_G(e_2) \neq t_G(e_i)$ for $i = 1$. Similarly, if $t_G(e_2) \neq y$, then there must exists edge $e_3 \in X$ where $s_G(e_3) = t_G(e_2)$, $t_G(e_3) \neq \mathsf{x}^\alpha$, and $t_G(e_3) \neq t_G(e_i)$ for $i = 1,2$. Similar property must hold for all $e \in X$, and since we have e_n where $t(e_n) = \mathsf{y}^\alpha$, the sequence $e_1,\ldots,e_n$ defines the directed path from x^α to y^α. □

The problem of checking whether a formula is valid over all (finite) graphs (i.e. $\vDash c$?) is undecidable. Trakhtenbrot's theorem [42] states "for every relational vocabulary σ with at least one binary relation symbol, it is undecidable whether a sentence Φ of vocabulary δ is finitely satisfiable". In our monadic second-order formula, we may have binary relation edge and path.

The problem of checking whether a graph G satisfies a formula c (i.e. $G \vDash c$?), or also called a model checking problem, is also undecidable. The undecidability come from the possibility

of having Boolean sub-expression that does not have any relation with graphs, but only talks about arithmetic. From our syntax for MSO formulas, it is possible to have a formula (or subformula) in Peano arithmetic, since we have the integer operator *. Peano arithmetic is known to be undecidable [52], so we may have a formula where its satisfiability is undecidable.

However, when we do not have multiplications, the possible arithmetic occurring in a formula will be a Presburger arithmetic, which is known to be decidable [52]. Hence, if a formula has no multiplication in arithmetic expression or only talks about graph structure, its satisfiability is decidable. It is decidable because there is a finite number of ways of assigning variables to elements of a host graph. Hence in the worst case, we can try each possibility to check if an assignment of c on G yields $G \vDash c$.

3.3 Structural induction on monadic second-order formulas

In this study, we will need to define or prove some properties related to our MSO formulas. Here, we define a structural induction on MSO formulas to show that a property holds.

For simplicity, in the structural induction we do not consider the predicates edge and path because we can express both predicates in another way. For $\mathsf{edge}(x, y)$, we can express it with $\exists_e z(\mathsf{s}(z) = x \wedge \mathsf{t}(z) = y)$. The optional arguments label and mark of the predicate edge, e.g. the predicate $\mathsf{edge}(x, y, 5, \mathsf{none})$, can be expressed as:
$\exists_e z(\mathsf{s}(z) = x \wedge \mathsf{t}(z) = y \wedge \mathsf{l}_e(z) = 5 \wedge \mathsf{m}_e z = \mathsf{none})$.

Then the predicate $\mathsf{path}(x, y)$ can also be express by MSO formula as in Lemma 3.10. The optional argument edge-set variable Y can be added by conjunct the formulas inside the edge-set quantifier with $\neg(Y \notin X)$.

We also do not consider the following forms of formula in structural induction : $X = Y, X \neq Y$, $X \subset Y$, and $X \subseteq Y$. For node set variables X, Y, we can replace the formulas with:
$\forall_v x((x \in X \Rightarrow x \in Y) \wedge (x \in Y \Rightarrow x \in X))$, $\neg(X = Y)$, $\forall_v x(x \in X \Rightarrow x \in Y)$, and $X \subset Y \vee X = Y$ respectively.
Similarly for edge set variables, we only need to change node quantifiers to edge quantifiers.

To get the intuition to prove a property, we always start with proving the property for first-order formulas before we have the complete proof for monadic second-order formulas. Hence, we have first-order formulas as a base case for structural induction on monadic second-order formulas.

To prove properties related to our first-order formulas, we classify first-order formulas into eight cases, based on their forms. To prove that some properties hold for these cases, we

define structural induction on first-order formulas. Three cases are defined as base cases since they are formed from terms while the others are defined as inductive cases since they can be formed from other FO formulas.

As mentioned before, terms can exist as a variable, a constant, or functions (including operators). Here, we also define a structural induction on terms, whose base cases are variables and constants.

Definition 3.11 (Structural induction on first-order terms)**.**
Let *Prop* be a property. Proving that *Prop* holds for all terms by *structural induction on FO terms* is done by:

- Base case.
  Show that *Prop* holds for all nodes, edges, and lists represented by node, edge, or label variables and constants.
- Inductive case.
  Assuming that *Prop* holds for lists x_1, x_2, integers i_1, i_2, strings s_1, s_2, a node v, and an edge e, show that *Prop* also holds for:
  1. integers result of $\mathsf{length}(\mathsf{x_1})$, and $\mathsf{i_1} \oplus \mathsf{i_2}$ for $\oplus \in \{,-,*,/\}$
  2. lists result of $\mathsf{l_e}(\mathsf{e_1})$ and $\mathsf{l_v}(\mathsf{v_1})$
  3. marks result of $\mathsf{m_e}(\mathsf{e_1})$ and $\mathsf{m_v}(\mathsf{v_1})$
  4. strings result of $\mathsf{s_1}.\mathsf{s_2}$ □

Definition 3.12 (Structural induction on first-order formulas)**.**
Let *Prop* be a property. Proving that *Prop* holds for all FO formulas by *structural induction on FO formulas* is done by:

- Base case.
  Show that *Prop* holds for:
  1. the formulas true and false
  2. predicates $\mathsf{int(z)}, \mathsf{char(z)}, \mathsf{string(z)}, \mathsf{atom(z)}$ for a list variable z, and $\mathsf{root(y)}$ for a term y representing a node
  3. Boolean operations $\mathsf{x_1} = \mathsf{x_2}$ and $\mathsf{x_1} \neq \mathsf{x_2}$ where both x_1, x_2 are terms representing nodes, edges, or lists, also $\mathsf{y_1} \ominus \mathsf{y_2}$, for terms y_1, y_2 representing integers and $\ominus \in \{=, \neq, <, \leq, >, \geq\}$
- Inductive case.
  Assuming that *Prop* holds for FO formulas c_1, c_2, show that *Prop* also holds for FO formulas $c_1 \wedge c_2$, $c_1 \vee c_2$, $\neg c_1$, $\exists_{\mathsf{v}}\mathsf{x}(c_1)$, $\exists_{\mathsf{e}}\mathsf{x}(c_1)$, and $\exists_{\mathsf{l}}\mathsf{x}(c_1)$. □

After we prove that a property holds for first-order formulas, we can extend the prove by showing by structural induction that it also holds for monadic second-order formulas.

Definition 3.13 (Structural induction on terms)**.**
Let *Prop* be a property. Proving that *Prop* holds for all terms by *structural induction on terms* is done by:

1. Show that *Prop* holds for all first-order terms, node set variables, and edge set variables.
2. Show that *Prop* also holds for the integer result of $\mathsf{card}(\mathsf{X})$ for any set variable X □

Definition 3.14 (Structural induction on monadic second-order formulas)**.**
Let *Prop* be a property. Proving that *Prop* holds for all monadic second-order formulas by *structural induction on monadic second-order formulas* is done by:

- Base case.
  Show that *Prop* holds for:
  1. all first-order formulas
  2. Boolean operations in the form $\mathsf{x} \in \mathsf{X}$ or $\mathsf{x} \notin \mathsf{X}$ where x and X are terms representing node (or edge) and set of nodes (or set of edges) respectively
  3. Boolean operations in the form $\mathsf{x} \otimes \mathsf{card}(\mathsf{X})$ for $\otimes \in \{=, \neq, <, \leq, >, \geq\}$, a node (or edge) set variable X, and a term x representing an integer
- Inductive case.
  Assuming that *Prop* holds for monadic second-order formulas c_1, c_2, show that *Prop* also holds for monadic second-order formulas $c_1 \wedge c_2$, $c_1 \vee c_2$, $\neg c_1$, $\exists_{\mathsf{v}}\mathsf{x}(c_1)$, $\exists_{\mathsf{e}}\mathsf{x}(c_1)$, $\exists_{\mathsf{I}}\mathsf{X}(c_1)$, $\exists_{\mathsf{V}}\mathsf{X}(c_1)$, and $\exists_{\mathsf{e}}\mathsf{x}(c_1)$. □

In the induction on monadic second-order formulas, we do not consider the Boolean operations in the form $y \otimes \mathsf{card}(\mathsf{X}) \oplus z$ for some $\otimes \in \{=, \neq, <, \leq, >, \geq\}, \oplus \in \{+, -, /, *\}$, and terms y, z representing integers. This is because we can always change them to the form $x \otimes \mathsf{card}(\mathsf{X})$ where x is an integer operation on y and the inverse of z w.r.t. $\oplus$.

Later in following chapter, we limit our pre- and postcondition to closed formulas. Closed formulas can be defined inductively as follows:

1. the formulas true or false
2. predicates $\mathsf{int}(\mathsf{x}), \mathsf{char}(\mathsf{x}), \mathsf{string}(\mathsf{x}), \mathsf{atom}(\mathsf{x})$ for some list variable x
3. Boolean operations $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable

4. Boolean operations $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant
5. Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge (set) variable
6. Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x} = \mathsf{s(y)}$ or $\mathsf{x} = \mathsf{t(y)}$ for some bounded edge variable y
7. $\exists_{\mathsf{l}}\mathsf{x}(\mathsf{c_1})$ for some closed formula c_1
8. $\exists_{\mathsf{v}}\mathsf{x}\,(\mathsf{c_1})$ for some closed formula c_1
9. $\exists_{\mathsf{e}}\mathsf{x}\,(\mathsf{c_1})$ for some closed formula c_1
10. $\exists_{\mathsf{V}}\mathsf{X}(\mathsf{c_1})$ for some closed formula c_1
11. $\exists_{\mathsf{E}}\mathsf{X}(\mathsf{c_1})$ for some closed formula c_1
12. $c_1 \vee c_2$ for some closed formula c_1, c_2
13. $c_1 \wedge c_2$ for some closed formula c_1, c_2
14. $\neg c_1$ for some closed formula c_1, c_2

We choose to limit the pre- and postcondition to closed formulas to avoid ambiguity on the type of variables we use in the formulas. By having closed formulas, we can easily identify a variable's type from the quantifier that binds it.

3.4 Monadic second-order formulas in rule schema application

MSO formulas we define in this chapter do not have a node or edge constant because we want to be able to check the satisfaction of an MSO formula on any given graph. However, in a rule schema application, we sometimes need to express the properties of the images of the match or comatch, which is dependent on the left-hand graph or right-hand graph. To be able to express properties based on a match or comatch, we need to allow some node and edge constants in MSO formulas. Hence, we define a condition over a graph.

Definition 3.15 (Conditions over a graph)**.** A *condition* is a monadic second-order formula without free node and edge variables. A *condition over a graph G* is a monadic second-order formula where every free node and edge variable is replaced with node and edge identifiers

in G. That is, if c is a monadic second-order formula and α_G is an assignment of free node and edge variables of c on G, then c^{α_G} is a condition over G. □

Basically, a condition over a graph is a closed monadic second-order formula that allow node and edge constants. For a monadic second-order formula c, a graph G, a formula assignment $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$, both c^{α_G} and c^{α} are conditions over G.

Example 3.2. Let G and H be graphs where $V_G = \{1, 2\}$ and $V_H = \{1\}$.

1. $c_1 = \exists_{\mathsf{e}} \mathsf{x}(\mathsf{s}(\mathsf{x}) = 1)$ is a condition over G, also over H
2. $c_2 = \forall_{\mathsf{v}} \mathsf{x}(\mathsf{edge}(\mathsf{x}, 1) \wedge \mathsf{indeg}(\mathsf{x}) = 2)$ is a condition over G, but not over H

Checking if a graph satisfies a condition over a graph is similar with checking satisfaction of a FO formula in a graph. However, the satisfaction of condition c in a graph G can be defined if and only if c is a condition over G.

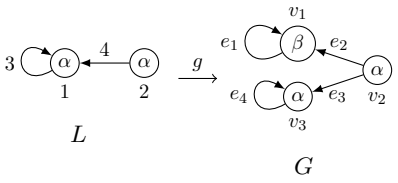
With a condition over a graph, we can express properties of left and right-hand graphs with explicitly mentioning node/edge identifiers in the graphs. In graph program verification, we need to express the initial and output graph's properties with respect to the given rule schema. In [1, 26], they express them by showing the satisfaction of a condition on a morphism. Here, we define a replacement graph H of a host graph G with respect to an injective morphism g, where H is isomorphic to G and there exists an inclusion from the domain of g to H.

Definition 3.16 (Replacement graph)**.** Let us consider an injective morphism $g : L \to G$ for host graphs L and G. Graph $\rho_g(G)$ is a replacement graph of G w.r.t. g if $\rho_g(G)$ is isomorphic to G with L as a subgraph. □

Let consider the injective morphism $g : L \to G$ where $V_G \cap V_L = \{v_1, \ldots, v_n\}$ and $E_G \cap E_L = \{e_1, \ldots, e_m\}$. Let also $U = \{u_1, \ldots, u_n\}$ be a set of identifiers not in V_L and V_G, and $W = \{w_1, \ldots, w_n\}$ be a set of identifiers not in E_L and E_G. Graph replacement $\rho_g(G)$ can be obtained from G by renaming every item $g(i)$ to i for $i \in V_G$ and $i \in E_G$, every v_i to u_i for $i = 1, \ldots, n$, and every e_i to w_i for $i = 1, \ldots, m$, such that $V_{\rho_g(G)} = (V_G - g(V_L)) \cup V_L \cup U$ and $E_{\rho_g(G)} = (E_G - g(E_L)) \cup E_L \cup W$.

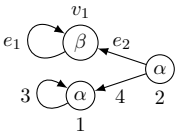
From the definition of a replacement graph, it is obvious that a host graph and its replacement graph are isomorphic. For a host graph G, a host graph L, and a morphism $g : L \to G$, it is also obvious that there exists an inclusion $f : L \to \rho_g(G)$, because g preserves identifiers, sources, targets, and labels of L.

Example 3.3. Let us consider graphs L and H, also an injective morphism $g : L \to G$ as follows:



$$g = \left\langle g_V : \left\{ \begin{array}{l} 1 \mapsto v_3 \\ 2 \mapsto v_2 \end{array} \right. , g_E : \left\{ \begin{array}{l} 3 \mapsto e_4 \\ 4 \mapsto e_3 \end{array} \right. \right\rangle$$

Then, $\rho_g(G)$ is the graph



In a rule schema application, an injective morphism g is a match if it satisfies the dangling condition and the rule schema's condition (if any). These conditions can be considered as application conditions of a rule schema. In a rule schema application, we only have application conditions with respect to the matches, but not comatch. Here, we define a generalised rule schema that allows us to consider comatches in its applications, and consider an unrestricted rule schema instead of rule schema. By considering unrestricted rule schema and application condition for the comatches, the conditional rule schema become inverted. This address the issue stated in [1] about the disability of constructing strongest liberal postcondition from the construction of weakest liberal precondition, or vice versa, because conditional rule schemata are invertible.

Definition 3.17 (Generalised rule schema)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. *A generalised rule schema* is a tuple $w = \langle r, ac_L, ac_R \rangle$ where ac_L is a condition over L and ac_R is a condition over R. We call ac_L the left application condition and ac_R the right application condition. The inverse of w, written w^{-1}, is then defined as the tuple $\langle r^{-1}, ac_R, ac_L \rangle$ where $r^{-1} = \langle R \leftarrow K \rightarrow L \rangle$. □

The application of a generalised rule schema is similar to the application of a rule schema. However, here we also check the satisfaction of both left and right application conditions in the replacement of input graph G and final graph H with respect to the match and comatch respectively.

Definition 3.18 (Application of generalised rule schema)**.** Let $w = \langle r, ac_L, ac_R \rangle$ be a generalised rule schema with an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. There exists a direct derivation from G to H by w, written $G \Rightarrow_{w,g,g^*} H$ (or $G \Rightarrow_w H$) iff there exists premorphism $g : L \to G$ and $g^* : R \to H$ and label assignments α_L and β_R where $\beta_R(i) = \alpha_L(i)$ for every common variable i in L and R, also for every node/edge i where $m_L(i) = m_R(i) = \texttt{any}$, such that:

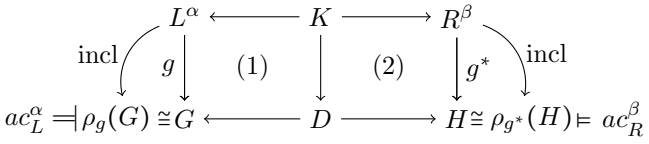


FIGURE 3.2: Direct derivation for generalised rule schema

(i) $g : L^\alpha \to G$ is an injective morphism

(ii) $g^* : R^\beta \to H$ is an injective morphism

(iii) $\rho_g(G) \models ac_L^\alpha$,

(iv) $\rho_{g^*}(H) \models ac_R^\beta$,

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

where $G \Rightarrow_{r^\alpha, g} H$ denotes the existence of natural pushouts (1) and (2) as in the diagram of Figure 3.2. □

Recall the application of conditional rule schema in Definition 2.22. The rule schema's condition is clearly can be considered the left-application condition of the rule schema. Since there is no right-application condition in a conditional rule schema, there is no requirement about the condition. We can always consider true as the right-application condition of a conditional rule schema.

Definition 3.19 (Generalised version of a conditional rule schema)**.** Let us consider a conditional rule schema $\langle r, \Gamma \rangle$. The generalised version of r, denoted by $r^\vee$, is the generalised rule schema $r^\vee = \langle r, \Gamma^\vee, \mathsf{true} \rangle$ where $\Gamma^\vee$ is obtained from Γ by replacing the notations $\mathsf{not}, !=, \mathsf{and}, \mathsf{or}, \#$ with $\neg, \neq, \wedge, \vee, `,'$ (comma symbol) respectively. □

Lemma 3.20. Let $\langle r, \Gamma \rangle$ be a conditional rule schema with $r = \langle L \leftarrow K \rightarrow R \rangle$. Then for any host graphs G, H,

$$G \Rightarrow_r H \text{ if and only if } G \Rightarrow_{r^\vee} H.$$

Proof. (Only if). Recall the restrictions about variables and any-mark of a rule schema. It is obvious that every variable in R is in L and every node/edge with mark any in R is marked any in L as well. From Definition 2.22, we know that $G \Rightarrow_r H$ asserts the existence of α_L and premorphism $g : L \to G$ such that: 1) $g : L^\alpha \to G$ is an injective morphism, 2) $\Gamma^{\alpha,g}$ is true in G, and 3) $G \Rightarrow_{r^{\alpha,g},g} H$. From 3) and the variable restrictions mentioned above, it is obvious that there exists morphism $g^* : R^\alpha \to H$, and $\rho_{g^*}(H) \models ac_R$ because all graphs satisfy true. Hence, (ii), (iv), and (v) of Definition 3.18 are satisfied. Point 1) then asserts (i) of Definition 3.18. The fact that $\Gamma^{\alpha,g}$ is true in G from point 2) is then asserts

$\rho_g(G) \vDash \Gamma^\vee$ because it is obvious that the change of symbols does not change the semantics of the condition. Moreover, $\rho_g(G)$ is a replacement graph w.r.t. g such that evaluating $\Gamma^{\alpha,g}$ in G is the same as evaluating Γ^α in $\rho_g(G)$.
(If). Similarly, from Definition 3.18, we know that $G \Rightarrow_{r^\vee} H$ asserts the existence of label assignment α_L and premorphism $g : L \to G$ such that: 1) $g : L^\alpha \to G$ is an injective morphism, 2)$\rho_g(G) \vDash \Gamma^\vee$, and 3)$G \Rightarrow_{r^{\alpha,g},g} H$. These obviously assert $G \Rightarrow_r H$ from Definition 2.22 and the argument about $\Gamma^\vee$ above. □

Remark 3.21. For morphism $g : L^\alpha \to G$, the semantics of Γ in G with respect to g and $\Gamma^\vee$ in $\rho_g(G)$ is identical. From here, Γ also refers to $\Gamma^\vee$ when it obviously refers to a condition over L.

Note that the inverse of a rule schema is not always defined, because a rule schema only allows condition for left-hand graph, and there are some restrictions for left and right-hand graphs' labels. Of course, the restrictions are useful in practice. However, since we are interested in observing the properties of a rule schema application, we also want to know the properties of the inverse of a rule. In a generalised rule schema, we omit the restrictions that may exist in a standard rule schema, so that now we can observe properties of the inverse of a generalised rule schema.

Lemma 3.22. Let $w = \langle r, ac_L, ac_R \rangle$ be a generalised rule schema with an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and label assignment α_L. Then for host graphs G and H with premorphisms $g : L \to G$ and $g^* : R \to H$,

$$G \Rightarrow_{w,g,g^*} H \text{ if and only if } H \Rightarrow_{w^{-1},g^*,g} G.$$

Proof.
(Only if.) From the definition of generalised rule schema application (Definition 3.18), we know that when $G \Rightarrow_{w,g,g^*} H$, it means that there exists label assignment α_L and β_R where $\alpha_L(i) = \alpha_R(i)$ for every common variable i in L and R, and for every node/edge i where $m_L(i) = m_R(i) = \texttt{any}$, such that $g : L^\alpha \to G$ and $g^* : R^\beta \to H$ are injective morphisms where

(i) $\rho_g(G) \vDash ac_L^\alpha$

(ii) $\rho_{g^*}(H) \vDash ac_R^\beta$

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

These are obviously defines direct derivation $H \Rightarrow_{(r^{-1})^{g^*,\alpha},g^*} G$ such that $H \Rightarrow_{w^{-1},g^*,g} G$.
(If). We can apply the above proof analogously. □

The application of a rule schema depends on the existence of morphisms. Showing the existence of a morphism $L \to G$ for host graphs L, G can be done by checking the existence of the structure of L in G. For this, we define a condition over a graph to specify the structure and labels of a graph.

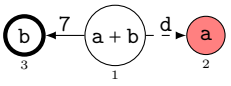
Definition 3.23 (Specifying a totally labelled graph)**.** Let us consider a totally labelled graph L with $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. Let $X = \{x_1, \ldots, x_k\}$ be the set of all list variables in L, and $\mathrm{Type}(x)$ for $x \in X$ is $\mathsf{int(x)}$, $\mathsf{char(x)}$, $\mathsf{string(x)}$, $\mathsf{atom(x)}$, or true if x is an integer, char, string, atom, or list variable respectively. Let also $\mathrm{Root}_L(v)$ for $v \in V_L$ be a function such that $\mathrm{Root}_L(v) = \mathsf{root(v)}$ if $p_L(v) = 1$, and $\mathrm{Root}_L(v) = \neg\mathsf{root(v)}$ otherwise. *A specification of* L, denoted by $\mathrm{Spec}(L)$, is the condition over L:

$$\bigwedge_{i=1}^{k} \mathrm{Type}(x_i) \ \wedge \ \bigwedge_{i=1}^{n} \mathsf{l_v(v_i)} = l_L^V(v_i) \ \wedge \ \mathsf{m_v(v_i)} = m_L^V(v_i) \ \wedge \ \mathrm{Root}_L(v_i)$$

$$\wedge \bigwedge_{i=1}^{m} \mathsf{s(e_i)} = s_L(e_i) \ \wedge \ \mathsf{t(e_i)} = t_L(e_i) \ \wedge \ \mathsf{l_e(e_i)} = l_L^E(e_i) \ \wedge \ \mathsf{m_e(e_i)} = m_L^E(e_i) \quad \square$$

Since morphisms require the preservation of sources, targets, labels, and rootedness, we need to explicitly state rootedness and label of each node, source and target of each edge. Also, since we also want to specify rule graphs, the type of each variable needs to explicitly stated as well. Note that we only specify totally labelled graphs, so that the label and rootedness of a node are always defined.

Example 3.4 (Specification of L)**.** Let us consider the graph L below:



where the edge incident to 1 and 2 is edge $e1$ and the other one is edge $e2$, and a, b, are integer variables while d is a list variable. Then, $\mathrm{Spec}(L)$ is the condition over L:

$$\mathsf{int(a)} \wedge \mathsf{int(b)} \wedge \mathsf{l_v(1) = a + b} \wedge \mathsf{l_v(2) = a} \wedge \mathsf{l_v(3) = b}$$

$$\wedge \mathsf{m_v(1) = none} \wedge \mathsf{m_v(2) = red} \wedge \mathsf{m_v(3) = none} \wedge \neg\mathsf{root(1)} \wedge \neg\mathsf{root(2)} \wedge \mathsf{root(3)}$$

$$\wedge \mathsf{s(e1) = 1} \wedge \mathsf{t(e1) = 2} \wedge \mathsf{s(e2) = 1} \wedge \mathsf{t(e2) = 3} \wedge \mathsf{l_e(e1) = d} \wedge \mathsf{l_e(e2) = 7}$$

$$\wedge \mathsf{m_e(e1) = dashed} \wedge \mathsf{m_e(e2) = none}$$

Let us consider a graph G that satisfies $\mathrm{Spec}(L)$, where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. From the definition of $\mathrm{Spec}(L)$, the satisfaction means that $V_L \subseteq V_G$ and $E_L \subseteq E_G$, where $s_G(e) = s_L(e), t_G(e) = t_L(e)$, and $p_G(v) = p_L(v)$ for all $v \in V_L$ and $e \in E_L$. As for labels and marks, if G and L are both host graphs, then there is no variable for the labels

and marks so that $l^V_G(v) = l^V_L(v), l^E_G(e) = l^E_L(e), m^V_G(v) = m^V_L(v)$, and $m^E_G(e) = m^E_L(e)$. However, if G is a host graph and L is a rule graph, then L may contain a label or mark variable. This means, there must exists a label and assignment α on L such that $l^V_G(v) = (l^V_L(v))^\alpha, l^E_G(e) = (l^E_L(e))^\alpha, m^V_G(v) = (m^V_L(v))^\alpha$, and $m^E_G(e) = (m^E_L(e))^\alpha$. This means, there must exists an inclusion $L^\alpha \to G$.

Proposition 3.24 (Spec(L) and inclusion)**.** Let us consider a rule graph L and a host graph G where $V_L \subseteq V_G$ and $E_L \subseteq E_G$. Then, $G \vDash \text{Spec}(L)$ if and only if there exists a label assignment α_L such that there exists inclusion $g : L^\alpha \to G$.

Proof. Let us consider the construction of Spec(L). From the definition of Spec, we can see that there are no node or edge variables in the condition. Hence, G satisfies Spec(L) if and only if there exists an assignment β for all list variables in Spec(L) and a partial function $\mu = \langle \mu_V : V_L \to \mathbb{M}\backslash\{\texttt{none}, \texttt{dashed}\}, \mu_E : E_L \to \mathbb{M}\backslash\{\texttt{nonegrey}\}\rangle$ for every node/edge, i whose mark is `any` such that substituting $\beta(x)$ for every variable x and $\mu(i)$ for every `any`-mark associated with i in Spec(L) resulting a valid statement in G.

Let we denote by $V_L = \{v_1, \ldots, v_n\}$, $E_L = \{e_1, \ldots, e_m\}$, and $X = \{x_1, \ldots, x_p\}$ the set of all nodes, edges, and label variables in L. From the semantics of satisfaction, we know that that $G \vDash \text{Spec}(L)$ iff the following formula is true in G:

$$\bigwedge_{i=1}^{n} l^V_G(v_i) = (l^V_L(v_i))^\beta \ \wedge\ m^V_G(v_i) = (m^V_L(v_i))^\mu \ \wedge\ \text{Root}_G(v_i)$$

$$\wedge \bigwedge_{i=1}^{m} s_G(e_i) = s_L(e_i) \ \wedge\ t_G(e_i) = t_L(e_i) \ \wedge\ l^E_G(e_i) = (l^E_L(e_i))^\beta \ \wedge\ m^E_G(e_i) = (m^E_L(e_i))^\mu$$

Define $g(i) = i$ for every item $i \in V_L$ and $i \in E_L$ (such that identifiers are preserved by g), and $\alpha = \langle \beta, \mu_V, \mu_E \rangle$. We can see from the conjunction above that sources, targets, lists, marks, and rootedness for each v_i and e_i are the same in G and L, so that g preserves sources, targets, lists, marks, and rootedness. □

Note that Spec(L) is a condition over L, so a graph satisfying the condition must have node and edge identifiers of L in the graph. It is obviously not practical, but we can make it more general by replacing the identifiers with fresh variables such that a graph satisfying the condition does not necessarily contain identifiers of L.

Definition 3.25 (Variablisation of a condition over a graph)**.** Let us consider a graph L and a condition c over L where $\{v_1, \ldots, v_n\}$ and $\{e_1, \ldots, e_m\}$ represent the set of node and edge constants in c respectively. Let $x_1, \ldots, x_n$ be node variables not in c and $y_1, \ldots, y_m$ be edge

variables not in c. *Variablisation of* c, denoted by $\mathrm{Var}(c)$, is the FO formula

$$\bigwedge_{i=1}^{n}\bigwedge_{j\neq i} x_i \neq x_j \ \wedge \ \bigwedge_{i=1}^{m}\bigwedge_{j\neq i} y_i \neq y_j \ \wedge \ c^{[v_1\mapsto x_1]\ldots[v_n\mapsto x_n][e_1\mapsto y_1]\ldots[e_m\mapsto y_m]}$$

where $c^{[a\mapsto b]}$ is obtained from c by replacing every occurrence of a with b, and $c^{[a\mapsto b][d\mapsto e]} = (c^{[a\mapsto b]})^{[d\mapsto e]}$. □

Observe that by the variablisation, we only change node and edge constants to node and edge variables. Hence, when a graph G satisfies a condition over G, graphs that are isomorphic to G should satisfy the variablisation of the condition. This is because the variablisation does not change any properties expressed by the condition; it only makes it general so that it does not depend on node/edge identifiers.

Lemma 3.26. Let us consider a graph L and a condition c over L. For every host graph G and morphism $g : L \to G$,

$$G \vDash \mathrm{Var}(c) \text{ if and only if } \rho_g(G) \vDash c.$$

Proof. Let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ represent the set of node and edge constants in c respectively, and $X = x_1, \ldots, x_n$ be node variables not in c and $Y = y_1, \ldots, y_m$ be edge variables not in c such that $\mathrm{Var}(c)$ is the FO formula shown in the definition above.
Let α_G be an assignment such that $\alpha_G(x_i) = v_i$ and $\alpha_G(y_i) = e_i$ for all $x_i \in X$ and $y_i \in Y$. It is obvious that $(\mathrm{Var}(c))^{\alpha_G} \equiv c$, since we only replace each node/edge variable with the constant that was replaced by the variable to obtain $\mathrm{Var}(c)$. Therefore, $\rho_g(G) \vDash c$ iff $\rho_g(G) \vDash \mathrm{Var}(c)^{\alpha_G}$ iff $G \vDash \mathrm{Var}(c)^{\alpha_G}$, which means that G satisfies $\mathrm{Var}(c)$. □

From Proposition 3.24, we understand that a satisfaction of $\mathrm{Spec}(L)$ for a graph L in a graph G also refer to the existence of inclusion $L^\alpha \to G$ for some assignment α_L. When we consider variablisation of $\mathrm{Spec}(L)$, it means that we do not have node/edge identifiers anymore. So, instead of inclusion, the variablisation should refer to morphisms.

Lemma 3.27. Let L be a rule graph and G be a host graph. Then, $G \vDash \mathrm{Var}(\mathrm{Spec}(L))$ if and only if there exists a label assignment α such that there exists injective morphism $g : L^\alpha \to G$.

Proof. G satisfying $\mathrm{Var}(\mathrm{Spec}(L))$ if and only if there exists formula assignment $\gamma = \langle \gamma_V, \gamma_L.\gamma_{\mathbb{L}} \rangle$ and a mapping $\mu = \langle \mu_V : V_L \to \mathbb{M}\backslash\{\texttt{none}, \texttt{dashed}\}, \mu_E : E_L \to \mathbb{M}\backslash\{\texttt{none}, \texttt{grey}\} \rangle$ for every item i whose mark is `any`, such that $(\mathrm{Var}(\mathrm{Spec}(L))^\gamma)^\mu$ is true in G.

If we consider $\mathrm{Var}(\mathrm{Spec}(L))^{\gamma_G}$, it clearly gives us a condition similar to $\mathrm{Spec}(L)$, but with different identifiers. Let X denotes the set of images of γ_G, and $\beta : (V_L \cup G_L) \to X$ be a bijective mapping such that $\mathrm{Spec}(L)^{\beta} = \mathrm{Var}(\mathrm{Spec}(L))^{\gamma_G}$.

Let we denote by $V_L = \{v_1, \ldots, v_n\}$, $E_L = \{e_1, \ldots, e_m\}$, and $X = \{x_1, \ldots, x_p\}$ the set of all nodes, edges, label variables in L. From the semantics of satisfaction, we know that that:

$$\bigwedge_{i=1}^{n} l_G^V(\beta(v_i)) = (l_L^V(v_i))^{\gamma_{\mathbb{L}}} \ \wedge \ m_G^V(\beta(v_i)) = (m_L^V(v_i))^{\mu} \ \wedge \ \mathrm{Root}_G(\beta(v_i))$$

$$\wedge \bigwedge_{i=1}^{m} s_G(\beta(e_i)) = s_L(e_i) \ \wedge \ t_G(\beta(e_i)) = t_L(e_i) \ \wedge \ l_G^E(\beta(e_i)) = (l_L^E(e_i))^{\gamma_{\mathbb{L}}} \ \wedge \ m_G^E(\beta(e_i)) = (m_L^E(e_i))^{\mu}$$

Define $g(i) = \beta(i)$ for every item $i \in V_L \cup E_L$, and $\alpha = \langle \gamma, \mu \rangle$. We can see from the conjunction above that sources, targets, lists, marks, and rootedness for each v_i and e_i in L is the same as their the match, so that $g : L^{\alpha} \to G$ preserves sources, targets, lists, marks, and rootedness. □

3.5 Properties of monadic second-order formulas

In the later chapters, we use the monadic second-order formulas as assertions for verifying graph programs. To prove the soundness of the use of the formulas, we need some basic properties of the formulas. Here, we present some properties that will be used in the later chapters.

Lemma 3.28. Let us consider a monadic second-order formula c and two isomorphic host graphs G and H with isomorphism $f : G \to H$. Let $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ and $\beta = \langle \beta_H, \beta_{\mathbb{L}} \rangle$ be formula assignments where $\beta_H(x) = f(\alpha_G(x))$ for every node and edge variable x in c and $\beta_{\mathbb{L}}(x) = \alpha_{\mathbb{L}}(x)$ for every list variable x in c. Then,

$$G \vDash^{\alpha} c \text{ if and only if } H \vDash^{\beta} c$$

Proof. Here, we prove the Lemma inductively.
(Base case).

1. If $c = \mathsf{true}$ or $c = \mathsf{false}$, it is obvious that $G \vDash^{\alpha} c$ iff $H \vDash^{\beta} c$ because there is no graph satisfying false and all graphs satisfy true.
2. If c is a predicate $\mathsf{P(x)}$ for $\mathsf{P} \in \{\mathsf{int}, \mathsf{char}, \mathsf{string}, \mathsf{atom}\}$ and some list variable x, $G \vDash^{\alpha} c$ iff $\mathsf{P}(\mathsf{x}^{\alpha})$ is true in G. Observe that the truth value of $\mathsf{P}(\mathsf{x}^{\alpha})$ is actually independent on G, such that $G \vDash \mathsf{P}(\mathsf{x}^{\alpha})$ iff $H \vDash \mathsf{P}(\mathsf{x}^{\alpha})$. Since $\beta_{\mathbb{L}} = \alpha_{\mathbb{L}}$, $\mathsf{P}(\mathsf{x}^{\alpha}) = \mathsf{P}(\mathsf{x}^{\beta})$ so that $G \vDash \mathsf{P}(\mathsf{x}^{\alpha})$ iff $H \vDash \mathsf{P}(\mathsf{x}^{\beta})$.

3. If $c = \mathsf{root}(\mathsf{x})$ for some term x representing a node, $\mathsf{x}^\beta = g(\mathsf{x}^\alpha)$. Note that G and H are host graphs (which means the rootedness function is a total function). From Definition 2.4 we know that $p_G(x^\alpha) = p_H(g(x^\alpha))$. Hence, $\mathsf{root}(\mathsf{x}^\alpha$ is true in G iff $\mathsf{root}(\mathsf{x}^\beta)$ is true in H.

4. If $c = x_1 \otimes x_2$ for $\otimes \in \{=, \neq, \in\}$ and terms x_1, x_2 representing (set of) edges or nodes, $x_1^\beta = g(x_1^\alpha)$ and $x_2^\beta = g(x_2^\alpha)$. Because g is injective, we know that $x_1^\alpha \otimes x_2^\alpha$ iff $g(x_1^\alpha) \otimes g(x_2^\alpha)$.

5. If $c = x_1 \otimes x_2$ for $\otimes \in \{=, \neq, \leq, \geq\}$ and terms x_1, x_2 representing lists, $x_1^\alpha = x_1^\beta$ and $x_2^\alpha = x_2^\beta$ (note that $l_V(x^\alpha) = l_V(g(x^\alpha)) = l_V(x^\beta)$ for all node variable x in c, and analogously for $l_E(x)$). Since the truth value of $x_1^\alpha \otimes x_2^\alpha$ does not depend on host graphs, $x_1^\alpha \otimes x_2^\alpha$ is true in G iff $x_1^\beta \otimes x_2^\beta$ is true in H.

(Inductive case). Next, we prove the Lemma for the inductive cases. Let c_1, c_2 be FO formulas such that $G \vDash^\alpha c_1$ iff $H \vDash^\beta c_1$ and $G \vDash^\alpha c_2$ iff $H \vDash^\beta c_2$. Also, let $c^{x \mapsto v}$ for some variable x and constant v represents c after replacement of every free variable x in c with v.

1. If $c = \neg c_1$, $G \vDash^\alpha \neg c_1$ iff c_1^α is false in G iff c_1^β is false in H iff $H \vDash^\beta \neg c_1$

2. If $c = c_1 \vee c_2$, $G \vDash^\alpha c_1 \vee c_2$ iff $G \vDash^\alpha c_1 \vee G \vDash^\alpha c_2$ iff $H \vDash^\beta c_1 \vee H \vDash^\beta c_2$ iff $H \vDash^\beta c_1 \vee c_2$

3. If $c = c_1 \wedge c_2$, $G \vDash^\alpha c_1 \wedge c_2$ iff $G \vDash^\alpha c_1 \wedge G \vDash^\alpha c_2$ iff $H \vDash^\beta c_1 \wedge H \vDash^\beta c_2$ iff $H \vDash^\beta c_1 \wedge c_2$

4. $G \vDash^\alpha \exists_{\mathsf{v}}\mathsf{x}(c_1)$ iff $(c_1^\alpha)^{[x \mapsto v]}$ for some $v \in V_G$ is true in G iff $(c_1^\beta)^{[x \mapsto g(v)]}$ is true in H iff $H \vDash^\beta \exists_{\mathsf{v}}\mathsf{x}(c_1)$

5. $G \vDash^\alpha \exists_{\mathsf{e}}\mathsf{x}(c_1)$ iff $(c_1^\alpha)^{[x \mapsto e]}$ for some $e \in E_G$ is true in G iff $(c_1^\beta)^{[x \mapsto g(e)]}$ is true in H iff $H \vDash^\beta \exists_{\mathsf{e}}\mathsf{x}(c_1)$

6. $G \vDash^\alpha \exists_{\mathsf{l}}\mathsf{x}(c_1)$ iff $(c_1^\alpha)^{[x \mapsto i]}$ for some $i \in \mathbb{L}$ is true in G iff $(c_1^\beta)^{[x \mapsto i]}$ is true in H iff $H \vDash^\beta mrm\exists_{\mathsf{l}}x(c_1)$

7. $G \vDash^\alpha \exists_{\mathsf{V}}\mathsf{X}(c_1)$ iff $(c_1^\alpha)^{[X \mapsto V]}$ for some $V \in 2^{V_G}$ is true in G iff $(c_1^\beta)^{[X \mapsto g(V)]}$ is true in H iff $H \vDash^\beta \exists_{\mathsf{V}}\mathsf{X}(c_1)$

8. $G \vDash^\alpha \exists_{\mathsf{E}}\mathsf{X}(c_1)$ iff $(c_1^\alpha)^{[X \mapsto E]}$ for some $E \in 2^{E_G}$ is true in G iff $(c_1^\beta)^{[x \mapsto gEe)]}$ is true in H iff $H \vDash^\beta \exists_{\mathsf{E}}\mathsf{X}(c_1)$

□

Corollary 3.29. *Let us consider two isomorphic host graphs G and H, and a FO formula c. It is true that*

$$G \vDash c \text{ if and only if } H \vDash c$$

Proof. $G \vDash c$ iff there exists an assignment $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ such that $G \vDash^{\alpha} c$. By Lemma 3.28, $G \vDash^{\alpha} c$ iff $H \vDash^{\beta} c$ for $\beta = \langle \beta_H, \alpha_{\mathbb{L}} \rangle$ where $\beta_H(x) = g(\alpha_G(x))$ for all node and edge variables x iff $H \vDash c$. □

Fact 3.1. Let G be a host graph and c_1, c_2 be MSO formulas. Then, the following holds:

1. $G \vDash c_1 \vee c_2$ if and only if $G \vDash c_1 \vee G \vDash c_2$
2. $G \vDash c_1 \wedge c_2$ if and only if $G \vDash^{\alpha} c_1 \wedge G \vDash^{\alpha} c_2$ for some assignment α
3. $G \vDash \neg c_1$ if and only if $\neg(G \vDash^{\alpha} c_1)$ for some assignment α
4. $V_G \neq \varnothing \wedge G \vDash \exists_{\mathsf{v}} x(c_1)$ if and only if $G \vDash c_1$
5. $E_G \neq \varnothing \wedge G \vDash \exists_{\mathsf{e}} x(c_1)$ if and only if $G \vDash c_1$
6. $G \vDash \exists_{\mathsf{l}} x(c_1)$ if and only if $G \vDash c_1$
7. $G \vDash \exists_{\mathsf{V}} X(c_1)$ if and only if $G \vDash c_1$
8. $G \vDash \exists_{\mathsf{E}} X(c_1)$ if and only if $G \vDash c_1$

Furthermore, the above properties also hold if c_1, c_2 are conditions over G.

Lemma 3.30. Let us consider a host graph G and a condition c over G. Let $V = \{v_1, \ldots, v_n\} \subseteq V_G$, $E = \{e_1, \ldots, e_m\} \subseteq E_G$, $2^V = \{V_1, \ldots, V_{2^n}\}$, and $2^E = \{E_1, \ldots, E_{2^m}\}$. Then,

1. $\exists_{\mathsf{v}} \mathsf{x}(\mathsf{c}) \equiv \mathsf{c}^{[\mathsf{x} \mapsto \mathsf{v}_1]} \vee \ldots \vee \mathsf{c}^{[\mathsf{x} \mapsto \mathsf{v}_n]} \vee \exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq \mathsf{v}_1 \wedge \ldots \wedge \mathsf{x} \neq \mathsf{v}_n \wedge \mathsf{c})$
2. $\exists_{\mathsf{e}} \mathsf{x}(\mathsf{c}) \equiv \mathsf{c}^{[\mathsf{x} \mapsto \mathsf{e}_1]} \vee \ldots \vee \mathsf{c}^{[\mathsf{x} \mapsto \mathsf{e}_m]} \vee \exists_{\mathsf{e}} \mathsf{x}(\mathsf{x} \neq \mathsf{e}_1 \wedge \ldots \wedge \mathsf{x} \neq \mathsf{v}_m \wedge \mathsf{c})$
3. $$\begin{aligned}\exists_{\mathsf{e}} \mathsf{x}(\mathsf{c}) \equiv \exists_{\mathsf{e}} \mathsf{x}(\bigvee_{\mathsf{i}=1}^{\mathsf{n}} (\bigvee_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) = \mathsf{v}_\mathsf{i} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v}_\mathsf{j} \wedge \mathsf{c}^{[\mathsf{s}(\mathsf{x}) \mapsto \mathsf{v}_\mathsf{i}, \mathsf{t}(\mathsf{x}) \mapsto \mathsf{v}_\mathsf{j}]}) \\ \vee (\mathsf{s}(\mathsf{x}) = \mathsf{v}_\mathsf{i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t}(\mathsf{x}) \neq \mathsf{v}_\mathsf{j} \wedge \mathsf{c}^{[\mathsf{s}(\mathsf{x}) \mapsto \mathsf{v}_\mathsf{i}]}) \\ \vee (\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) \neq \mathsf{v}_\mathsf{j} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v}_\mathsf{i} \wedge \mathsf{c}^{[\mathsf{t}(\mathsf{x}) \mapsto \mathsf{v}_\mathsf{i}]}) \\ \vee (\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) \neq \mathsf{v}_\mathsf{i} \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{t}(\mathsf{x}) \neq \mathsf{v}_\mathsf{i} \wedge \mathsf{c}))\end{aligned}$$
4. $\exists_{\mathsf{V}} \mathsf{X}(\mathsf{c}) \equiv \exists_{\mathsf{V}} \mathsf{X}(\bigwedge_{\mathsf{i}=0}^{2^\mathsf{n}} (\mathsf{V}_\mathsf{i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j} \in \mathsf{V} - \mathsf{V}_\mathsf{i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathsf{c}))$
5. $\exists_{\mathsf{E}} \mathsf{X}(\mathsf{c}) \equiv \exists_{\mathsf{E}} \mathsf{X}(\bigwedge_{\mathsf{i}=0}^{2^\mathsf{m}} (\mathsf{E}_\mathsf{i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j} \in \mathsf{E} - \mathsf{E}_\mathsf{i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathsf{c}))$

Proof.

1. $$\begin{aligned}\exists_{\mathsf{v}} \mathsf{x}(\mathsf{c}) &\equiv \exists_{\mathsf{v}} \mathsf{x}(((\mathsf{x} = \mathsf{v}_1 \vee \ldots \vee \mathsf{x} = \mathsf{v}_\mathsf{n}) \vee \neg(\mathsf{x} = \mathsf{v}_1 \vee \ldots \vee \mathsf{x} = \mathsf{v}_\mathsf{n})) \wedge \mathsf{c}) \\ &\equiv \exists_{\mathsf{v}} \mathsf{x}((\mathsf{x} = \mathsf{v}_1 \wedge \mathsf{c}) \vee \ldots \vee (\mathsf{x} = \mathsf{v}_\mathsf{n} \wedge \mathsf{c}) \vee (\mathsf{x} \neq \mathsf{v}_1 \wedge \ldots \wedge \mathsf{x} \neq \mathsf{v}_\mathsf{n} \wedge \mathsf{c})) \\ &\equiv \exists_{\mathsf{v}} \mathsf{x}(\mathsf{c}^{[\mathsf{x} \mapsto \mathsf{v}_1]} \vee \ldots \vee \mathsf{c}^{[\mathsf{x} \mapsto \mathsf{v}_\mathsf{n}]} \vee (\mathsf{x} \neq \mathsf{v}_1 \wedge \ldots \wedge \mathsf{x} \neq \mathsf{v}_\mathsf{n} \wedge \mathsf{c}))\end{aligned}$$

$$\equiv \quad c^{[x\mapsto v_1]} \vee \ldots \vee c^{[x\mapsto v_n]} \vee \exists_v x(x \neq v_1 \wedge \ldots \wedge x \neq v_n \wedge c)$$

2. Analogous to point 1

3. $$\exists_e x(c) \quad \equiv \quad \exists_e x(((s(x) = v_1 \vee \ldots \vee s(x) = v_n) \vee \neg(s(x) = v_1 \vee \ldots \vee s(x) = v_n))$$
$$\wedge\, (t(x) = v_1 \vee \ldots \vee t(x) = v_n \vee \neg(t(x) = v_1 \vee \ldots \vee t(x) = v_n)) \wedge c)$$
$$\equiv \quad \exists_e x((s(x) = v_1 \wedge (t(x) = v_1 \vee \ldots \vee t(x) = v_n) \wedge c)$$
$$\ldots$$
$$(s(x) = v_n \wedge (t(x) = v_1 \vee \ldots \vee t(x) = v_n) \wedge c)$$
$$(s(x) \neq v_1 \wedge \ldots \wedge s(x) \neq v_n \wedge (t(x) = v_1 \vee \ldots \vee t(x) = v_n) \wedge c)$$
$$(s(x) = v_i \wedge (t(x) \neq v_1 \wedge \ldots \wedge t(x) \neq v_n) \wedge c)$$
$$(s(x) \neq v_1 \wedge \ldots \wedge s(x) \neq v_n \wedge (t(x) \neq v_1 \wedge \ldots \wedge t(x) \neq v_n) \wedge c)$$
$$\equiv \quad \exists_e x(\textstyle\bigvee_{i=1}^{n}(\bigvee_{j=1}^{n} s(x) = v_i \wedge t(x) = v_j \wedge c^{[s(x)\mapsto v_i, t(x)\mapsto v_j]})$$
$$\vee\, (\textstyle\bigwedge_{j=1}^{n} s(x) \neq v_j \wedge t(x) = v_i \wedge c^{[t(x)\mapsto v_i]})$$
$$\vee\, (s(x) = v_i \wedge \textstyle\bigwedge_{j=1}^{n} t(x) \neq v_j \wedge c^{[s(x)\mapsto v_i]})$$
$$\vee\, (\textstyle\bigwedge_{i=1}^{n} s(x) \neq v_i \wedge \bigwedge_{i=1}^{n} t(x) \neq v_i \wedge c))$$

4. $$\exists_V X(c) \quad \equiv \quad \exists_V X((v_1 \in X \vee v_1 \notin X) \wedge (v_2 \in X \vee v_2 \notin X) \wedge \ldots \wedge (v_n \in X \vee v_n \notin X) \Rightarrow c)$$
$$\equiv \quad \exists_V X(((v_1 \in X \wedge v_2 \in X) \vee (v_1 \in X \wedge v_2 \notin X)$$
$$\vee\, (v_1 \notin X \wedge v_2 \in X) \vee (v_1 \notin X \wedge v_2 \notin X))$$
$$\wedge\, (v_3 \in X \vee v_3 \notin X) \wedge \ldots \wedge (v_n \in X \vee v_n \notin X) \Rightarrow c)$$
$$\equiv \quad \exists_V X((((\{v_1, v_2\} \subseteq X) \vee (\{v_1\} \subseteq X \wedge v_2 \notin X)$$
$$\vee\, (\{v_2\} \subseteq X \wedge v_1 \notin X) \vee (v_1 \notin X \wedge v_2 \notin X)))$$
$$\wedge\, (v_3 \in X \vee v_3 \notin X) \wedge \ldots \wedge (v_n \in X \vee v_n \notin X) \Rightarrow c)$$
$$\equiv \quad \exists_V X((((\{v_1, v_2, v_3\} \subseteq X) \vee (\{v_1, v_2\} \subseteq X \wedge v_3 \notin X)$$
$$\vee\, (\{v_1, v_3\} \subseteq X \wedge v_2 \notin X) \vee (\{v_2, v_3\} \subseteq X \wedge v_1 \notin X)$$
$$\vee\, (\{v_1\} \subseteq X \wedge v_2 \notin X \wedge v_3 \notin X)$$
$$\vee\, (\{v_2\} \subseteq X \wedge v_1 \notin X \wedge v_3 \notin X)$$
$$\vee\, (\{v_3\} \subseteq X \wedge v_1 \notin X \wedge v_2 \notin X)$$
$$\vee\, (v_1 \notin X \wedge v_2 \notin X \wedge v_3 \notin X)))$$
$$\wedge\, (v_3 \in X \vee v_3 \notin X) \wedge \ldots \wedge (v_n \in X \vee v_n \notin X) \Rightarrow c)$$
$$\wedge\, (v_4 \in X \vee v_4 \notin X) \wedge \ldots \wedge (v_n \in X \vee v_n \notin X) \Rightarrow c)$$
$$\equiv \quad \exists_V X((\textstyle\bigvee_{i=0}^{2^n}(V_i \subseteq X \wedge \bigwedge_{j\in V - V_i} j \notin X)) \Rightarrow c)$$
$$\equiv \quad \exists_V X(\textstyle\bigwedge_{i=0}^{2^n}(V_i \subseteq X \wedge \bigwedge_{j\in V - V_i} j \notin X \Rightarrow c))$$

5. Analogous to point 4

□

3.6 Summary

This chapter defines monadic second-order formulas that can be used to specify GP 2 graphs. The formula is defined based on standard logic and considering properties of GP 2 graphs and properties that may occur in (conditional) rule schemata.

Unlike (counting) monadic second-order formulas defined in [45], here we use the function card as a cardinality function to express the number of elements in a set of nodes or edges in a graph. By the defined monadic second-order formulas, we show how to express the existence of a directed path (with or without the predicate path).

We also show how to use the formulas to express properties of graphs based on (pre)morphisms that may exist in a rule schema application. We define condition over a graph so that we can express the properties of graphs with respect to the left-hand graph of a rule schema. If we have a (pre)morphism $g : L \to G$ between the left-hand graph L and a host graph G, then the morphism satisfies a condition over L when the replacement graph $\rho_g(G)$ satisfies the condition. In the simplest case, the replacement graph $\rho_g(G)$ can be obtained from G by replacing the identifier $g(i)$ to i, for all node/edge i in L.

This chapter also presents how we can express the specification of a graph L as a condition over L, such that the condition explicitly expresses the structures, marks, and rootedness of nodes and edges in L. We also show how we can turn a condition over a graph into a monadic second-order formula. In addition, we also present some properties of monadic second-order formulas that can be used later to prove other properties of MSO formulas.

Chapter 4

Calculating a strongest liberal postcondition

In this chapter, we describe the intuition of constructing a strongest liberal postcondition over a graph program where the precondition is a monadic second-order formula.

For this section, let us consider a first- or monadic second-order formula. A strongest liberal postcondition is a predicate transformer in the sense of [53] for forward reasoning. It expresses properties that must be satisfied by every graph result from the application of the input rule schema to a graph satisfying the input precondition.

Definition 4.1 (Strongest liberal postcondition over a conditional rule schema)**.** An assertion d is a *liberal postcondition* w.r.t. a conditional rule schema r and a precondition c, if for all host graphs G and H,

$$G \vDash c \text{ and } G \Rightarrow_r H \text{ implies } H \vDash d.$$

A *strongest liberal postcondition* w.r.t. c and r, denoted by $\mathrm{SLP}(c, r)$, is a liberal postcondition w.r.t. c and r that implies every liberal postcondition w.r.t. c and r. □

Our definition of a strongest liberal postcondition is different with the definitions in [26, 53, 54] where they define $\mathrm{SLP}(c, r)$ as a condition such that for every host graph H satisfying the condition, there exists a host graph G satisfying c where $G \Rightarrow_r H$. Lemma 4.2 shows that their definition and ours are equivalent.

Lemma 4.2. Let us consider a rule schema r and a precondition c. Let d be a liberal postcondition w.r.t. r and c. Then d is a strongest liberal postcondition w.r.t. r and c if and only if for every graph H satisfying d, there exists a host graph G satisfying c such that $G \Rightarrow_r H$.

Proof.
(If).
Let H be a host graph satisfying d. Then, there must exists a graph G such that $G \vDash c$ and $G \Rightarrow_r H$. Hence, $H \vDash a$ for any liberal postcondition a from the definition of a liberal postcondition.
(Only if).
Assume that it is not true that for every host graph H, $H \vDash d$ implies there exists a host graph G satisfying c such that $G \Rightarrow_r H$. We show that a graph satisfying d can not imply the graph satisfying any liberal postcondition w.r.t r and c. From the assumption, there exists a host graph H such that every host graph G does not satisfy c or does not derive H by r. In the case of G does not derive H by r, we clearly can not guarantee characteristic of H w.r.t. c. Then for the case where G does not satisfy c but derives H by r, we also can not guarantee the satisfaction of any liberal postcondition a over c and r in H because a is dependent of c. Hence, we can not guarantee that H satisfies all liberal postcondition w.r.t. r and c. □

In [1, 17, 26], a weakest liberal condition is obtained from a given postcondition and a rule by generating a right application condition, then using the obtained condition to generate a left-application condition, to finally obtain a weakest liberal precondition. Similarly, here we use the approach of constructing left and right-application conditions as well, as shown in Figure 4.1.

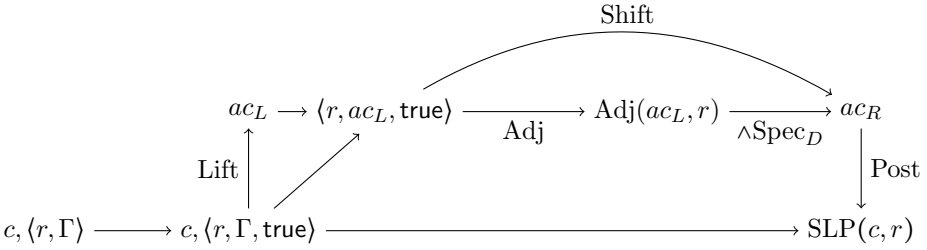


FIGURE 4.1: Constructing $\mathrm{SLP}(c, r)$ for $r = \langle L \leftarrow K \rightarrow R, \Gamma\rangle$

To obtain a strongest liberal postcondition from a given precondition c and conditional rule schema $\langle r, \Gamma\rangle$, we obtain the left-application condition ac_L and the right-application condition ac_R w.r.t. the application of the rule schema to obtain a result graph. Here, we consider the construction for the generalised rule schema instead of the rule schema itself to be able to apply the construction on the inverse of a rule. Note that from the application of generalised rule schema (see Definition 3.18), ac_L must be satisfied by the left morphism, and express the dangling condition, while ac_R must be satisfied by the right morphism. Here, c and the obtained SLP must not contain node/edge identifiers because we should be able to check the satisfaction of the two on any graph. However, ac_L and ac_R may contain node/edge identifiers in L and R (resp.) because they depend on the left and right-hand graph (resp.).

We use transformation Lift to construct ac_L, giving us an updated generalised rule schema, $\langle r, ac_L, \mathsf{true}\rangle$. Next, since we want to construct ac_R which must be satisfied by the right morphism, we need to do adjustment to properties that might be changed due to the rule application. For this, we use transformation Adj. Because we want to express a strongest condition, we should also express properties that must be hold in the resulting graph, based on the rule schema. Hence, we define $\mathrm{Shift}(\langle r, a, b\rangle)$ for conditions a and b as $\mathrm{Adj}(a, r) \wedge \mathrm{Spec}_D$ for $\mathrm{Spec}_D = \mathrm{Spec}(R) \wedge \mathit{dang}(R) \wedge b$, where $\mathit{dang}(R)$ is a condition over R that expresses the satisfaction of the dangling condition w.r.t. r^{-1}. If Shift takes suitable ac_L as the condition a, it will result in a suitable ac_R. Note that ac_R may contain node/edge identifiers of R, while a postcondition should be a closed monadic second-order formula. We then use Post to obtain the strongest liberal postcondition.

Theorem 4.3. For any rule schema r and a precondition c, there exist a transformation Lift, Shift, and Post such that $\mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, r), \mathsf{true}\rangle))$ is a strongest liberal postcondition.

The proof of the above theorem are presented in Chapter 5 and 6. Chapter chap:FOL shows us that there exists such constructions to obtain a strongest liberal postcondition over a first-order formulas, while in Chapter 6, we have a larger class of formulas that is monadic second-order formulas.

In each chapter, we show that the constructed $\mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, r), \mathsf{true}\rangle))$ result in a strongest liberal postcondition by showing that the following properties holds (cf. Figure 4.1):

i) $\rho_g(G) \vDash \mathrm{Lift}(c, r)$ iff $G \vDash c$ and g satisfies the dangling condition

ii) $\rho_g(G) \vDash ac_L$ implies $\rho_{g^*}(H) \vDash \mathrm{AdjLift}(c, r), r$

iii) $\rho_g(G) \vDash \mathrm{Lift}(c, r)$ iff $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(\mathrm{Lift}(c, r), r^{-1}))$

iv) if $H \vDash \mathrm{Var}(\mathrm{Spec}_D)^\gamma$ for some label assignment γ_R, then there exists an injective morphism $R^\gamma \to H$ that satisfies the dangling condition

v) $\rho_{g^*}(H) \vDash \mathrm{AdjLift}(c, r), r$ iff $\rho_{g^*}(H) \vDash \mathrm{Shift}(\langle r, \mathrm{Lift}(c, \langle r, \Gamma, \mathsf{true}\rangle\rangle))$

vi) $\rho_{g^*}(H) \vDash \mathrm{Shift}(\langle r, \mathrm{Lift}(c, \langle r, \Gamma, \mathsf{true}\rangle\rangle))$ iff $H \vDash \mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, r), \mathsf{true}\rangle))$

Table 4.1 shows where we can find the proof of the above properties in Chapter 5 and 6. We prove properties i). ii), iii), and v) in both chapter, but for point iv) and vi), we only prove them in Chapter 5 and reuse the Lemma/Proposition in Chapter 6. This is because we use the same definition for Post, Spec, and Dang for both first-order and monadic second-order formulas.

TABLE 4.1: Properties to support the proof of Theorem 4.3

Property	**in Chapter 5**	**in Chapter 6**
i)	Proposition 5.8	Proposition 6.7
ii)	Lemma 5.10	Lemma 6.11
iii)	Lemma 5.11	Lemma 6.12
iv)	Corollary 5.2	Corollary 5.2
v)	Proposition 5.13	Proposition 6.14
vi)	Proposition 5.15	Proposition 5.15

In addition to the five properties, we present Theorem 5.16 and Theorem 6.15 to prove that the constructed $\mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, r), \mathsf{true}\rangle))$ are indeed a strongest liberal postcondition. Both theorems then are the main proof of Theorem 4.3 we have above.

Chapter 5

A strongest liberal postcondition for first-order formulas

In this chapter, we introduce a way to construct a strongest liberal postcondition over a graph program where the precondition is a first-order formula. Here, conditions refer to first-order formulas, possibly with node or edge constant (i.e. conditions over a graph. However, pre- and postcondition are limited to closed first-order formula (without node/edge constant)

5.1 Construction of a strongest liberal postcondition

To construct $\mathrm{SLP}(c, r)$, we use the generalised version of r to open a possibility of constructing a strongest liberal postcondition over the inverse of a rule schema. Since a rule schema has some restriction on the existence of variables and `any`-mark, a rule schema may not be invertible. By using the generalised version of a rule schema, we omit this limitation so that the generalised version of the inverse of a rule schema is also a generalised rule schema so that we can use the construction for an inversed rule schema as well.

In this thesis, $\mathrm{SLP}(c, r)$ is obtained by defining transformations Lift, Shift, and Post. The transformation Lift transforms the given condition c into a left-application condition w.r.t. the given rule schema r. Then, we transform the left-application condition to right-application condition by transformation Shift. Finally, the transformation Post transforms the right-application condition to a strongest liberal postcondition (see Figure 4.1). Recall the generalised rule schema application (See Definition 3.18). For a rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and precondition c, the obtained left and right-application condition should satisfies the properties in the application as well, as can be seen in Figure 5.1.

For a conditional rule schema $\langle r, \Gamma \rangle$ with rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and a precondition c, when a graph G satisfying c and there exists a label assignment α_L such that $G \Rightarrow_{r^\alpha, g} H$

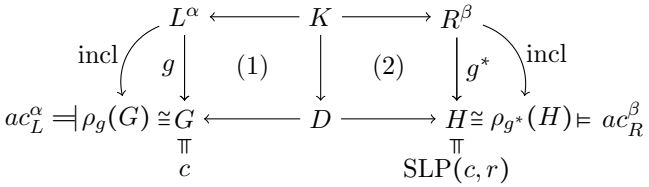
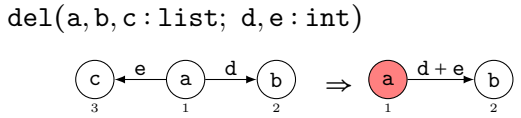


FIGURE 5.1: Generalised rule schema application and strongest liberal postcondition




```
del (a,b,c:list;d,e:int;)
[ | (1,a) (2,b) (3,c)
  | (e1,1,2,d) (e2,1,3,e)]
=>
[ | (1#red,a) (2,b)
  | (e1,1,2,d+e)]
interface = {1,2}
where d ≥ e
```

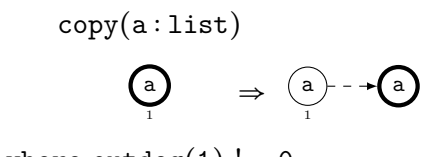
FIGURE 5.2: GP 2 conditional rule schema $\texttt{del} = \langle r_1, \Gamma_1 \rangle$

for some host graph H and injective graph morphism $g : L^\alpha \to G$, $ac_L^\alpha = (\text{Lift}_{\mathsf{FO}}(c, r^\vee))^\alpha$ should be satisfied by G w.r.t. g. The replacement graph $\rho_g(G)$ should satisfies ac_L which means ac_L should consist of the precondition c, rule schema condition Γ, and the dangling condition.

From the definition of rule schema application (see Definition 2.22), we know that $G \Rightarrow_{r^\alpha, g} H$ with injective graph morphism $g : L^\alpha \to G$ and label assignment α_L obviously assert the existence of injective morphism $g^* : R^\beta \to H$ for some label assignment β_R such that $\alpha_L(i) = \beta_R(i)$ for every common element i (see Figure 3.2). The graph replacement $\rho_{g^*}(H)$ then should satisfy $ac_R^\beta = (\text{Shift}_{\mathsf{FO}}(\langle r, ac_L, \mathsf{true} \rangle))^\beta$. The graph condition ac_R should describe the elements of the image of the comatch and some properties of c that are still relevant after the rule schema application.

Basically, ac_R is already a strongest property that must be satisfied by a resulting graph. However, it has node/edge constants so that we need to change it into a closed formula so that we finally obtain a strongest liberal postcondition. This part is done by the transformation Post.

To give a better idea of the transformations we define in this chapter, we show examples after each definition. We use the conditional rule schemata $\texttt{del} = \langle r_1, \Gamma_1 \rangle$ of Figure 5.2 and $\texttt{copy} = \langle r_2, \Gamma_2 \rangle$ of Figure 5.3 and the preconditions $q_1 = \neg \exists_{\mathsf{e}} \mathsf{x}(\mathsf{m_v}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none})$ and $q_2 = \exists_{\mathsf{v}} \mathsf{x}(\neg \mathsf{root}(\mathsf{x}))$ as running examples. We denote by Γ_1 and Γ_2 the GP 2 rule schema conditions $\mathsf{d} \geq \mathsf{e}$ and $\mathsf{outdeg}(1) \neq 0$ respectively. Also, we denote by r_1 and r_2 the rule schema (without condition) of `del` and `copy` respectively. Based on Definition 3.19, we have the generalised version of rule schemata: $\texttt{del}^\vee = \langle r_1, \Gamma_1^\vee, \mathsf{true} \rangle$ and $\texttt{copy}^\vee = \langle r_2, \Gamma_2^\vee, \mathsf{true} \rangle$.




```
copy (a : list;)
[ | (1(R), a) | ]
=>
[ | (1, a) (2(R), a)
  | (e1, 1, 2, empty#dashed)]
interface = {1}
where outdeg(1) != 0
```

FIGURE 5.3: GP 2 conditional rule schema copy = $\langle r_2, \Gamma_2 \rangle$

5.2 The dangling condition

The dangling condition must be satisfied by an injective morphism g if $G \Rightarrow_{r,g} H$ for some rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and host graphs G, H. Since we want to express properties of $\rho_g(G)$ where such derivation exists, we need to express the dangling condition as a condition over the left-hand graph.

Recall the dangling condition from Definition 2.8. $\rho_g(G)$ satisfies the dangling condition if every node $v \in L - K$ is not incident to any edge outside L. This means that the indegree and outdegree of every node $v \in L - K$ in L represent the indegree and outdegree of v in G as well.

Definition 5.1 (Condition Dang)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ where $\{v_1, \cdots, v_n\}$ is the set of all nodes in $L - K$. Let $indeg_L(v)$ and $outdeg_L(v)$ denotes the indegree and outdegree of v in L, respectively. The condition Dang(r) is defined as:

1. if $V_L - V_K = \varnothing$ then Dang(r) = true
2. if $V_L - V_K \neq \varnothing$ then

$$\text{Dang}(r) = \bigwedge_{i=1}^{n} \mathsf{indeg}(\mathsf{v_i}) = indeg_L(v_i) \wedge \mathsf{outdeg}(\mathsf{v_i}) = outdeg_L(v_i)$$

□

Example 5.1 (Condition Dang)**.** Let us consider r_1 and r_2 of Figure 5.2 and Figure 5.3. For r_1, node 3 gets deleted by the rule, so that the degree of node 3 in the match must be the same as the degree of node 3. However, for r_2, there is no node gets deleted so that the dangling condition always true. Hence,

1. Dang(r_1) = indeg(3) = 1 ∧ outdeg(3) = 0
2. Dang(r_2) = true

Observation 5.1. Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. Let G be a host graph and $g : L \rightarrow G$ be a premorphism. The dangling condition is satisfied if and only if $\rho_g(G) \vDash \mathrm{Dang}(r)$.

Proof. From the definition of the dangling condition (see Definition 2.8), the dangling condition is satisfied when no edge in $G - g(L)$ is incident to any node in $g(L - K)$. By the definition of replacement graph (see Definition 3.16), we can see that $G - g(L)$ is equivalent to $\rho_g(G) - L$ because G and $\rho_g(G)$ only different in node/edge identifiers of the match, which now gets deleted. Then, evaluating the construct of $g(L - K)$ in G w.r.t. g is the same as evaluating the $L - K$ in $\rho_g(G)$. Hence, the dangling condition is satisfied iff no edge in $\rho_g(G) - L$ incident to any node in $L - K$, which means all nodes in $L - K$ only incident to edges in L. Hence, $\mathrm{Dang}(r)$ is true. □

Corollary 5.2. For every rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and every graph H,
if $H \vDash \mathrm{Var}(\mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1}))^{\alpha}$ for some label assignment α_R, then there exists an injective morphism $R^{\gamma} \rightarrow H$ that satisfies the dangling condition.

Proof. From Lemma 3.27, we know that $H \vDash \mathrm{Var}(\mathrm{Spec}(R^{\alpha}))$ for some label assignment α_R implies the existence of an injective morphism $g^* : R^{\alpha} \rightarrow H$. Since Observation 5.1 asserts the satisfaction of $\mathrm{Dang}(r^{-}1)$ implies the satisfaction of the dangling condition w.r.t. r^{-1}, then $H \vDash \mathrm{Var}(\mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1}))^{\alpha}$ implies that the morphism $g^* : R^{\alpha} \rightarrow H$ satisfies the dangling condition. □

5.3 From precondition to left-application condition

Now, we start with transforming a precondition c to a left-application condition with respect to a generalised rule $w = \langle r, ac_L, ac_r \rangle$. Intuitively, the transformation is done by:

1. Find all possibilities of variables in c representing nodes/edges in the images of a match and form a disjunction from all possibilities, denoted by $\mathrm{Split}_{\mathsf{FO}}(c, r)$;
2. Express the dangling condition as a condition over L, denoted by $\mathrm{Dang}(r)$;
3. Evaluate terms and Boolean expression we can evaluate in $\mathrm{Split}_{\mathsf{FO}}(c, r)$, $\mathrm{Dang}(r)$, and Γ with respect to the left-hand graph of the given rule, then form a conjunction from the result of evaluation, and simplify the conjunction.

A possibility of variables in c representing nodes/edges in the images of a match mentioned above refers to how variables in c can represent node or edge constants in the replacement

of the input graph. A simple example would be for a precondition $c = \exists_{\mathsf{v}}\mathsf{x}(c_1)$ for some FO formula c_1 containing a free variable x, c holds on a host graph G if there exists a node v in G such that c_1^{α} where $\alpha(x) = v$ is true in G. The node v can be any node in G. In the replacement graph of G, v can be any node in the left-hand graph of the rule schema, or any node outside it. $\mathrm{Split}_{\mathsf{FO}}(c, r)$ is obtained from the disjunction of all these possibilities.

Definition 5.3 (Transformation $\mathrm{Split}_{\mathsf{FO}}$)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. where $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let c be a condition over L sharing no variables with r (note that it is always possible to replace the label variables in c with new variables that are distinct from variables in r). We define the condition $\mathrm{Split}_{\mathsf{FO}}(c, r)$ over L inductively as follows:

- Base case.

  If c is true, false, or a predicate $\mathsf{int(t)}, \mathsf{char(t)}, \mathsf{string(t)}, \mathsf{atom(t)}, \mathsf{root(t)}$ for some list variable t, or in the form $\mathsf{t_1} \ominus \mathsf{t_2}$ for $\ominus \in \{=, \neq, <, \leq, >, \geq\}$ and some terms $\mathsf{t_1}, \mathsf{t_2}$,

  $$\mathrm{Split}_{\mathsf{FO}}(c, r) = c$$

- Inductive case.

  Let c_1 and c_2 be conditions over L.

  1) $\mathrm{Split}_{\mathsf{FO}}(c_1 \vee c_2, r) = \mathrm{Split}_{\mathsf{FO}}(c_1, r) \vee \mathrm{Split}_{\mathsf{FO}}(c_2, r)$,

  2) $\mathrm{Split}_{\mathsf{FO}}(c_1 \wedge c_2, r) = \mathrm{Split}_{\mathsf{FO}}(c_1, r) \wedge \mathrm{Split}_{\mathsf{FO}}(c_2, r)$,

  3) $\mathrm{Split}_{\mathsf{FO}}(\neg c_1, r) = \neg\, \mathrm{Split}_{\mathsf{FO}}(c_1, r)$,

  4) $\mathrm{Split}_{\mathsf{FO}}(\exists_{\mathsf{v}}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \mathrm{Split}_{\mathsf{FO}}(c_1^{[x \mapsto v_i]}, r)) \vee \exists_{\mathsf{v}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1, r)$,

  5) $\mathrm{Split}_{\mathsf{FO}}(\exists_{\mathsf{e}}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}_{\mathsf{FO}}(c_1^{[x \mapsto e_i]}, r)) \vee \exists_{\mathsf{e}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{inc}(c_1, r))$,

     where

     $$\begin{aligned}\mathrm{inc}(c_1, r) = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} (&\bigvee_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s(x)} = \mathsf{v_i} \wedge \mathsf{t(x)} = \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}, \mathsf{t(x)} \mapsto \mathsf{v_j}]}, r)) \\ &\vee (\mathsf{s(x)} = \mathsf{i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}]}, r)) \\ &\vee (\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_j} \wedge \mathsf{t(x)} = \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1^{[\mathsf{t(x)} \mapsto \mathsf{v_i}]}, r)) \\ \vee (&\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1, r))\end{aligned}$$

  6) $\mathrm{Split}_{\mathsf{FO}}(\exists_{\mathsf{l}}\mathsf{x}(c_1), r) = \exists_{\mathsf{l}}\mathsf{x}(\mathrm{Split}_{\mathsf{FO}}(c_1, r))$

where $c^{[a \mapsto b]}$ for a variable a and constant b represents the condition c after the replacement of all occurrence of a with b. Similarly, $c^{[d \mapsto b]}$ for $d \in \{\mathsf{s(x)}, \mathsf{t(x)}\}$ is also a replacement d with b. □

As can be seen in the definition above, $\mathrm{Split}_{\mathsf{FO}}$ of an edge quantifier is not as simple as $\mathrm{Split}_{\mathsf{FO}}$ of a node quantifier. For an edge variable x in a precondition, x may represent any edge in G. Moreover, the term $\mathsf{s(x)}$ or $\mathsf{t(x)}$ may represent a node in the image of the match. Hence, we need to check these possibilities as well. However, if the precondition does not contain a term $\mathsf{s(x)}$ or $\mathsf{t(x)}$ for some edge variable x, we do not need to consider nodes that can be represented by the functions.

Observation 5.2. Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ where $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let $c = \exists_{\mathsf{e}} x(c_1)$ be a condition over L. Then, the following holds:

1. If c_1 does not contain the term $\mathsf{s(x)}$,
   $\mathrm{inc}(c_1, r) = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{t(x)} = \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1^{[\mathsf{t(x)} \mapsto \mathsf{v_i}]}, r)) \vee \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{t(x)} \neq \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1, r))$
2. If c_1 does not contain the term $\mathsf{t(x)}$,
   $\mathrm{inc}(c_1, r) = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{s(x)} = \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}]}, r)) \vee \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{s(x)} \neq \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{FO}}(c_1, r))$
3. If c_1 does not contain the terms $\mathsf{s(x)}$ and $\mathsf{t(x)}$,
   $\mathrm{inc}(c_1, r) = \mathrm{Split}_{\mathsf{FO}}(c_1, r)$

Proof. Let us observe each point of the Observation:

1. If c_1 does not contain the term $\mathsf{s(x)}$, then for any i, j, $c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}, \mathsf{t(x)} \mapsto \mathsf{v_j}]} = c_1^{[\mathsf{t(x)} \mapsto \mathsf{v_j}]}$, and $c_1^{[\mathsf{s(x)}]} = c_1$. The first and the third line of $\mathrm{inc}(c_1, r)$ is the disjunction of all possibilities of $(t(x))$ is one of nodes in L while the second and forth line is about $(t(x))$ is outside the match.
2. Analogously to above.
3. If c_1 does not contain the terms $\mathsf{s(x)}$ and $\mathsf{t(x)}$, we know that $c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_i}, \mathsf{t(x)} \mapsto \mathsf{v_j}]} = c_1^{[\mathsf{t(x)} \mapsto \mathsf{v_j}]} = c_1^{[\mathsf{s(x)} \mapsto \mathsf{v_j}]} = c_1$ because we do not have nothing to substitute in c_1.

□

Example 5.2 (Transformation $\mathrm{Split}_{\mathsf{FO}}$)**.** Let us consider q_1, q_2 we defined in Section 5.1 and r_1, r_2 of Figure 5.2 and Figure 5.3. Then based on Definition 5.3,

$$
\begin{aligned}
\mathrm{Split}_{\mathsf{FO}}(q_1, r_1) &= \neg\, \mathrm{Split}_{\mathsf{FO}}(\exists_{\mathsf{e}} \mathsf{x}(\mathsf{m_v(s(x))} \neq \mathsf{none}), r_1) \\
&= \neg(\mathsf{m_v(s(e1))} \neq \mathsf{none} \vee \mathsf{m_v(s(e2))} \neq \mathsf{none} \vee \\
&\quad \exists_{\mathsf{e}} \mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge ((\mathsf{s(x)} = 1 \wedge \mathsf{m_v}(1) \neq \mathsf{none}) \\
&\qquad \vee (\mathsf{s(x)} = 2 \wedge \mathsf{m_v}(2) \neq \mathsf{none}) \\
&\qquad \vee (\mathsf{s(x)} = 3 \wedge \mathsf{m_v}(3) \neq \mathsf{none}) \\
&\qquad \vee (\mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{s(x)} \neq 3 \\
&\qquad\quad \wedge \mathsf{m_v(s(x))} \neq \mathsf{none})))) \\
\mathrm{Split}_{\mathsf{FO}}(q_2, r_2) &= \neg\mathsf{root}(1) \vee \exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \neg\mathsf{root(x)})
\end{aligned}
$$

Since $\mathrm{Split}_{\mathsf{FO}}(c, r)$ only disjunct all possibilities of nodes and edges that can be represented by node and edge variables in c, it should not change the semantic of c. However, we transform

a condition c to a condition over L such that we may not be able to check satisfaction of $\mathrm{Split}_{\mathsf{FO}}(c,r)$ in G. However, we can always check its satisfaction in $\rho_g(G)$ for some premorphism $g : L \to G$.

Lemma 5.4. Let us consider a condition c and an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, sharing no variables with c. For a host graph G, let $g : L \to G$ be a premorphism. Then,

$$G \vDash c \text{ if and only if } \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c,r).$$

Proof. Here, we prove the lemma inductively. The texts above the symbol $\Leftrightarrow$ bellow refer to lemmas that imply the associated implication, e.g. L4 refers to Lemma 4.

(Base case).

$$G \vDash c \overset{\text{L3.28}}{\Leftrightarrow} \rho_g(G) \vDash c \Leftrightarrow \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c,r)$$

(Inductive case).

Assuming that for some conditions c_1 and c_2 over L, the lemma holds.

1) $G \vDash c_1 \vee c_2 \overset{\text{F3.1}}{\Leftrightarrow} G \vDash c_1 \vee G \vDash c_2 \Leftrightarrow \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c_1,r) \vee \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c_2,r) \overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c_1,r) \vee \mathrm{Split}_{\mathsf{FO}}(c_2,r)$

2) $G \vDash c_1 \wedge c_2 \overset{\text{F3.1}}{\Leftrightarrow} G \vDash^{\alpha} c_1 \wedge G \vDash^{\alpha} c_2$ for some assignment α $\Leftrightarrow \rho_g(G) \vDash^{\beta} \mathrm{Split}_{\mathsf{FO}}(c_1,r) \vee \rho_g(G) \vDash^{\beta} \mathrm{Split}_{\mathsf{FO}}(c_2,r)$ where $\beta(x) = \alpha(x)$ if $x \notin V_L$; $\beta(x) = g^{-1}(\alpha(x))$ otherwise $\overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c_1,r) \vee \mathrm{Split}_{\mathsf{FO}}(c_2,r)$

3) $G \vDash \neg c_1 \overset{\text{F3.1}}{\Leftrightarrow} \neg(G \vDash^{\alpha} c_1)$ for some assignment α $\Leftrightarrow \neg(\rho_g(G) \vDash^{\beta} \mathrm{Split}_{\mathsf{FO}}(c_1,r))$ where $\beta(x) = \alpha(x)$ if $x \notin V_L$; $\beta(x) = g^{-1}(\alpha(x))$ otherwise $\overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \neg \mathrm{Split}_{\mathsf{FO}}(c_1,r)$

4) $G \vDash \exists_{\mathsf{v}} \mathsf{x}(c_1) \overset{\text{L3.30}}{\Leftrightarrow} G \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{n}} c_1^{[\mathsf{x} \mapsto \mathsf{v}_\mathsf{i}]} \vee \exists_{\mathsf{v}} x(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v}_\mathsf{i} \wedge c_1) \Leftrightarrow \rho_g(G) \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{n}} \mathrm{Split}_{\mathsf{FO}}(\mathsf{c}_1^{[\mathsf{x} \mapsto \mathsf{v}_\mathsf{i}]}, \mathsf{r}) \vee \exists_{\mathsf{v}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v}_\mathsf{i} \wedge \mathrm{Split}_{\mathsf{FO}}(\mathsf{c}_1, \mathsf{r}))$

5) $G \vDash \exists_{\mathsf{e}} \mathsf{x}(c_1) \overset{\text{L3.30}}{\Leftrightarrow} G \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{m}} c_1^{[\mathsf{x} \mapsto \mathsf{e}_\mathsf{i}]} \vee \exists_{\mathsf{v}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e}_\mathsf{i} \wedge \mathsf{c}_1) \overset{\text{L3.30}}{\Leftrightarrow} \rho_g(G) \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}_{\mathsf{FO}}(\mathsf{c}_1^{[\mathsf{x} \mapsto \mathsf{e}_\mathsf{i}]}, \mathsf{r}) \vee \exists_{\mathsf{v}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{v}_\mathsf{i} \wedge \mathrm{Split}_{\mathsf{FO}}(\mathsf{c}_1, \mathsf{r})) \overset{\text{L3.30}}{\Leftrightarrow} \rho_g(G) \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}_{\mathsf{FO}}(\mathsf{c}_1^{[\mathsf{x} \mapsto \mathsf{e}_\mathsf{i}]}, \mathsf{r}) \vee \exists_{\mathsf{v}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{v}_\mathsf{i} \wedge \mathrm{inc}(\mathsf{c}_1, \mathsf{r}))$

6) $G \vDash \exists_{\mathsf{l}} \mathsf{x}(c_1) \overset{\text{F3.1}}{\Leftrightarrow} G \vDash c_1 \Leftrightarrow \rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c_1,r) \overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \exists_{\mathsf{l}} \mathsf{x}(\mathrm{Split}_{\mathsf{FO}}(c_1,r))$

□

After splitting the precondition into all possibilities of representations, we check the value of some functions and Boolean operators to check if any possibility violates the precondition such that we can omit the possibility.

Definition 5.5 (Valuation of c)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, a condition c over L, a host graph G, and premoprhism $g : L \rightarrow G$. Let c shares no variable with L unless c is a rule schema condition. Let also $F = \{\mathsf{s}, \mathsf{t}, \mathsf{l_v}, \mathsf{l_e}, \mathsf{m_v}, \mathsf{m_e}, \mathsf{indeg}, \mathsf{outdeg}, \mathsf{length}$ be the set of function syntax. Let also $y \oplus_L z$ for $\oplus \in \{+, -, *, /, :, .\}$ and $y, z \in \mathbb{L}$ denotes the value of $y \oplus z$ as described in Section 3.3, and $f_L(z)$ for a constant z and $f \in F$ denotes the value of $f(y)$ in L. *Valuation of c w.r.t. r*, written $\mathrm{Val}_{\mathsf{FO}}(c, r)$, is constructed by applying the following steps to c:

1. Obtain c' by changing every term x in c with $T(x)$, where
   (a) If x is a constant or variable, $T(x) = x$
   (b) If $x = f(y)$ for $f \in F$,

$$T(x) = \begin{cases} f_L(y) & \text{if } f \in F \backslash \{\mathsf{indeg}, \mathsf{outdeg}\} \text{ and } y \text{ is a constant} \\ & \quad \text{or } f \in \{\mathsf{indeg}, \mathsf{outdeg}\} \text{ and } y \in V_L - V_K \\ f_L(T(x)) & \text{if } f \in \{\mathsf{l_v}, \mathsf{m_v}\},\ (y = \mathsf{s(e)} \text{ or } y = \mathsf{t(e)}), e \in E_L \\ & \quad \text{or } f \in \{\mathsf{indeg}, \mathsf{outdeg}\} \text{ and } T(y) \in V_L - V_K \\ incon(T(y)) + f_L(T(y)) & \text{if } f = \mathsf{indeg} \text{ and } y \in V_K \\ outcon(T(y)) + f_L(T(y)) & \text{if } f = \mathsf{outdeg} \text{ and } y \in V_K \\ f(y) & \text{otherwise} \end{cases}$$

   (c) If $x \oplus z$ for $\oplus \in \{+, -, /, *, :, .\}$,

$$T(x) = \begin{cases} y \oplus_L z & \text{if } y, z \in \mathbb{L} \\ T(y) \oplus T(z) & \text{if } T(y) =\notin \mathbb{L} \text{ or } T(z) =\notin \mathbb{L} \\ T(T(y) \oplus T(z)) & \text{otherwise} \end{cases}$$

2. Obtain c" by replacing predicates and Boolean operators x in c' with $B(x)$, where

$$B(x) = \begin{cases} y \otimes_{\mathbb{B}} z & \text{if } x = y \otimes z \text{ for } \otimes \in \{=, \neq, \leq, \geq\} \text{ and constants } y, z \\ \mathsf{true} & \text{if } x = \mathsf{root(v)} \text{ for } v \in r_L \\ \mathsf{false} & \text{if } x = \mathsf{root(v)} \text{ for } v \notin r_L \\ x & \text{otherwise} \end{cases}$$

3. Simplify c" such that there are no subformulas in the form $\neg\,\mathsf{true}, \neg(\neg\, a)$ $\neg(a \vee b)$, $\neg(a \wedge b)$ for some conditions a, b. We can always simplify them to $\mathsf{false}, a, \neg a \wedge \neg b, \neg a \vee \neg b$ respectively.

□

Intuitively, $\mathrm{Val}_{\mathsf{FO}}$ gives some terms with node/edge constants their value in L. Recall that if there exists injective morphism $g : L^{\alpha} \to G$ for some label assignment α_L, then there must be an inclusion $L^{\alpha} \to \rho_g(G)$. This should assert that the value of terms we valuate in L is equal to their value in $\rho_g(G)$.

Example 5.3 (Valuation of a graph condition)**.** For rule schemata r_1 and r_2, $\mathrm{Split}_{\mathsf{FO}}(q_1, r_1)$ and $\mathrm{Split}_{\mathsf{FO}}(q_2, r_2)$ we obtained in Example 5.2, also rule application conditions Γ_1 of r_1 and Γ_2 of r_2,

1. $\mathrm{Val}_{\mathsf{FO}}(\mathrm{Split}_{\mathsf{FO}}(q_1, r_1), r_1)$
   $= \neg(\mathsf{none} \neq \mathsf{none} \vee \mathsf{none} \neq \mathsf{none} \vee$
   $\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge ((\mathsf{s}(\mathsf{x}) = 1 \wedge \mathsf{none} \neq \mathsf{none})$
   $\vee (\mathsf{s}(\mathsf{x}) = 2 \wedge \mathsf{none} \neq \mathsf{none})$
   $\vee (\mathsf{s}(\mathsf{x}) = 3 \wedge \mathsf{none} \neq \mathsf{none})$
   $\vee (\mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{s}(\mathsf{x}) \neq 3 \wedge \mathsf{m_v}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none}))))$
   $\equiv \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge \mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{s}(\mathsf{x}) \neq 3 \wedge \mathsf{m_v}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none})$
   Here, we replace the terms $\mathsf{s(e1)}, \mathsf{s(e2)}$ with node constant 1, then replace $\mathsf{m_v}(1)$, $\mathsf{m_v}(2)$, $\mathsf{m_v}(3)$ with none. Then, we simplify the resulting condition by evaluating $\mathsf{none} \neq \mathsf{none}$ which is equivalent to false.

2. $\mathrm{Val}_{\mathsf{FO}}(\mathrm{Split}_{\mathsf{FO}}(q_2, r_2), r_1) = \mathsf{false} \vee \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \neg\mathsf{root}(\mathsf{x}))$
   $\equiv \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \neg\mathsf{root}(\mathsf{x}))$
   Here, we substitute false for $\neg\mathsf{root}(1)$ since the node 1 in L is a rooted node.

3. $\mathrm{Val}_{\mathsf{FO}}(\Gamma_1, r_1) = \mathsf{d} \geq \mathsf{e}$
   For this case, we change nothing.

4. $\mathrm{Val}_{\mathsf{FO}}(\Gamma_2, r_2) = \mathsf{outcon}(1) \neq 0$
   In this case, we change $\mathsf{outdeg}(1)$ with $\mathsf{outcon}(1) + 0$ because the outdegree of node 1 in L is 0.

Lemma 5.6. Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, a host graph G, and an injectiva morphism $g : L^{\alpha} \to G$ for a label assignment α_L. For a graph condition c,

$$\rho_g(G) \vDash c \text{ if and only if } \rho_g(G) \vDash (\mathrm{Val}_{\mathsf{FO}}(c, r))^{\alpha}$$

Proof. Let us consider the construction of $\mathrm{Val}_{\mathsf{FO}}(c)$ step by step. In step 1, we change terms x in c with $T(x)$. Here, we change functions $\mathsf{s(e)}, \mathsf{t(e)}, \mathsf{l_v(v)}, \mathsf{m_v(v)}$, $\mathsf{l_e(e)}, \mathsf{m_e(e)}, \mathsf{l_v(s(e))}, \mathsf{l_v(t(e))}, \mathsf{m_v(s(e))}, \mathsf{m_v(t(e))}$ for $e \in E_L$ and $v \in V_L$ with their values in L. Since $L^{\alpha} \to \rho_g(G)$ is an inclusion, then $s_L(e) = s_{\rho_g(G)}(e)$ and $t_L(e) = t_{\rho_g(G)}(e)$. Also, $(l^V_L(v))^{\alpha} = l^V_{\rho_g(G)}(v)$, $(l^E_L(e))^{\alpha} = l^E_{\rho_g(G)}(e)$, $(m^V_L(v))^{\alpha} = m^V_{\rho_g(G)}(v)$, and $(m^E_L(e))^{\alpha} =$

$m^E_{\rho_g(G)}(e)$ for all $v \in V_G$ and $e \in E_G$ such that the replacement does not change the satisfaction of c in $\rho_g(G)$. Then for function $\mathsf{indeg(v)}$ for $v \in V_L - V_K$, we change it to $indeg_L(x)$ due to the dangling condition, and for $v \in V_K$, we change it to $incon(v) + indeg_L(v)$ which is equivalent to $indeg_G(v) = indeg_{\rho_g(G)}(v)$ because $incon(v) = indeg_G(v) - indeg_L(v)$ (and analogously for $\mathsf{outdeg(v)}$). In step 2, changing Boolean operators whose arguments are constants to their Boolean value clearly does not change the satisfaction in $\rho_g(G)$. Also, by the definition of morphism, $p_L(v) = p_{\rho_g(G)}(v)$ for all $v \in V_L$ so that the Boolean value of $\mathsf{root(v)}$ in L is equivalent to the Boolean value of $\mathsf{root(v)}$ in $\rho_g(G)$. Finally, in step 3, simplification clearly does not change satisfaction. □

Finally, we define the transformation Lift, which takes a precondition and a generalised rule schema as an input and gives a left-application condition as an output. The output should express the precondition, the dangling condition, and the existing left-application condition of the given generalised rule schema.

Definition 5.7 (Transformation Lift)**.** Let us consider a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$. Let c be a precondition. A left application condition w.r.t. c and w, denoted by $\mathrm{Lift}_{\mathsf{FO}}(c, w)$, is the condition over L:

$$\mathrm{Lift}_{\mathsf{FO}}(c, w) = \mathrm{Val}_{\mathsf{FO}}(\mathrm{Split}_{\mathsf{FO}}(c \wedge ac_L, r) \wedge \mathrm{Dang}(r), r).$$

□

Example 5.4 (Transformation Lift)**.** Let us consider $q_1, q_2, \mathsf{del}^\vee$, and $\mathsf{copy}^\vee$ we defined in Section 5.1. Based on Definition 5.7, also from Example 5.1, 5.2, and 5.3, we have

1. $\mathrm{Lift}_{\mathsf{FO}}(q_1, \mathsf{del}^\vee)$
   $= \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge \mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{s(x)} \neq 3 \wedge \mathsf{m_v(s(x))} \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e}$
2. $\mathrm{Lift}_{\mathsf{FO}}(q_2, \mathsf{copy}^\vee) = \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \neg\mathsf{root(x)}) \wedge \mathsf{outcon}(1) \neq 0 \wedge \mathsf{true}$
   $\equiv \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \neg\mathsf{root(x)}) \wedge \mathsf{outcon}(1) \neq 0$

Proposition 5.8 (Left-application condition)**.** Let us consider a host graph G and a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$. Let c be a precondition and α_L be a label assignment such that there exists an injective morphism $g : L^\alpha \rightarrow G$.

$$G \vDash c \text{ and } G \Rightarrow_{r^\alpha, g} H \text{ for some host graph } H \text{ iff } \rho_g(G) \vDash (\mathrm{Lift}_{\mathsf{FO}}(c, w))^\alpha$$

Proof. From Lemma 5.6, we know that $\rho_g(G) \vDash \mathrm{Val}_{\mathsf{FO}}(\mathrm{Split}_{\mathsf{FO}}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r), r)^\alpha$ iff $\rho_g(G) \vDash (\mathrm{Split}_{\mathsf{FO}}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r))^\alpha$. From 5.4, we know that $\rho_g(G) \vDash \mathrm{Split}_{\mathsf{FO}}(c, r)$ iff

$G \vDash c$. Also, from 5.1, we get that $\rho_g(G) \vDash \mathrm{Dang}(r)$ iff g satisfies the dangling condition. We also know that $\rho_g(G) \vDash (\Gamma^\vee)^\alpha$ iff $\Gamma^{\alpha,g}$ is true in G because the change of symbols does not change the semantics of the condition. From the definition of rule schema application, the satisfaction of the dangling condition and $\Gamma^{\alpha,g}$ is true in G iff $G \Rightarrow_{r^\alpha,g} H$ for some host graph H. □

Recall the construction of $\mathrm{Split}_{\mathsf{FO}}(c, r)$ for a precondition c and an unrestricted rule schema r. A node/edge quantifier is preserved in the result of the transformation with additional restriction about x not representing any node/edge in L. Hence in the resulting condition over L from transformation Lift, every node/edge variable should not represent any node/edge in L.

Observation 5.3. Let us consider a host graph G and a generalised rule $w = \langle r^\alpha, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, assignment α, and a precondition c. For every node/edge variable x in $\mathrm{Lift}_{\mathsf{FO}}(c, w)$, x does not represent any node/edge in L.

Proof. Here we show that for every node/edge variable x, there exists an existential quantifier over x such that there exists constraint $\bigwedge_{i \in V_L} x \neq i$ or $\bigwedge_{i \in E_L} x \neq i$ inside the quantifier.

$\mathrm{Lift}_{\mathsf{FO}}(c, w)$ is a conjunction of $\mathrm{Val}_{\mathsf{FO}}(\mathrm{Split}_{\mathsf{FO}}(c, r), r)$, $\mathrm{Dang}(r)$, and $\mathrm{Val}_{\mathsf{FO}}(\Gamma, r)$. The transformation $\mathrm{Val}_{\mathsf{FO}}$ clearly does not remove or change subformulas in the form $x \neq i$ and does not add any new node/edge variable. Hence, we just need to show that for every node/edge variable x in $\mathrm{Split}_{\mathsf{FO}}(c, r)$, $\mathrm{Dang}(r)$, and Γ, there exists constraint $\bigwedge_{i \in V_L} x \neq i$ or $\bigwedge_{i \in E_L} x \neq i$.

We can see from Definition 2.20 that Γ does not have node and edge variable from its syntax. For $\mathrm{Dang}(r)$, it clearly only has one edge variable and there exists constraint $\bigwedge_{i \in E_L} x \neq i$ inside the existential quantifier for the variable. Finally for $\mathrm{Split}_{\mathsf{FO}}(c, r)$, since c is a closed formula, every node/edge variable must be bounded by existential quantifier, such that from Definition 5.3, the variable must be bounded by existential quantifier with constraint $\bigwedge_{i \in V_L} x \neq i$ or $\bigwedge_{i \in E_L} x \neq i$ inside. □

5.4 From left to right-application condition

To obtain a right-application condition from a left-application condition, we need to consider what properties could be different in the initial and the result graphs. Recall that in constructing a left-application condition, we evaluate all functions with a node/edge constant argument and change them with constant, including the constant $\mathsf{incon}(\mathsf{v})$ and $\mathsf{outcon}(\mathsf{v})$ when evaluating $\mathsf{indeg}(\mathsf{v})$ and $\mathsf{outdeg}(\mathsf{v})$ for node v in the interface. In the result graph H, $indeg_H(v)$ is clearly equal to $incon(v) + indeg_R(H)$, and analogous for $outdeg_H(v)$.

The Boolean value for $\mathsf{x} = \mathsf{i}$ for any node/edge variable x and node/edge constant i not in R must be false in the resulting graph. Analogously, $\mathsf{x} = \mathsf{i}$ is always true. Also, all variables in the left-application condition should not represent any new nodes and edges in the right-hand side.

Definition 5.9 (Adjusment)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and a condition c over L. Let c' be a condition over L that is obtained from c by changing every term $\mathsf{incon}(\mathsf{x})$ (or $\mathsf{outcon}(\mathsf{x})$) for $x \in V_K$ with $\mathsf{indeg}(\mathsf{x})$–$indeg_R(x)$ (or $\mathsf{outdeg}(\mathsf{x})$–$outdeg_R(x)$). Let also $\{v_1, \cdots, v_n\}$ and $\{e_1, \cdots, e_m\}$ denote the set of all nodes and edges in $R - K$ respectively. The *adjusted* condition of c w.r.t r, denoted by $\mathrm{Adj}_{\mathsf{FO}}(c, r)$, is a condition over R that is defined inductively, where c_1, c_2 are conditions over L:

1. If c is true or false, $\mathrm{Adj}_{\mathsf{FO}}(c, r) = c'$;
2. If c is the predicates $\mathsf{int}(\mathsf{x}), \mathsf{char}(\mathsf{x}), \mathsf{string}(\mathsf{x})$ or $\mathsf{atom}(\mathsf{x})$ for a list variable x, $\mathrm{Adj}_{\mathsf{FO}}(c, r) = c'$;
3. If $c = \mathsf{root}(\mathsf{x})$ for some term x representing a node, $\mathrm{Adj}_{\mathsf{FO}}(c, r) = c'$
4. If $c = x_1 \ominus x_2$ for some terms x_1, x_2 and $\ominus \in \{=, \neq, <, \leq, >, \geq\}$,
$$\mathrm{Adj}_{\mathsf{FO}}(c, r) = \begin{cases} \mathsf{false} & \text{, if} \ominus \in \{=\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ \mathsf{true} & \text{, if } \ominus \in \{\neq\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ c' & \text{, otherwise} \end{cases}$$
5. $\mathrm{Adj}_{\mathsf{FO}}(c_1 \vee c_2, r) = \mathrm{Adj}_{\mathsf{FO}}(c_1, r) \vee \mathrm{Adj}_{\mathsf{FO}}(c_2, r)$
6. $\mathrm{Adj}_{\mathsf{FO}}(c_1 \wedge c_2, r) = \mathrm{Adj}_{\mathsf{FO}}(c_1, r) \wedge \mathrm{Adj}_{\mathsf{FO}}(c_2, r)$
7. $\mathrm{Adj}_{\mathsf{FO}}(\neg c_1, r) = \neg \mathrm{Adj}_{\mathsf{FO}}(c_1, r)$
8. $\mathrm{Adj}_{\mathsf{FO}}(\exists_{\mathsf{v}} \mathsf{x}(c_1), r) = \exists_{\mathsf{v}} \mathsf{x}(x \neq v_1 \wedge \cdots \wedge x \neq v_n \wedge \mathrm{Adj}_{\mathsf{FO}}(c_1, r))$
9. $\mathrm{Adj}_{\mathsf{FO}}(\exists_{\mathsf{e}} \mathsf{x}(c_1), r) = \exists_{\mathsf{e}} \mathsf{x}(x \neq e_1 \wedge \cdots \wedge x \neq e_m \wedge \mathrm{Adj}_{\mathsf{FO}}(c_1, r))$
10. $\mathrm{Adj}_{\mathsf{FO}}(\exists_{\mathsf{l}} \mathsf{x}(c_1), r) = \exists_{\mathsf{l}} \mathsf{x}(\mathrm{Adj}_{\mathsf{FO}}(c_1, r))$ □

Example 5.5.
Let p_1 denotes $\mathrm{Lift}_{\mathsf{FO}}(q_1, \mathsf{del}^{\vee})$ and p_2 denotes $\mathrm{Lift}_{\mathsf{FO}}(q_2, \mathsf{copy}^{\vee})$ we obtained in Example 5.4 for $q_1, q_2, \mathsf{del}^{\vee}$, and $\mathsf{copy}^{\vee}$ we defined in Section 5.1. Then based on Definition 5.9,

.

1. $\mathrm{Adj}_{\mathsf{FO}}(p_1, r_1) = \neg \exists_{\mathsf{e}} \mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e}$
2. $\mathrm{Adj}_{\mathsf{FO}}(p_2, r_2) = \exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \neq 2 \wedge \neg \mathsf{root}(\mathsf{x})) \wedge \mathsf{outdeg}(1) \neq 1$

The main purpose of transformation Adj is to adjust the obtained left-application condition such that it can be satisfied by the replacement graph of the resulting graph.

Lemma 5.10. Let us consider a host graph G, a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $g : L^\alpha \rightarrow G$ for some label assignment α_L, and a precondition c. Let H be a host graph such that $G \Rightarrow_{w,g,g^*} H$ for some injective morphism $g^* : R^\beta \rightarrow H$ where $\beta_R(i) = \alpha_L(i)$ for all common item i in domain β_R and $\alpha_L(i)$. Then,

$$\rho_g(G) \vDash (\mathrm{Lift}_{\mathsf{FO}}(c, w))^\alpha \text{ implies } \rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Lift}_{\mathsf{FO}}(c, w), r))^\beta$$

Proof. Note that $\mathrm{Adj}_{\mathsf{FO}}(c, r)$ does not change any term representing label in c such that $\mathrm{Adj}_{\mathsf{FO}}(c^\alpha, r) \equiv \mathrm{Adj}_{\mathsf{FO}}(c, r)^\alpha$ for all label assignment α_L. Also, note that $\mathrm{Adj}_{\mathsf{FO}}(c, r)$ does not contain any variable x in R that does not exist in L. Hence, $\mathrm{Adj}_{\mathsf{FO}}(c, r)^\alpha = \mathrm{Adj}_{\mathsf{FO}}(c, r)^\beta$. Assuming $\rho_g(G) \vDash c^\alpha$ for $c = \mathrm{Lift}_{\mathsf{FO}}(c, w)$, we prove that $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(c, r))^\beta$ inductively bellow:
Base case.

1. If c is true or false, we know that the lemma holds because every graph satisfies true and no graph satisfies false

2. If c is the predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$ or $\mathsf{atom(x)}$ for a list variable x, $c' \equiv c$ and satisfaction of c is independent on the host graph such that $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash c'^\alpha$ and $c'^\alpha = c'^\beta$.

3. If c is the predicate $\mathsf{root(x)}$ for some term x representing a node, then $x \notin V_L$ (see Definition 5.5 point 2), x is a variable representing $V_{\rho_g(G)} - (V_L) = V_{\rho_{g^*}(H)} - V_R$ (see Observation 5.3), or x is the function $\mathsf{s(x)}$ or $\mathsf{t(x)}$ for some edge variable x representing an edge in $E_{\rho_g(G)} - E_L = E_{\rho_{g^*}(H)} - E_R$ (see Definition 5.5 point 1(b) and 5.3). Hence, x representing a node in $\rho_g(G) - L$, which is also in $\rho_{g^*}(H) - R$ so that if $\mathsf{root(x)}$ is true in $\rho_g(G)$, $\mathsf{root(x)}$ must be true in $\rho_{g^*}(H)$, and label assignment has nothing to do with this.

4. If $c = x_1 \ominus x_2$, if x_1 and x_2 are terms representing lists, then x_1 and x_2 independent to nodes and edges in V_L unless x_1 or x_2 is in the form $\mathsf{incon(v)} or \mathsf{outcon(v)}$ for some $v \in V_K$ (see Definition 5.5 point 1(b) and 5.3). However, because $outcon(v) = outdeg_{\rho_g(G)}(v) - outdeg_L(v) = outdeg_{\rho_{g^*}(H)}(v) - outdeg_R(v)$, then semantics of $\mathsf{outcon(v)}$ in $\rho_g(G)$ is equivalent to semantics of $\mathsf{indeg(v)} - indeg_R(v)$ in $\rho_{g^*}(H)$. Hence, c is either independent to nodes and edges in V_L or contain $\mathsf{outcon(x)}$ or $\mathsf{incon(x)}$, $\rho_g(G) \vDash c$ implies $\rho_{g^*}(H) \vDash c' = \mathrm{Adj}_{\mathsf{FO}}(c, r)$, or c. If c is $x_1 = x_2$ and x_1 or x_2 is a constant in $(V_L - V_K)$ or in E_L, we know that there is no node/edge in $\rho_{g^*}(H)$ that is equal to the constant because the constant gets deleted by the rule, such that

$\rho_{g^*}(H) \vDash \mathsf{false} = \mathrm{Adj}_{\mathsf{FO}}(c, r)$. Analogously, if c is $x_1 \neq x_2$ and x_1 or x_2 is a constant in $(V_L - V_K)$ or in E_L, every node/edge in $\rho_{g^*}(H)$ does not equal to the node or edge such that $\rho_{g^*}(H) \vDash \mathsf{true} = \mathrm{Adj}_{\mathsf{FO}}(c, r)$.

Inductive case. Assuming $\rho_g(G) \vDash c_1^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{FO}}(c_1, r)^\beta$ and $\rho_g(G) \vDash c_2^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{FO}}(c_2, r)^\beta$ for some conditions c_1, c_2 over L,

1. $\rho_g(G) \vDash (c_1 \vee c_2)^\alpha$ implies $\rho_g(G) \vDash c_1^\alpha$ or $\rho_g(G) \vDash c_2^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{FO}}(c_1, r)^\beta$ or $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{FO}}(c_2, r)^\beta$, implies $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(c_1, r) \vee \mathrm{Adj}_{\mathsf{FO}}(c_2, r))^\beta$.
2. $\rho_g(G) \vDash (c_1 \wedge c_2)^\alpha$ implies $\rho_g(G) \vDash^\mu c_1^\alpha$ and $\rho_g(G) \vDash^\mu c_2^\alpha$ for some assignment μ which implies $\rho_{g^*}(H) \vDash^\mu \mathrm{Adj}_{\mathsf{FO}}(c_1, r)^\beta$ and $\rho_{g^*}(H) \vDash^\mu \mathrm{Adj}_{\mathsf{FO}}(c_2, r)^\beta$ implies $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(c_1, r) \wedge \mathrm{Adj}_{\mathsf{FO}}(c_2, r))^\beta$
3. $\rho_g(G) \vDash \neg c_1^\alpha$ implies $\neg(\rho_g(G) \vDash \mu c_1^\alpha)$ for some assignment μ which implies $\neg(\rho_{g^*}(H) \vDash^\mu (\mathrm{Adj}_{\mathsf{FO}}(c_1, r))^\beta)$, implying $\rho_{g^*}(H) \vDash \neg(\mathrm{Adj}_{\mathsf{FO}}(c_1, r))^\beta$
4. If $c = \exists_{\mathsf{v}} x(c_1)$, recall that every node variable x in c does not represent node in L. $\rho_g(G) \vDash (\exists_{\mathsf{v}} x(c_1))^\alpha$ implies $\rho_g(G) \vDash (c_1^{x \mapsto v})^\alpha$ for some $v \in V_{\rho_g(G)} - V_L = V_{\rho_{g^*}(H)} - V_R$ which implies $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(c_1^{[x \mapsto v]}, r))^\beta$. Since $v \notin V_R$, $\rho_{g^*}(H) \vDash (\exists_{\mathsf{v}} x(x \neq v_1 \wedge \cdots \wedge x \neq v_n \wedge \mathrm{Adj}_{\mathsf{FO}}(c_1, r)))^\beta$
5. If $c = \exists_{\mathsf{v}} x(c_1)$, the proof is analogous to above
6. $\rho_g(G) \vDash (\exists_{\mathsf{l}} x(c_1))^\alpha$ implies $\rho_g(G) \vDash (c_1^{x \mapsto k})^\alpha$ for some $k \in \mathbb{L}$ which implies $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(c_1^{[x \mapsto k]}, r))^\beta = (\mathrm{Adj}_{\mathsf{FO}}(c_1, r)^{[x \mapsto k]})^\beta$, which means $\rho_{g^*}(H) \vDash (\exists_{\mathsf{l}} x(\mathrm{Adj}_{\mathsf{FO}}(c_1, r)))^\beta$.

□

Note that any unrestricted rule schema r is invertible. The transformation Adj adjusts a left-application condition to the properties of the resulting graph w.r.t the given unrestricted rule schema. This means, adjusting the properties of the resulting graph w.r.t the inverse of the unrestricted rule schema should resulting in the initial left-application condition.

Lemma 5.11. Let us consider host graph G, a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, and a precondition c. Let $g : L^\alpha \rightarrow R$ for some label assignment α_L be an injective morphism satisfying the dangling condition. Then

$$\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Lift}_{\mathsf{FO}}(c, w), r), r^{-1})^\alpha \text{ if and only if } \rho_g(G) \vDash \mathrm{Lift}_{\mathsf{FO}}(c, w)^\alpha$$

Proof. Here we prove that $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c, r), r^{-1})$ if and only if $\rho_g(G) \vDash c$ inductively, where $c = \mathrm{Lift}_{\mathsf{FO}}(c, r)$:
Base case.

1. If c is true or false, $\mathrm{Adj}_{\mathsf{FO}}(c,r) = c' = \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
2. If c is the predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$ or $\mathsf{atom(x)}$ for a list variable x, $c' \equiv c$ such that $\mathrm{Adj}_{\mathsf{FO}}(c,r) = c' = \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
3. If c is the predicate $\mathsf{root(x)}$, $c' \equiv c$ such that $\mathrm{Adj}_{\mathsf{FO}}(c,r) = c' = \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
4. If c is $x_1 = x_2$ for x_1 or x_2 a node or edge constant in $L - K$, both x_1 and x_2 cannot be constants (see construction of $\mathrm{Val}_{\mathsf{FO}}$ which is used to construct c). Then, one of them must be a node or edge variable (which does not represent node in L - see Observation 5.3), or the function $\mathsf{s(x)}$ or $\mathsf{t(x)}$ for some edge variable x. Observation 5.3 shows us that x does not representing edge in L, and g satisfies the dangling condition implies $\mathsf{s(x)}$ and $\mathsf{t(x)}$ do not represent nodes in $\rho_g(G) - (L - K)$. Hence, $x_1 = x_2$ is always false in $\rho_g(G)$. Otherwise for $c = x_1 \ominus x_2$, $\mathrm{Adj}_{\mathsf{FO}}(c,r) = c' = \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$.

Inductive case.
Assume that $c_1 \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1})$ and $c_2 \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_2,r),r^{-1})$ for conditions c_1, c_2 over L.

1. $\rho_g(G) \vDash c_1 \vee c_2$
   iff $\rho_g(G) \vDash c_1$ or $\rho_g(G) \vDash c_2$
   iff $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1})$ or $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_2,r),r^{-1})$
   iff $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}) \vee$
   $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_2,r),r^{-1}) \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
2. $\rho_g(G) \vDash c_1 \wedge c_2$ implies $\rho_g(G) \vDash^{\beta} c_1 \wedge \rho_g(G) \vDash^{\beta} c_2$ for some assignment β
   iff $\rho_g(G) \vDash^{\beta} \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}) \wedge \rho_g(G) \vDash^{\beta} \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_2,r),r^{-1}))$
   iff $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}) \wedge \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_2,r),r^{-1})) \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
3. $\rho_g(G) \vDash \neg c_1$ iff $\neg(\rho_g(G) \vDash \beta c_1)$ for some assignment β
   iff $\neg(\rho_g(G) \vDash^{\beta} \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1})$,
   iff $\rho_g(G) \vDash \neg\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}) \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
4. If $c = \exists_{\mathsf{v}} x(c_1)$,
   $\mathrm{Adj}_{\mathsf{FO}}(c,r) = \exists_{\mathsf{v}}\mathsf{x}(x \neq v_1 \wedge \cdots \wedge x \neq v_n \wedge \mathrm{Adj}_{\mathsf{FO}}(c_1,r))$,
   so that $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1}) = \exists_{\mathsf{v}}\mathsf{x}(\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}))$.
   Hence,
   $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c,r),r^{-1})$
   iff $\rho_g(G) \vDash \exists_{\mathsf{v}}\mathsf{x}(\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1,r),r^{-1}))$
   iff $\rho_g(G) \vDash \exists_{\mathsf{v}}\mathsf{x}(c_1) = c$
5. If $c = \exists_{\mathsf{v}} x(c_1)$, the proof is analogous to above

6. If $c = \exists_{\mathsf{l}} x(c_1)$,
   $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c, r), r^{-1})$
   iff $\rho_g(G) \vDash \exists_{\mathsf{l}}\mathsf{x}(\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c_1, r), r^{-1}))$
   iff $\rho_g(G) \vDash \exists_{\mathsf{l}}\mathsf{x}(c_1) = c$

Since the construction of $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c, r), r^{-1})$ does not any term representing labels, $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c^\alpha, r), r^{-1}) \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c, r)^\alpha, r^{-1}) \equiv \mathrm{Adj}_{\mathsf{FO}}(\mathrm{Adj}_{\mathsf{FO}}(c, r), r^{-1})^\alpha$. Hence, the lemma is valid. □

Actually, from the transformation Adj we already obtain a right-application condition. However, we want a stronger condition such that we add the specification of the right-hand graph. In addition, since the resulting graph should also satisfy the existing right-application of the given generalised rule schema, and the comatch should also satisfy the dangling condition.

Definition 5.12 (Shifting)**.** Let us consider a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$. Right application condition w.r.t. w, denoted by $\mathrm{Shift}_{\mathsf{FO}}(w)$, is defined as:

$$\mathrm{Shift}_{\mathsf{FO}}(w) = \mathrm{Adj}_{\mathsf{FO}}(ac_L, r) \wedge ac_R \wedge \mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1}).$$

□

Example 5.6. Let us consider $\mathrm{Lift}_{\mathsf{FO}}(q_1, \mathsf{del}^\vee)$ and $\mathrm{Lift}(q_2, \mathsf{copy}^\vee)$ we obtained in Example 5.4 for q_1, q_2 we defined in 5.1, and the generalised rule schemata $\mathsf{del}^\vee, \mathsf{copy}^\vee$ which are the generalised version of r_1, r_2 of Figure 5.2 and Figure 5.3. For generalised rule schemata $w_1 = \langle r_1, \mathrm{Lift}_{\mathsf{FO}}(q_1, \mathsf{del}^\vee), \mathsf{true}\rangle$ and $w_2 = \langle r_2, \mathrm{Lift}_{\mathsf{FO}}(q_2, \mathsf{copy}^\vee), \mathsf{true}\rangle$, based on Definition 5.12, 3.23, and 5.1, we have

$$\begin{aligned}
\mathrm{Shift}_{\mathsf{FO}}(w_1) \;=\; & \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{m_v}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e} \\
& \wedge \mathsf{l_v}(1) = \mathsf{a} \wedge \mathsf{l_v}(2) = \mathsf{b} \wedge \mathsf{l_e}(\mathsf{e1}) = \mathsf{d} + \mathsf{e} \wedge \mathsf{m_v}(1) = \mathsf{red} \\
& \wedge \mathsf{m_v}(2) = \mathsf{none} \wedge \mathsf{m_e}(\mathsf{e1}) = \mathsf{none} \wedge \mathsf{s}(\mathsf{e1}) = 1 \wedge \mathsf{t}(\mathsf{e1}) = 2 \\
& \wedge \neg\mathsf{root}(1) \wedge \neg\mathsf{root}(2) \wedge \mathsf{int}(\mathsf{d}) \wedge \mathsf{int}(\mathsf{e}) \\
\mathrm{Shift}_{\mathsf{FO}}(w_2) \;=\; & \exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \neq 2 \wedge \neg\mathsf{root}(\mathsf{x})) \wedge \mathsf{outdeg}(1) \neq 1 \\
& \wedge \mathsf{l_v}(1) = \mathsf{a} \wedge \mathsf{l_v}(2) = \mathsf{a} \wedge \mathsf{l_e}(\mathsf{e1}) = \mathsf{empty} \wedge \mathsf{m_v}(1) = \mathsf{none} \\
& \wedge \mathsf{m_v}(2) = \mathsf{none} \wedge \mathsf{m_e}(\mathsf{e1}) = \mathsf{dashed} \wedge \mathsf{s}(\mathsf{e1}) = 1 \wedge \mathsf{t}(\mathsf{e1}) = 2 \\
& \wedge \neg\mathsf{root}(1) \wedge \mathsf{root}(2) \wedge \mathsf{indeg}(2) = 1 \wedge \mathsf{outdeg}(2) = 0
\end{aligned}$$

Proposition 5.13 (Shifting)**.** Let us consider a host graph G, a generalised rule $w = \langle r, ac_L, ac_R\rangle$ an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$, an injective morphism $g : L^\alpha \rightarrow G$ for some label assignment α_L, and a precondition c. Then for host graphs H such that $G \Rightarrow_{w,g,g^*} H$ with an right morphism $g^* : R^\beta \rightarrow H$ where $\beta_R(i) = \alpha_L(i)$ for every variable

i in L such that i in R, and for every node (or edge) i where $m_L^V(i) = m_R^V(i) = \texttt{any}$ (or $m_L^E(i) = m_R^E(i) = \texttt{any}$),

$\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Lift}_{\mathsf{FO}}(c,w)),r)^\beta$ if and only if $\rho_{g^*}(H) \vDash (\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c,w), ac_R\rangle))^\beta$

Proof. From the semantics of conjunction, we know that $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Lift}_{\mathsf{FO}}(c,w)),r)^\beta$ is implied by $\mathrm{Shift}_{\mathsf{FO}}(w)^\beta$, so now we show that $\mathrm{Adj}_{\mathsf{FO}}(\mathrm{Lift}_{\mathsf{FO}}(c,w)),r)^\beta$ implies $\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c,w), ac_R\rangle))^\beta$. That is, $\rho_{g^*}(H)$ satisfies $ac_R^\beta \wedge \mathrm{Spec}(R)^\beta \wedge \mathrm{Dang}(r^{-1})^\beta$. From Definition 3.18, $G \Rightarrow_{w,g,g^*} H$ implies $\rho_{g^*}(H) \vDash ac_R^\beta$. From the construction of $\mathrm{Spec}(R)$, $\mathrm{Spec}(R)^\beta \equiv \mathrm{Spec}(R^\beta)$ such that $\rho_{g^*}(H) \vDash \mathrm{Spec}(R)^\beta$ is implied by the injective morphism g^*. Finally, there is no label variable in $\mathrm{Dang}(r^{-1})$ such that $\mathrm{Dang}(r^{-1}) \equiv \mathrm{Dang}(r^{-1})^\beta$, which is implied by $G \Rightarrow_{w,g,g^*} H$ because nodes in $R-K$ must not incident to any edge in $\rho_{g^*}(H)-R$ so that their indegree and outdegree in R represents their indegree and outdegree in $\rho_{g^*}(H)$. □

5.5 From right-application condition to postcondition

The right-application condition we obtain from transformation Shift is strong enough to express properties of the replacement graph of any resulting graph. However, since we need a condition (without node/edge constant), we define transformation Post.

Definition 5.14 (Formula Post)**.** Let us consider a condition a (possibly contain node/edge constants). Let $\{x_1,\cdots,x_n\}$, $\{y_1,\cdots,y_m\}$, and $\{z_1,\cdots,z_k\}$ denote the set of free node, edge, and label (resp.) variables in $\mathrm{Var}(a)$. A postcondition w.r.t. a, denoted by $\mathrm{Post}(a)$, is the FO formula:

$$\mathrm{Post}(a) = \exists_{\mathsf{v}} x_1,\cdots,x_n(\exists_{\mathsf{e}} y_1,\cdots,y_m(\exists_{\mathsf{l}} z_1,\cdots,z_k(\mathrm{Var}(a)))).$$

□

To obtain a closed FO formula from the obtained right-application condition, we only need to variablise (see Definition 3.25) the node/edge constants in the right-application condition, then put an existential quantifier for each free variable in the resulting FO formula.

Example 5.7. Let us consider $a_1 = \mathrm{Shift}_{\mathsf{FO}}(\langle r_1, \mathrm{Lift}_{\mathsf{FO}}(q_1, \mathsf{del}^\vee), \mathsf{true}\rangle)$ and $a_2 = \mathrm{Shift}_{\mathsf{FO}}(\langle r_2, \mathrm{Lift}_{\mathsf{FO}}(q_2, \mathsf{copy}^\vee), \mathsf{true}\rangle)$ for $q_1, q_2, r_1, r_2, \mathsf{del}^\vee$, and $\mathsf{copy}^\vee$ we defined in Section

5.1. Then based on Definition 5.14 and 2.32, we have

$$\begin{aligned}
\mathrm{Post}(a_1) \;=\;& \exists_{\mathsf{v}} \mathsf{u}, \mathsf{v}(\mathsf{u} \neq \mathsf{v} \wedge \exists_{\mathsf{e}} \mathsf{w}(\exists_{\mathsf{l}} \mathsf{a}, \mathsf{b}, \mathsf{d}, \mathsf{e}(\\
& \neg \exists_{\mathsf{e}} \mathsf{x}(\mathsf{x} \neq \mathsf{w} \wedge \mathsf{s}(\mathsf{x}) \neq \mathsf{u} \wedge \mathsf{s}(\mathsf{x}) \neq \mathsf{v} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{s}(\mathsf{w})) \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e} \\
& \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{u}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{v}) = \mathsf{b} \wedge \mathsf{l}_{\mathsf{e}}(\mathsf{w}) = \mathsf{d} + \mathsf{e} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{u}) = \mathsf{red} \\
& \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{v}) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{e}}(\mathsf{w}) = \mathsf{none} \wedge \mathsf{s}(\mathsf{w}) = \mathsf{u} \wedge \mathsf{t}(\mathsf{w}) = \mathsf{v} \\
& \wedge \neg \mathsf{root}(\mathsf{u}) \wedge \neg \mathsf{root}(\mathsf{v}) \wedge \mathsf{int}(\mathsf{d}) \wedge \mathsf{int}(\mathsf{e}))))) \\
\mathrm{Post}(a_2) \;=\;& \exists_{\mathsf{v}} \mathsf{u}, \mathsf{v}(\mathsf{u} \neq \mathsf{v} \wedge \exists_{\mathsf{e}} \mathsf{w}(\exists_{\mathsf{l}} \mathsf{a}(\\
& \exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq \mathsf{u} \wedge \mathsf{x} \neq \mathsf{v} \wedge \neg \mathsf{root}(\mathsf{x})) \wedge \mathsf{outdeg}(\mathsf{u}) \neq 1 \\
& \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{u}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{v}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{e}}(\mathsf{w}) = \mathsf{empty} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{u}) = \mathsf{none} \\
& \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{v}) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{e}}(\mathsf{w}) = \mathsf{dashed} \wedge \mathsf{s}(\mathsf{w}) = \mathsf{u} \wedge \mathsf{t}(\mathsf{w}) = \mathsf{v} \\
& \wedge \neg \mathsf{root}(\mathsf{u}) \wedge \mathsf{root}(\mathsf{v}) \wedge \mathsf{indeg}(\mathsf{v}) = 1 \wedge \mathsf{outdeg}(\mathsf{v}) = 0)))
\end{aligned}$$

Proposition 5.15 (Post)**.** Let us consider a host graph G, a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, and a precondition c. Then for all host graph H such that there exists an injective morphism $g^* : R^\beta \rightarrow H$ for a label assignment β_R,

$$\rho_{g^*}(H) \vDash (\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta \text{ iff } H \vDash \mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$$

Proof. From Lemma 3.26, $\rho_{g^*}(H) \vDash (\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$ iff $H \vDash \mathrm{Var}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$. If there is no node (or edge) in H, then there is no node (or edge) constant in $\rho_{g^*}(H)$ since they are isomorphic. Hence, there is no free node (or edge) variable in $\mathrm{Var}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$ so that there is no additional node (or edge) quantifier for $\mathrm{Var}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$. If there exists a node (or edge) in H, then from Fact 3.1, adding an existential quantifier will not change its satisfaction on H. Hence, $H \vDash \mathrm{Var}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$ iff $H \vDash \mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, w), ac_R \rangle))^\beta$. □

Finally, we show that $\mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, \langle r, \Gamma^\vee, \mathsf{true} \rangle \rangle), \mathsf{true}))$ is a strongest liberal postcondition w.r.t. c and $\langle r, \Gamma \rangle$. That is, by showing that for all host graph G, $G \vDash c$ and $G \Rightarrow_{\langle r, \Gamma \rangle} H$ implies $H \vDash \mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, \langle r, \Gamma^\vee, \mathsf{true} \rangle \rangle), \mathsf{true}))$, and showing that for all host graph H, $H \vDash \mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, \langle r, \Gamma^\vee, \mathsf{true} \rangle \rangle), \mathsf{true}))$ implies the existence of host graph G such that $G \vDash c$ and $G \Rightarrow_{\langle r, \Gamma \rangle} H$.

Theorem 5.16 (Post is a strongest liberal postcondition)**.** Let us consider a precondition c and a conditional rule schema $r = \langle \langle L \leftarrow K \rightarrow R \rangle, \Gamma \rangle$. Then, $\mathrm{Post}(\mathrm{Shift}_{\mathsf{FO}}(\langle r, \mathrm{Lift}_{\mathsf{FO}}(c, r^\vee), \mathsf{true} \rangle))$ is a strongest liberal postcondition w.r.t. c and r.

Proof. Based on the definition of liberal postcondition (see Definition 4.1), `true` is a liberal postcondition for any precondition and conditional rule schema because . Hence, there must

exist a strongest liberal postcondition because we have at least one liberal postcondition for any precondition and conditional rule schema.

Then, let us also consider $c, r, \Gamma, G, H, g, g^*, \alpha, \beta$ as stated in the theorem. From Lemma 3.20, $G \Rightarrow_r H$ iff $G \Rightarrow_{w,g,g*} H$ for $w = \langle r, \Gamma, \mathsf{true}\rangle$, some injective morphisms $g : L^\alpha \to G$ and $g^* : R^\beta \to H$ with label assignment α_L and β_R where $\beta_R(i) = \alpha_L(i)$ for every variable i in L such that i is in R, and for every node (or edge) i where $m_L^V(i) = m_R^V(i) = \mathtt{any}$ (or $m_L^E(i) = m_R^E(i) = \mathtt{any}$). For $G \vDash c$, from Proposition 5.8, we get that $\rho_g(G) \vDash \mathrm{Lift}((c, w))^\alpha$. Then, from Lemma 5.10 and Proposition 5.13 we know that $\rho_{g^*}(H) \vDash \mathrm{Shift}(\langle r, \mathrm{Lift}(c, w), \mathsf{true}\rangle)^\beta$. Finally, from Proposition 5.15, we have $\mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, w\rangle, \mathsf{true})))$ is a liberal postcondition w.r.t. c and r because $H \vDash \mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, w\rangle, \mathsf{true})))$.

To show that $d = \mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, w\rangle, \mathsf{true})))$ is a strongest liberal postcondition, based on Lemma 4.2, we need to show that for every graph H satisfying d, there exists a host graph G satisfying c such that $G \Rightarrow_r H$.

From Corollary 5.2 and the way we define $\mathrm{Shift}(\langle r, \mathrm{Lift}(c, w\rangle, \mathsf{true})$ (which clearly implies $\mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1})$ due to the semantic of conjunction), we know there exists injective morphism $g' : R^\gamma \to H$ that satisfies the dangling condition for some assignment γ. Also, from Proposition 5.15, $H \vDash \mathrm{Post}(\mathrm{Shift}(\langle r, \mathrm{Lift}(c, w), \mathsf{true}\rangle)))$ implies
$\rho_{g'}(H) \vDash \mathrm{Shift}(\langle r, \mathrm{Lift}(c, w), \mathsf{true}\rangle))$. Hence, there exists a natural double-pushout bellow where every morphism is inclusion:

$$\begin{array}{ccccc} R^\gamma & \longleftarrow & K & \longrightarrow & L^\gamma \\ \downarrow & (1) & \downarrow & (2) & \downarrow \\ \rho_{g'}(H) & \longleftarrow & D & \longrightarrow & A \end{array}$$

$\rho_{g'} H \vDash \mathrm{Shift}(\langle r, \mathrm{Lift}(c, w), \mathsf{true}\rangle))$ also implies that $\rho_{g'}(H) \vDash \mathrm{Adj}(\mathrm{Lift}(c, w), r)$ from the definition of Shift and the semantics of conjunction. From Lemma 5.10, this implies A satisfies $\mathrm{Adj}(\mathrm{Adj}(\mathrm{Lift}(c, w), r), r^{-1})$, which implies $A \vDash c$ from Lemma 5.11. Since direct derivations are invertible, $A \Rightarrow_w H$. Hence, $A \Rightarrow_r H$. □

5.6 Summary

This chapter shows us how can we construct a strongest liberal postcondition over a given precondition and a (conditional) rule schema, where preconditions are limited to closed first-order formulas. From the generalised rule schema application we described in the previous chapter, we know that the match in an application must satisfy the dangling condition and

the rule schema condition. Hence, the left-application condition must satisfy the properties. Moreover, we need to consider the given precondition as well since we are only interested in the graphs satisfying the precondition.

We define the transformation $\text{Split}_{\mathsf{FO}}$ by considering all possibilities of node/edge variables in the given precondition expressing nodes/edges in the given rule schema's left-hand graph. The obtained condition is then evaluated based on the left-hand graph to obtain a simpler condition with respect to the left-hand graph. We then obtain a left-application condition from the conjunction of this condition, the dangling condition, and the rule schema condition.

To obtain a right application condition that must be satisfied by the comatch of any possible derivation, we introduce transformation Shift. This transformation basically considers the difference that may occur due to the deletion of nodes/edges by the given rule schema. We also add the condition that expresses the specification of the right-hand graph to have a stronger right-application condition. Finally, we transform the right-application condition to a closed first-order formula to obtain a liberal postcondition. We then prove that the obtained liberal postcondition is actually a strongest liberal postcondition.

The approach we use to obtain a strongest liberal postcondition is basically similar to the approach used in [1, 26] to obtain a weakest liberal precondition with respect to a given postcondition and a rule (schema). The approach is similar in how it considers all possibilities instance based on morphisms due to the rule schema application, then consider how the deletion of nodes/edges may affect each of the possibility. However, it is likely that finding possibilities of variables expressing nodes/edges in a matching is simpler than finding instances of diagrams based on morphisms. Finding instances in standard logic only consider the matching between nodes/edges we have in left-hand graph with node/edge variables we have in the precondition. However, finding instances based on morphisms need to consider the possibility of adjacency/incidence with other nodes we have in a graph, with considering marks of the nodes/edges as well.

In [1], it is shown that a weakest liberal precondition is essential in graph program verification because it can be used as an axiom in Hoare-style verification. However, it is stated that a strongest liberal postcondition can not be obtained from the construction of a weakest liberal precondition because rule schemata are not invertible. Here, we are able to construct a strongest liberal postcondition based on a generalised rule schema, which is invertible. Hence, we should be able to use the construction we defined in this chapter to obtain a weakest liberal precondition as well (see Chapter 7).

Chapter 6

Extension to monadic second-order logic

In the previous chapter, we have defined the construction of a strongest liberal postcondition over a closed first-order formula. Here, we extend the construction for monadic second-order formulas. Before we start with the extension, we first analyse by example the comparison in constructing a strongest liberal postcondition for first-order formulas and monadic second-order formulas to find the intuition for the extension.

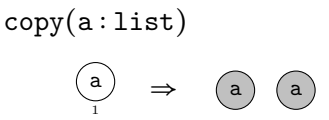
6.1 Constructing left and right-application condition by example

In the previous chapter, we have shown how to construct a strongest liberal postcondition for first-order formulas. To upgrade it to monadic second-order formulas, we need to consider functions and predicates of MSO formulas that are not considered in FO formulas. Before we start to define the construction formally, let us observe the construction by considering the rule r_3 of Figure 6.1. Let us also consider the formulas c_1, c_2, and c_3 below:

- $c_1 \equiv \forall_{\mathsf{v}} \mathsf{x}(\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none})$
- $c_2 \equiv \exists_{\mathsf{V}} \mathsf{X}(\forall_{\mathsf{v}} \mathsf{x}(\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$
- $c_3 \equiv \exists_{\mathsf{V}} \mathsf{X}(\forall_{\mathsf{v}} \mathsf{x}(\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \mathsf{card}(\mathsf{X}) = 2 * \mathsf{n})$

where c_1 expresses all nodes are unmarked, c_2 expresses there exists at least 2 unmarked nodes, and c_3 expresses the number of grey nodes is even.

Note that the interface of the rule copy is the empty graph. We intentionally does not preserve the node 1 and have two new nodes instead to see the effect of both removal and addition of an element in constructing a strongest liberal postcondition.




```
copy (a : list; )
[ | (1, a)
  | ]
=>
[ | (2#grey, a) (3#grey, a)
  | ]
interface = {}
where true
```

FIGURE 6.1: GP 2 conditional rule schema $\texttt{copy} = \langle r_3, \Gamma_3 \rangle$

6.1.1 Constructing left-application condition

In the previous chapter, the left-application condition is constructed by having transformations $\text{Split}_{\mathsf{FO}}$, Val_{FO}, $\text{Lift}_{\mathsf{FO}}$, and condition Dang. For a rule schema r, the condition $\text{Dang}(r)$ is a condition that expresses the dangling condition. Since the dangling condition is related to the individual of deleted nodes, first-order formula is enough to express it so that $\text{Dang}(r)$ can be used as in the construction of left-application condition for FO formulas. $\text{Dang}(r_3)$ then can be constructed as in the previous chapter (see Definition 5.1), that is:

$$\mathsf{indeg}(1) = 0 \wedge \mathsf{outdeg}(1) = 0$$

For a rule schema r and a precondition c, $\text{Split}_{\mathsf{FO}}(c, r)$ forms a disjunction from all possibilities connection between the node/edge variables in c and nodes/edges in the left-hand graph of r. In c_1, we only have a first-order variables so that as the connection between a node (or edge) variable x in c_1 and left-hand graph node (or edge) i of r_3 is either $x = i$ or $x \neq i$. Hence, $\text{Split}_{\mathsf{MSO}}(c_1, r_3)$ should be equal to $\text{Split}_{\mathsf{FO}}(c_1, r_3)$, that is:

$$\neg(\mathsf{m_v}(1) \neq \mathsf{none}) \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none})$$

In first-order formulas, a node/edge variable can only represent exactly one node/edge in a graph. Hence, we can use substitution when we consider a variable x representing a node/edge i. However, we can not use substitution for set variables. When a node (or edge) set variable exists, then a connection between a node (or edge) in the left-hand graph will be either the node is an element of the set represented by the variable or not. Then, we can have $\text{Split}_{\mathsf{MSO}}(c_2, r_3)$ as follows:

$$\begin{aligned}\exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg(1 \in \mathsf{X} \wedge \mathsf{m_v}(1) \neq \mathsf{none}) \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2) \\ \wedge (1 \notin \mathsf{X} \Rightarrow \neg(1 \in \mathsf{X} \wedge \mathsf{m_v}(1) \neq \mathsf{none}) \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2))\end{aligned}$$

Similarly, if we denote by c_4 the formula $\forall_{\mathsf{v}} \mathsf{x}(\mathsf{m_v}(\mathsf{x}) = \mathsf{grey} \leftrightarrow \mathsf{x} \in \mathsf{X})$, then $\text{Split}_{\mathsf{MSO}}(c_3, r_3)$ should be:

$$\exists_{\mathsf{V}} \mathsf{X}((1 \in \mathsf{X} \Rightarrow \text{Split}_{\mathsf{MSO}}(\mathsf{c_4}, \mathsf{r_3}) \wedge \mathsf{card}(\mathsf{X}) = 2^* \mathsf{n}) \wedge (1 \notin \mathsf{X} \Rightarrow \text{Split}_{\mathsf{MSO}}(\mathsf{c_4}, \mathsf{r_3}) \wedge \mathsf{card}(\mathsf{X}) = 2^* \mathsf{n}))$$

where $\text{Split}_{\mathsf{MSO}}(c_4, r_3)$ is:

$$\begin{aligned} &\neg(1 \notin \mathsf{X} \wedge \mathsf{m_v}(1) = \mathsf{grey}) \wedge \neg(\mathsf{m_v}(1) \neq \mathsf{grey} \wedge 1 \in \mathsf{X}) \\ \wedge &\neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \end{aligned}$$

Next, the transformation $\text{Val}_{\mathsf{MSO}}$. This transformation is done to simplify a condition with respect to the left-hand graph of the given rule. Since there is no node or edge set variable in c_1, $\text{Val}_{\mathsf{MSO}}(\text{Split}_{\mathsf{MSO}}(c_1, r_3), r_3)$ should be equal to $\text{Val}_{\mathsf{FO}}(\text{Split}_{\mathsf{MSO}}(c_1, r_3), r_3)$, that is:

$$\neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none})$$

Since we do not have functions with node/edge set constant as their argument, we only need to simplify the condition we have so that $\text{Val}_{\mathsf{MSO}}(\text{Split}_{\mathsf{MSO}}(c_2, r_3), r_3)$ should be:

$$\begin{aligned} \exists_{\mathsf{V}} \mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2) \\ \wedge (1 \notin \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)) \end{aligned}$$

because $m_L^V(1) = \texttt{none}$ so that we change $\mathsf{m_v}(1) \neq \mathsf{none}$ to false and then simplify the condition. In addition, for each implication we have, we can simplify the implication by substituting true for the premise in the right-hand side. For example, for the implication with $1 \in \mathsf{X}$ as the premise, we replace $1 \in X$ on the right-hand side of the implication to true.

Similarly, $\text{Val}_{\mathsf{MSO}}(\text{Split}_{\mathsf{MSO}}(c_3, r_3), r_3)$ should be:

$$\begin{aligned} &\exists_{\mathsf{V}} \mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathsf{false}) \\ &\quad \wedge (1 \notin \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \wedge \mathsf{card}(\mathsf{X}))) \end{aligned}$$

because $m_L^V(1) = \texttt{none}$ so that $\mathsf{m_v}(1) = \mathsf{grey}$ is false. The implication with $1 \in \mathsf{X}$ as premise can be simplified as $1 \in \mathsf{X} \Rightarrow \mathsf{false}$. Then for $\text{Val}_{\mathsf{MSO}}(\text{Dang}(r_3))$, it should be the same as $\text{Val}_{\mathsf{FO}}(\text{Dang}(r_3))$, that is: true.

Finally, $\text{Lift}_{\mathsf{MSO}}$ should similar to $\text{Lift}_{\mathsf{FO}}$, which is formed from the valuation of the conjunction of Dang, Split, and the given left-application condition. Let w denotes the generalised rule schema $r_3^\vee$. Since there is no rule schema application condition in r_3, left-application condition of w is true so that $\text{Lift}_{\mathsf{MSO}}(c_1, r_3)$ is $\text{Val}_{\mathsf{MSO}}(\text{Split}_{\mathsf{MSO}}(c_1, r_3), r_3)$ and $\text{Lift}_{\mathsf{MSO}}(c_2, r_3)$

is $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(c_2, r_3), r_3)$.

6.1.2 Constructing right-application condition

In the previous chapter, we use transformation $\mathrm{Adj}_{\mathsf{FO}}$ to adjust a left-application condition such that it can always be satisfied by the resulting graph. In constructing $\mathrm{Adj}_{\mathsf{FO}}$, we changed $\mathsf{x} = \mathsf{i}$ for some deleted node/edge i of the given rule schema to false and $\mathsf{x} \neq \mathsf{i}$ to true because i is not in the resulting graph so that x cannot be equal to i. However, it gets a lot trickier in MSO formula because of the function card. If a node gets deleted and the node is a member of a set of nodes represented by a set variable X, then the deletion change the value of the cardinality of the set.

We do not have a node/edge set variable in c_1, so that Adj for this case should be the same as in FO formula such that $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_1, w), r_3)$ is:

$$\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none})$$

because we change $x \neq 1$ to true and we add constraint $x \neq 2 \wedge x \neq 3$ since 2 and 3 are new nodes.

The tricky part can be seen in the case of c_2. If $i \in X$ is true and i gets deleted, then cardinality of X should be changed. For example, when a replacement graph of the input graph satisfies $1 \in \mathsf{X} \wedge \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}$, then the replacement graph of any resulting graph where 1 is a deleted node should satisfy $\mathsf{card}(\mathsf{X}) + 1 = 2^*\mathsf{n}$. Hence, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_3, w), r_3)$ should be:

$$\begin{aligned}\exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) + 1 \geq 2)\\ \wedge(1 \notin \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2))\end{aligned}$$

Then, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_3, w), r)$ should be: $\exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathsf{false})$
$\wedge (1 \notin \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \wedge \mathsf{card}(\mathsf{X})))$

For the transformation Shift, in the previous chapter, $\mathrm{Shift}_{\mathsf{FO}}$ is constructed by forming conjunction from $\mathrm{Adj}_{\mathsf{FO}}$, $\mathrm{Dang}(r^{-1})$, the given right application condition, and $\mathrm{Spec}(R)$. As mentioned before, there is no change in condition Dang for MSO formulas. Similarly, specification of the right-hand graph will be the same.

Let R denotes the right-hand graph of r_3 and $\mathrm{New}(r_3)$ denotes the conjunction of $\mathrm{Dang}(r_3^{-1})$, $\mathrm{Spec}(R)$, and the combination above, that is:

$$\mathsf{m_v}(2) = \mathsf{grey} \wedge \mathsf{m_v}(3) = \mathsf{grey} \wedge \mathsf{l_v}(2) = \mathsf{a} \wedge \mathsf{l_v}(3) = \mathsf{a}$$

$$\wedge \mathsf{indeg}(2) = 0 \wedge \mathsf{indeg}(3) = 0 \wedge \mathsf{outdeg}(2) = 0 \wedge \mathsf{outdeg}(3) = 0$$

Then, the condition $\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c_1, w), \mathsf{true}\rangle)$, $\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c_2, w), \mathsf{true}\rangle)$, and $\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c_3, w), \mathsf{true}\rangle)$ (respectively) should be $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_1, w), r_3) \wedge \mathrm{New}(r_3)$, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_2, w), r_3) \wedge \mathrm{New}(r_3)$, and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(c_3, w), r) \wedge \mathrm{New}(r_3)$.

From the examples above, we can see that the possibilities of nodes (or edges) becoming an element of node (or edge) set represented by a variable X is important in the transformation process. To cover this matter, we define a *subset formula* for a set which can be used to represent a subset of the set.

Definition 6.1 (Subset Formula)**.** Let us consider a set of nodes $L = \{v_1, \ldots, v_n\}$ for some $n \geq 1$. A *subset formula* for L with respect to a node set variable X has the form $c_1 \wedge c_2 \wedge \ldots \wedge c_n$ where for $i = 1, \ldots, n$, $c_i = v_i \in X$ or $v_i \notin X$. The formula true is the only subset formula for the empty set `empty` with respect to any set variable.

We say that the subset formula c for L w.r.t X represents a subset M of L if c implies $i \in X$ for every $i \in M$ and c implies $j \notin X$ for every $j \in L - M$. □

Example 6.1. For $L = \{1, 2\}$, both $1 \in X \wedge 2 \notin X$ and $2 \in X \wedge 1 \in X$ are subset formulas for L with respect to X.

Each subset formula for L represents a subset of L, and each subset of L is represented by a subset formula for L that is unique up to a reordering of the conjuncts.

6.2 From precondition to left-application condition

From the example we have above, constructing a left-application condition from a precondition and a rule schema for monadic second-order formulas is similar to what we have done for first-order formulas. We only have additional possibility of nodes or edges in the left-hand graph of a given rule schema being an element of a node or edge set represented by set variables in the precondition.

Definition 6.2 (Transformation Split)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. where $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let $2^{V_L} = \{V_1, \ldots, V_{2^n}\}$ be the power set of V_L, and $d_1, \ldots, d_{2^n}$ be subset formulas of V_L w.r.t. X where for every $i = 1, \ldots, 2^n$, d_i represents V_i. Similarly, let $2^{E_L} = \{E_1, \ldots, E_{2^m}\}$ be the power set of E_L, and $a_1, \ldots, a_{2^m}$ be subset formulas of E_L w.r.t. X where for every $i = 1, \ldots, 2^m$, a_i represents E_i.

For a condition c over L sharing no variables with r (note that it is always possible to replace the label variables in c with new variables that are distinct from variables in r), we define the condition $\mathrm{Split}_{\mathsf{MSO}}(c, r)$ over L inductively as follows:

- Base case.

  If c is a first-order formula,

  $$\mathrm{Split}_{\mathsf{MSO}}(c, r) = \mathrm{Split}_{\mathsf{MSO}}(c, r)$$

  If c is in the form $\mathsf{x} \in \mathsf{X}$ or $\mathsf{x} \notin \mathsf{X}$,

  $$\mathrm{Split}_{\mathsf{MSO}}(c, r) = c$$

  If c is in the form $x \otimes \mathsf{card}(\mathsf{X})$ for $\otimes \in \{=, \neq, <, \leq, >, \geq\}$,

  $$\mathrm{Split}_{\mathsf{MSO}}(c, r) = c$$

- Inductive case.

  Let c_1 and c_2 be conditions over L.

  1) $\mathrm{Split}_{\mathsf{MSO}}(c_1 \vee c_2, r) = \mathrm{Split}_{\mathsf{MSO}}(c_1, r) \vee \mathrm{Split}_{\mathsf{MSO}}(c_2, r)$,
  2) $\mathrm{Split}_{\mathsf{MSO}}(c_1 \wedge c_2, r) = \mathrm{Split}_{\mathsf{MSO}}(c_1, r) \wedge \mathrm{Split}_{\mathsf{MSO}}(c_2, r)$,
  3) $\mathrm{Split}_{\mathsf{MSO}}(\neg c_1, r) = \neg \mathrm{Split}_{\mathsf{MSO}}(c_1, r)$,
  4) $\mathrm{Split}_{\mathsf{MSO}}(\exists_{\mathsf{v}} \mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \mathrm{Split}_{\mathsf{MSO}}(c_1^{[x \mapsto v_i]}, r)) \vee \exists_{\mathsf{V}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1, r)$,
  5) $\mathrm{Split}_{\mathsf{MSO}}(\exists_{\mathsf{e}} \mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}_{\mathsf{MSO}}(c_1^{[x \mapsto e_i]}, r)) \vee \exists_{\mathsf{e}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{inc}(c_1, r, x))$,

     where

     $$\begin{aligned}\mathrm{inc}(c_1, r, x) = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} (\bigvee_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) = \mathsf{v_i} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1^{[\mathsf{s}(\mathsf{x}) \mapsto \mathsf{v_i}, \mathsf{t}(\mathsf{x}) \mapsto \mathsf{v_j}]}, r)) \\ \vee (\mathsf{s}(\mathsf{x}) = \mathsf{v_i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t}(\mathsf{x}) \neq \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1^{[\mathsf{s}(\mathsf{x}) \mapsto \mathsf{v_i}]}, r)) \\ \vee (\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) \neq \mathsf{v_j} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v_i} \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1^{[\mathsf{t}(\mathsf{x}) \mapsto \mathsf{v_i}]}, r)) \\ \vee (\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) \neq \mathsf{v_i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t}(\mathsf{x}) \neq \mathsf{v_j} \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1, r))\end{aligned}$$

  6) $\mathrm{Split}_{\mathsf{MSO}}(\exists_{\mathsf{I}} \mathsf{x}(c_1), r) = \exists_{\mathsf{I}} \mathsf{x}(\mathrm{Split}_{\mathsf{MSO}}(c_1, r))$
  7) $\mathrm{Split}_{\mathsf{MSO}}(\exists_{\mathsf{V}} \mathsf{X}(c_1), r) = \exists_{\mathsf{V}} \mathsf{X}(\bigwedge_{i=1}^{2^n} d_i \Rightarrow \mathrm{Split}_{\mathsf{MSO}}(c_1, r))$
  8) $\mathrm{Split}_{\mathsf{MSO}}(\exists_{\mathsf{E}} \mathsf{X}(c_1), r) = \exists_{\mathsf{E}} \mathsf{X}(\bigwedge_{i=1}^{2^m} a_i \Rightarrow \mathrm{Split}_{\mathsf{MSO}}(c_1, r))$

where $c^{[a \mapsto b]}$ for a variable or function a and constant b represents the condition c after the replacement of all occurrence of a with b. □

Example 6.2. Let us consider the rule $\mathsf{copy} = \langle r_e, \Gamma_3 \rangle$ of Figure 6.1 and MSO formulas:

a) $c_1 \equiv \forall_{\mathsf{v}} \mathsf{x}(\mathsf{m_v}(\mathsf{x}) = \mathsf{none})$,

b) $c_2 \equiv \exists_{\mathsf{V}} \mathsf{X}(\forall_{\mathsf{v}} \mathsf{x}(\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m_v}(\mathsf{x}) = \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$, and

c) $c_3 \equiv \exists_{\mathsf{V}} \mathsf{X}(\neg \exists_{\mathsf{v}} \mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \exists_{\mathsf{I}} \mathsf{n}(\mathsf{card}(\mathsf{X}) = 2 * \mathsf{n}))$

From the definition of Split_{MSO},

a) $\text{Split}_{\mathsf{MSO}}(c_1, r) = \neg(\mathsf{m_v}(1) \neq \mathsf{none}) \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none})$

b) $\text{Split}_{\mathsf{MSO}}(c_2, r) = \exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg(1 \in \mathsf{X} \wedge \mathsf{m_v}(1) \neq \mathsf{none})$
$\wedge\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$
$\wedge(1 \notin \mathsf{X} \Rightarrow \neg(1 \in \mathsf{X} \wedge \mathsf{m_v}(1) \neq \mathsf{none}))$
$\wedge\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2))$

c) $\text{Split}_{\mathsf{MSO}}(c_3, r) = \exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathsf{d} \wedge \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}) \wedge (1 \notin \mathsf{X} \Rightarrow \mathsf{d} \wedge \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n}))$
where
$d = \neg(1 \notin \mathsf{X} \wedge \mathsf{m_v}(1) = \mathsf{grey}) \wedge \neg(\mathsf{m_v}(1) \neq \mathsf{grey} \wedge 1 \in \mathsf{X})$
$\wedge\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))$

Lemma 6.3. Let us consider a condition c and an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, sharing no variables with c. For a host graph G, let $g : L \rightarrow G$ be a premorphism. Then,

$$G \vDash c \text{ if and only if } \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r).$$

Proof. Here, we prove the lemma inductively. The texts above the symbol $\Leftrightarrow$ below refer to lemmas that imply the associated implication, e.g. L4 refers to Lemma 4.

(Base case).

1) If c is a first order formula,
$G \vDash c \overset{\text{L5.4}}{\Leftrightarrow} \rho_g(G) \vDash \text{Split}_{\mathsf{FO}}(c, r)$
$\Leftrightarrow \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r)$

2) If c is in the form $\mathsf{x} \in \mathsf{X}$ or $x \otimes \mathsf{card}(\mathsf{X})$ for $\otimes \in \{=, \neq, <, \leq, >, \geq\}$,
$G \vDash c \overset{\text{L3.28}}{\Leftrightarrow} \rho_g(G) \vDash c$
$\Leftrightarrow \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r)$

(Inductive case).

Assuming that for some conditions c_1 and c_2 over L, the lemma holds.

1) $G \vDash c_1 \vee c_2 \overset{\text{F3.1}}{\Leftrightarrow} G \vDash c_1 \vee G \vDash c_2$
$\Leftrightarrow \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c_1, r) \vee \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c_2, r)$
$\overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c_1, r) \vee \text{Split}_{\mathsf{MSO}}(c_2, r)$

2) $G \vDash c_1 \wedge c_2 \overset{\text{F3.1}}{\Leftrightarrow} G \vDash^{\alpha} c_1 \wedge G \vDash^{\alpha} c_2$ for some assignment α
$\Leftrightarrow \rho_g(G) \vDash^{\beta} \text{Split}_{\mathsf{MSO}}(c_1, r) \vee \rho_g(G) \vDash^{\beta} \text{Split}_{\mathsf{MSO}}(c_2, r)$
where $\beta(x) = \alpha(x)$ if $x \notin V_L$; $\beta(x) = g^{-1}(\alpha(x))$ otherwise
$\overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c_1, r) \vee \text{Split}_{\mathsf{MSO}}(c_2, r)$

3) $G \vDash \neg\, c_1 \overset{\text{F3.1}}{\Leftrightarrow} \neg(G \vDash^{\alpha} c_1)$ for some assignment α
$\Leftrightarrow \neg(\rho_g(G) \vDash^{\beta} \text{Split}_{\mathsf{MSO}}(c_1, r))$
where $\beta(x) = \alpha(x)$ if $x \notin V_L$; $\beta(x) = g^{-1}(\alpha(x))$ otherwise
$\overset{\text{F3.1}}{\Leftrightarrow} \rho_g(G) \vDash \neg\text{Split}_{\mathsf{MSO}}(c_1, r)$

4) $G \vDash \exists_{\mathsf{v}}\mathsf{x}(c_1) \overset{\text{L3.30}}{\Leftrightarrow} G \vDash \bigvee_{\mathsf{i}=1}^{\mathsf{n}} c_1{}^{[\mathsf{x} \mapsto \mathsf{v_i}]} \vee \exists_{\mathsf{v}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1})$

$$\Leftrightarrow \quad \rho_g(G) \vDash \bigvee_{i=1}^{n} \mathrm{Split}_{\mathsf{MSO}}(c_1^{[x \mapsto v_i]}, r) \vee \exists_V x(\bigwedge_{i=1}^{n} x \neq v_i \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1, r))$$

5) $G \vDash \exists_e x(c_1)$
$\overset{L3.30}{\Leftrightarrow}$ $G \vDash \bigvee_{i=1}^{m} c_1{}^{[x \mapsto e_i]} \vee \exists_V x(\bigwedge_{i=1}^{m} x \neq e_i \wedge c_1)$
$\overset{L3.30}{\Leftrightarrow}$ $\rho_g(G) \vDash \bigvee_{i=1}^{m} \mathrm{Split}_{\mathsf{MSO}}(c_1^{[x \mapsto e_i]}, r) \vee \exists_e x(\bigwedge_{i=1}^{m} x \neq e_i \wedge \mathrm{Split}_{\mathsf{MSO}}(c_1, r))$
$\overset{L3.30}{\Leftrightarrow}$ $\rho_g(G) \vDash \bigvee_{i=1}^{m} \mathrm{Split}_{\mathsf{MSO}}(c_1^{[x \mapsto e_i]}, r) \vee \exists_e x(\bigwedge_{i=1}^{m} x \neq e_i \wedge \mathrm{inc}(c_1, r, x))$

6) $G \vDash \exists_I x(c_1)$
$\overset{F3.1}{\Leftrightarrow}$ $G \vDash c_1$
$\Leftrightarrow$ $\rho_g(G) \vDash \mathrm{Split}_{\mathsf{MSO}}(c_1, r)$
$\overset{F3.1}{\Leftrightarrow}$ $\rho_g(G) \vDash \exists_I x(\mathrm{Split}_{\mathsf{MSO}}(c_1, r))$

7) $G \vDash \exists_V X(c_1)$
$\overset{L3.30}{\Leftrightarrow}$ $G \vDash \exists_V X(\bigvee_{i=0}^{2^n} V_i \subseteq X \wedge \bigwedge_{j \in V - V_i} j \notin X \Rightarrow c_1)$
$\overset{F3.1}{\Leftrightarrow}$ $G \vDash \bigvee_{i=0}^{2^n} V_i \subseteq X \wedge \bigwedge_{j \in V - V_i} j \notin X \Rightarrow c_1$
$\Leftrightarrow$ $\rho_g(G) \vDash \bigvee_{i=0}^{2^n} V_i \subseteq X \wedge \bigwedge_{j \in V - V_i} j \notin X \Rightarrow \mathrm{Split}_{\mathsf{MSO}}(c_1, r)$
$\overset{F3.1}{\Leftrightarrow}$ $\rho_g(G) \vDash \exists_V X(\bigwedge_{i=0}^{2^n} (V_i \subseteq X \wedge \bigwedge_{j \in V - V_i} j \notin X \Rightarrow \mathrm{Split}_{\mathsf{MSO}}(c_1, r)))$
$\Leftrightarrow$ $\rho_g(G) \vDash \exists_V X(\bigwedge_{i=0}^{2^n} (d_i \Rightarrow \mathrm{Split}_{\mathsf{MSO}}(c_1, r)))$

8) analogous to point 7

□

The transformation Val for MSO formulas basically have the same functions as the one for FO formulas, but with additional valuation for implications we obtained from Split. Then for condition Dang, we can use the condition Dang(r) we defined in the previous chapter.

Definition 6.4 (Transformation $\mathrm{Val}_{\mathsf{MSO}}$)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, a condition c over L, a host graph G, and premoprhism $g : L \rightarrow G$. Let c shares no variable with L unless c is a rule schema condition. *Valuation of c w.r.t. r*, written $\mathrm{Val}_{\mathsf{FO}}(c, r)$, is constructed by applying the following steps to c:

1. Obtain c' and c'' as defined in Definition 5.5.
2. Obtain c''' from c'' by changing every implication in the form $a \Rightarrow d$ for some subset formula a and condition d to $a \Rightarrow d^T$ where d^T is obtained from d by changing every subformula in the form $i \in X$ for $i \in V_L$ or $i \in E_L$ and set variable X to true if $i \in X$ is implied by a or false otherwise.
3. Simplify c''' such that the implications (with subset formula) still preserved, there are no subformulas in the form $\neg\,\mathsf{true}$,$\neg(\neg\, a)$ $\neg(a \vee b)$, $\neg(a \wedge b)$ for some conditions a, b. □

Example 6.3. Let d_1, d_2, and d_3 be the conditions $\mathrm{Split}_{\mathsf{MSO}}(c_1, r_3)$, $\mathrm{Split}_{\mathsf{MSO}}(c_2, r_3)$, and $\mathrm{Split}_{\mathsf{MSO}}(c_3, r_3)$ from Example 6.2. Then,

1. $\mathrm{Val}_{\mathsf{MSO}}(d_1, r_3) = \neg\exists_v x(x \neq 1 \wedge m_v(x) \neq \mathsf{none})$
   Here, since d_1 is a first-order logic, we use the same steps as in the previous chapter.

2. $\mathrm{Val}_{\mathsf{MSO}}(d_2, r_3) = \exists_{\mathsf{V}} X(\exists_{\mathsf{V}} \mathsf{X}((1 \in \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none} \wedge \mathsf{card}(\mathsf{X}) \geq 2)))$
$\wedge ((1 \notin X \Rightarrow \neg\exists_{\mathsf{v}} x(x \neq 1 \wedge x \in X \wedge \mathsf{m_v}(x) \neq none \wedge card(X) \geq 2))))$
Here, we change every $\mathsf{m_v}(1) \neq \mathsf{none}$ to false because $m_L^V(1) = none$. Also, we change $1 \in \mathsf{X}$ on the right-hand side of the first implication to true and the second implication to false. Finally, we simplify the obtained condition.

3. $\mathrm{Val}_{\mathsf{MSO}}(d_3, r_3) =$
$\exists_{\mathsf{V}} \mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathsf{false})$
$\wedge (1 \notin \mathsf{X} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \wedge \mathsf{card}(\mathsf{X})))$
Here, we change every $\mathsf{m_v}(1) \neq \mathsf{grey}$ to true and $\mathsf{m_v}(1) = \mathsf{grey}$ to false because $m_L^V = none$. Them we change $1 \in \mathsf{X}$ on the right-hand side of the first implication to true, and false for the second implication. Similarly, we change $1 \notin \mathsf{X}$ on the right-hand side of the first implication to false, and true for the second implication. before we finally simplify the obtained condition.

Lemma 6.5. Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, a host graph G, and an injectiva morphism $g : L^\alpha \rightarrow G$ for a label assignment α_L. For a graph condition c,

$$\rho_g(G) \vDash c \text{ implies } \rho_g(G) \vDash (\mathrm{Val}_{\mathsf{MSO}}(c, r))^\alpha$$

Proof. From Lemma 5.6, we understand that the first step of $\mathrm{Val}_{\mathsf{MSO}}$ does not change the semantics of the condition on $\rho_g(G)$. The second steps gives us an implication because of the meaning of implication. Finally, the simplification of a formula will not change its semantics. □

The transformation Lift basically only conjunct the conditions we established from the previous transformations. So there is no important change from the previous chapter.

Definition 6.6 (Transformation Lift)**.** Let us consider a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. Let c be a precondition. A left application condition w.r.t. c and w, denoted by $\mathrm{Lift}_{\mathsf{MSO}}(c, w)$, is the condition over L:

$$\mathrm{Lift}_{\mathsf{MSO}}(c, w) = \mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(c \wedge ac_L, r) \wedge \mathrm{Dang}(r), r).$$

□

Example 6.4. Note that $\Gamma = \mathsf{true}$ and $\mathrm{Dang}(r_3) = \mathsf{indeg}(1) = 0 \wedge \mathsf{outdeg}(1) = 0$ such that $\mathrm{Val}(\Gamma_3 \wedge \mathrm{Dang}(r_3), r_3) = \mathsf{true}$. Hence, $\mathrm{Lift}_{\mathsf{MSO}}(c, \mathsf{copy}^\vee) = \mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(c, r_3), r_3)$ for all $c = c_1, c_2, c_3$ (from Example 6.2)

Proposition 6.7 (Left-application condition for MSO formulas)**.** Let us consider a host graph G and a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow$

$K \to R\rangle$. Let c be a precondition and α_L be a label assignment such that there exists an injective morphism $g : L^\alpha \to G$. Then,

$$G \vDash c \text{ and } G \Rightarrow_{w,g,g^*} H \text{ for some host graph } H \text{ iff } \rho_g(G) \vDash (\text{Lift}_{\mathsf{MSO}}(c, w))^\alpha$$

Proof. From Lemma 6.3, we know that $G \vDash c$ implies $\rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r)$. Then $G \Rightarrow_{w,g,g^*} H$ implies $\rho_g(G) \vDash ac_L$ and the existence of natural double-pushout with match $g : L^\alpha \to G$. The latter implies the satisfaction of the dangling condition. The satisfaction of the dangling condition implies $\rho_g(G) \vDash \text{Dang}(r)$ based on Observation 5.1, such that $\rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r) \wedge ac_L \wedge \text{Dang}(r)$, and $\rho_g(G) \vDash \text{Split}_{\mathsf{MSO}}(c, r) \wedge ac_L \wedge \text{Dang}(r), r)^\alpha$ from Lemma 6.5. □

Let us consider the definition $\text{Val}_{\mathsf{MSO}}$ and $\text{Lift}_{\mathsf{MSO}}$. The first gives us restrictions on the simplification we can have, and $\text{Split}_{\mathsf{MSO}}$ gives us some forms for node and edge (set) variables such that these forms must be preserved in Lift based on the definition of $\text{Lift}_{\mathsf{MSO}}$.

Definition 6.8 (Lifted form)**.** Let us consider a rule graph L where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. Let $2^{V_L} = \{V_1, \ldots, V_{2^n}\}$ be the power set of V_L, and $d_1, \ldots, d_{2^n}$ be subset formulas of V_L w.r.t. X where for every $i = 1, \ldots, 2^n$, d_i represents V_i. Similarly, let $2^{E_L} = \{E_1, \ldots, E_{2^m}\}$ be the power set of E_L, and $a_1, \ldots, a_{2^m}$ be subset formulas of E_L w.r.t. X where for every $i = 1, \ldots, 2^m$, a_i represents E_i.

A condition c over L is in *lifted form* if c is in one of the following forms, which are defined inductively:

1. the formulas true or false
2. predicates $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}, \mathsf{atom(x)}$ for some list variable x
3. Boolean operations $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable
4. Boolean operations $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant
5. Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge (set) variable
6. Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x = s(y)}$ or $\mathsf{x = t(y)}$ for some bounded edge variable y
7. $\exists_{\mathsf{l}}\mathsf{x}(\mathsf{c_1}$ for some condition c_1 over L in lifted form

8. $\exists_{\mathsf{v}}\mathsf{x}\,(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form
9. $\exists_{\mathsf{e}}\mathsf{x}\,(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form
10. $\exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{n}}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form
11. $\exists_{\mathsf{E}}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{m}}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form
12. $c_1 \vee c_2$ for some conditions c_1, c_2 over L in lifted form
13. $c_1 \wedge c_2$ for some conditions c_1, c_2 over L in lifted form
14. $\neg c_1$ for some condition c_1 over L in lifted form

Lemma 6.9. Let us consider a precondition c and a rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$. Then, $\mathrm{Lift}_{\mathsf{MSO}}(c, r^{\vee})$ is a condition over L in lifted form.

Proof. Note that $\mathrm{Lift}_{\mathsf{MSO}}(c, r^{\vee})$ is formed from conjunctions of $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Dang}(r))$, $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(\Gamma_3, r))$, and $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(c, r))$. From the construction of $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Dang}(r))$, it always resulting the formula true. From the syntax of Γ, Γ cannot have any node or edge variable. However, we may have the predicate $\mathsf{edge(u, v\#m)}$ for some nodes u, v in L and some edge mark m. We need to change this to $\exists_{\mathsf{e}}\mathsf{x}(\mathsf{s(x)} = \mathsf{u} \wedge \mathsf{t(x)} = \mathsf{v} \wedge \mathsf{m_E(x)} = \mathsf{m})$ so that $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(\Gamma, r))$ will be in lifted form; that is form number 9 in Definition 6.8. Then, for $\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(c, r))$, we know that c is a closed formula so that every node and edge (set) variable is bounded by an existential quantifier. By the transformation split, we always have conjunction as we see in form 8, 9, 10, and 11 of Definition 6.8. Moreover, by $\mathrm{Val}_{\mathsf{MSO}}$, we always change $i \in X$ for every node or edge i in L to true or false, depends on the premise of each implication. □

6.3 From left to right-application condition

Similar to the previous chapter, to construct a right-application condition we use transformation Adj. This transformation is the trickiest transformation in obtaining a strongest liberal postcondition, because this transformation change a condition that express properties of the initial graph so that it can express properties of the final graph.

Definition 6.10 (Adjusment in MSO logic)**.** Let us consider an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ where $V_L = \{v_1, \ldots, v_n\}$, $E_L = \{e_1, \ldots, e_m\}$, $V_K = \{u_1, \ldots, u_k\}$, $V_R = \{w_1, \ldots, w_p\}$, and $E_R = \{z_1, \ldots, z_q\}$. Let $2^{V_L} = \{V_1, \ldots, V_{2^n}\}$ be the power set of V_L, and $d_1, \ldots, d_{2^n}$ be subset formulas of V_L w.r.t. X where for every $i = 1, \ldots, 2^n$, d_i represents V_i. Similarly, let $2^{V_K} = \{U_1, \ldots, U_{2^k}\}$ be the power set of V_K, and $b_1, \ldots, b_{2^k}$ be subset formulas of V_K w.r.t. X where for every $i = 1, \ldots, 2^k$, b_i represents U_i. Also, let $2^{E_L} = \{E_1, \ldots, E_{2^m}\}$

be the power set of E_L, and $a_1, \ldots, a_{2^m}$ be subset formulas of E_L w.r.t. X where for every $i = 1, \ldots, 2^m$, a_i represents E_i.

For a condition c over L in lifted form, the *adjusted* condition of c w.r.t. r is defined inductively as below, where $c_1, \ldots, c_s$ are conditions over L, for $s \geq 2^m$ and $s \geq 2^n$:

1. If c is the formulas true or false,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$
2. If c is predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$, or $\mathsf{atom(x)}$ for some list variable x,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$
3. If c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$
4. If c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$
5. If c is a Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge variable or any set variables,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \mathrm{Adj}_{\mathsf{FO}}(c, r)$
6. If c is a Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x = s(y)}$ or $\mathsf{x = t(y)}$ for some bounded edge variable y,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$
7. If $c = \exists_{\mathsf{l}} \mathsf{x}(\mathsf{c_1}$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \exists_{\mathsf{l}} x(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r))$
8. If $c = \exists_{\mathsf{v}} \mathsf{x}\,(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r_3) = \exists_{\mathsf{v}} x(\bigwedge_{i=1}^{p}, x \neq w_i \wedge \mathrm{Adj}_{\mathsf{MSO}}(c_1, r))$
9. If $c = \exists_{\mathsf{e}} \mathsf{x}\,(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \exists_{\mathsf{e}} x(\bigwedge_{i=1}^{q}, x \neq z_i \wedge \mathrm{Adj}_{\mathsf{MSO}}(c_1, r))$
10. If $c = \exists_{\mathsf{V}} \mathsf{X}(\bigwedge_{\mathsf{i=1}}^{2^{\mathsf{n}}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form or contains $\mathsf{card(X)}$
    $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \exists_{\mathsf{V}} X(\bigwedge_{v \in V_R - V_K} v \notin X \bigwedge_{i=1}^{2^k}(b_i \Rightarrow \bigvee_{j \in W_i} c'_j))$
    where $c'_j = \mathrm{Adj}_{\mathsf{MSO}}(c_j, r)^{[\mathsf{card(X)} \mapsto \mathsf{card(X)} + |(\mathsf{V_L} - \mathsf{V_K}) \cap \mathsf{V_j}|]}$ and for $i = 1, \ldots, 2^k$, W_i is a subset of $\{1, \ldots, 2^n\}$ such that for all $j \in \{1, \ldots, 2^n\}$, $j \in W_i$ iff d_j implies b_i

11. If $c = \exists_{\mathsf{E}} \mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{m}}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form, construction of $\mathrm{Adj}_{\mathsf{MSO}}(c, r)$ is analogous to point 10

12. If $c = c_1 \vee c_2$ for some conditions c_1, c_2 over L in lifted form, $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \mathrm{Adj}_{\mathsf{MSO}}(c_1, r) \vee \mathrm{Adj}_{\mathsf{MSO}}(c_2, r)$

13. If $c = c_1 \wedge c_2$ for some conditions c_1, c_2 over L in lifted form, $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \mathrm{Adj}_{\mathsf{MSO}}(c_1, r) \wedge \mathrm{Adj}_{\mathsf{MSO}}(c_2, r)$

14. If $c = \neg c_1$ for some condition c_1 over L in lifted form, $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \neg \mathrm{Adj}_{\mathsf{MSO}}(c_1, r)$

Example 6.5.

Let us consider the rule copy of Figure 5.3. Let $d_1 = \mathrm{Lift}_{\mathsf{MSO}}(c_1, \mathsf{copy}^\vee)$, $d_2 = \mathrm{Lift}_{\mathsf{MSO}}(c_2, \mathsf{copy}^\vee)$, and $d_3 = \mathrm{Lift}_{\mathsf{MSO}}(c_3, \mathsf{copy}^\vee)$ from Example 6.4. Then, based on Definition 6.10,

a) $\mathrm{Adj}_{\mathsf{MSO}}(d_1, r_3) = \mathrm{adj}(d_1, r_3)$
$= \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none})$
Here, d_1 is a first-order formula so that $d_1 = d_1'$ and $\mathrm{Adj}_{\mathsf{MSO}}(d_1, r_3) = \mathrm{Adj}_{\mathsf{FO}}(d_1, r_3)$.

b) $\mathrm{Adj}_{\mathsf{MSO}}(d_2, r_3) = \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 1)$
Here, we first change $\exists_{\mathsf{V}}\mathsf{X}(1 \in \mathsf{X} \wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 2)$ to $\exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) + 1 \geq 2)$. Then, we change every $\mathsf{x} \neq 1$ to false and $1 \notin X$ to true. We then add constraint $x \neq 2 \wedge x \neq 3$ inside the node existential quantifier, and finally we simplify the obtained condition.

c) $\mathrm{Adj}_{\mathsf{MSO}}(d_3, r) = \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_v}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_v}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \wedge \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})$
Here, we change every $\mathsf{x} \neq 1$ to false and $1 \notin X$ to true. We then add constraint $x \neq 2 \wedge x \neq 3$ inside the node existential quantifier, and finally we simplify the obtained condition.

Lemma 6.11. Let us consider a host graph G, a GP 2 conditional rule schema $\langle r, \Gamma \rangle$ with $r = \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $g : L^\alpha \rightarrow G$ for some label assignment α_L, and a precondition d. Let H be a host graph such that $G \Rightarrow_{w,g,g^*} H$ for some injective morphism $g^* : R^\alpha \rightarrow H$. Let us denote by c the condition $\mathrm{Lift}_{\mathsf{MSO}}(d, r^\vee)$. Then,

$$\rho_g(G) \vDash c^\alpha \text{ implies } \rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{MSO}}(c, r))^\alpha$$

Proof. From Lemma 6.9, we know that $c = \mathrm{Lift}_{\mathsf{MSO}}(d, r^\vee)$ is in a lifted form. Here, we prove that the lemma above holds by showing by induction on lifted form that for all condition c over L in lifted form, $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{MSO}}(c, r))^\alpha$ holds. Base case.

1. if $c = \mathsf{true}$, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = \mathsf{true}$ such that $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c, r))^\alpha$ holds.

2. if c is predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$, or $\mathsf{atom(x)}$ for some list variable x, $\rho_g(G) \vDash c^\alpha$ means that there exists a list l such that c true when we substitute l for x. Since the truth value of c does not depend on $\rho_g(G)$, then $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash c = \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

3. if c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable, then the truth value of c does not depend on $\rho_g(G)$, then $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash c = \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

4. if c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_2$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant, $\rho_g(G) \vDash c^\alpha$ means that f_1 and f_2 representing the same node (for $c : f_1 = f_2$) or they are representing different node (for $c : f_1 \neq f_2$) in $\rho_g(G)$. Note that there is no node/edge constant in c such that f_1, f_2 must be node/edge variables, or the function $\mathsf{s(x)}$ or $\mathsf{t(x)}$ for some edge variable x. Since the variables must be bounded, then they are bounded by quantifier in form point 8 or 9 of Definition 6.8, so that f_1, f_2 cannot represent nodes/edges in L. Since the nodes/edges represented by f_1, f_2 is in $\rho_g(G) - (L)$, then the nodes/edges must be in $\rho_{g^*}(H)$ so that $\rho_{g^*}(H) \vDash c = \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

5. if c is a Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge variable or any set variables, this means that c is a first-order formula. Since we have shown that $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{FO}}(c, r)^\alpha$ (for first-order lifted formula) in Lemma 5.10, then $\rho_g(G) \vDash c^\alpha$ implies $\mathrm{Adj}_{\mathsf{MSO}}(c, r)^\alpha$.

6. if c is a Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x} = \mathsf{s(y)}$ or $\mathsf{x} = \mathsf{t(y)}$ for some bounded edge variable y, $\rho_g(G) \vDash c^\alpha$ implies that there is a node/edge v and a node/set variable V such that $v \in V$. Since x is bounded, by point 8 of Definition 6.8, v is not in L so that v must be in $\rho_g(G) - L$. Hence, v is in $\rho_{g^*}(H)$ so that $\rho_{g^*}(H) \vDash x \in X = \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

Inductive case. Assume that for conditions c_i over L (for $i = 1, \ldots, s$ where $s \geq 2^n$ and $s \geq 2^m$) in lifted form, it is true that $\rho_g(G) \vDash c_i^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_i, r)^\alpha$. Then,

1. if $c = \exists \mathsf{I} \mathsf{x}(\mathsf{c_1}$ for some condition c_1 over L in lifted form, $\rho_g(G) \vDash c^\alpha$ implies that there exists a list i such that c_1^α is true in $\rho_g(G)$ when we substitute i for x. From the assumption, $\mathrm{Adj}_{\mathsf{MSO}}(c_1, r)^\alpha$ is true in $\rho_{g^*}(H)$ when we substitute i for x. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

2. if $c = \exists_{\mathsf{v}}\mathsf{x}\,(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form, $\rho_g(G) \vDash c$ implies that there is a node v in $\rho_g(G)$ such that v is not in L and c_1^α is true in $\rho_g(G)$ when we substitute v for x. From the assumption, the latter implies $\mathrm{Adj}_{\mathsf{MSO}}(c_1, r)$ is true in $\rho_{g^*}(H)$ when we substitute v for x. Since v is not in L, then v in in $\rho_g(G) - L$ which means v is in $\rho_{g^*}(H) - R$, so that $\bigwedge_{i=1}^{p} v \neq w_i$ must be true in $\rho_{g^*}(H)$. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_1, r))$.

3. if $c = \exists_{\mathsf{e}}\mathsf{x}\,(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1})$ for some condition c_1 over L in lifted form, $\rho_g(G) \vDash c$ implies that there is an edge e in $\rho_g(G)$ such that e is not in L and c_1^α is true in $\rho_g(G)$ when we substitute e for x. From the assumption, the latter implies $\mathrm{Adj}_{\mathsf{MSO}}(c_1, r)$ is true in $\rho_{g^*}(H)$ when we substitute e for x. Since e is not in L, then e in in $\rho_g(G) - L$ which means e is in $\rho_{g^*}(H) - R$, so that $\bigwedge_{i=1}^{q} e \neq z_i$ must be true in $\rho_{g^*}(H)$. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_1, r))$.

4. if $c = \exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{n}}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$, $\rho_g(G) \vDash c^\alpha$ implies that there exists a set node A subset of $V_{\rho_g(G)}$ such that for all $i = 1, \ldots, 2^n$, $d_i \Rightarrow c_i$ is true in $\rho_g(G)$.
   If c_i does not contain $\mathsf{card(X)}$, then c_i is in lifted form so that if substituting A for X in d_i yields true, $\rho_g(G) \vDash c_i$. From assumption, we know that $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_i)$. Note that from the rule of inference, if $a \wedge b$ implies c and $a \wedge \neg b$ implies d, then a implies $c \vee d$. Hence, if substituting A for X is true in b_j, $\rho_{g^*}(H) \vDash \bigvee_{j \in W_i} \mathrm{Adj}_{\mathsf{MSO}}(c_j, r)$.
   Similar reasoning happens when c_i contains $\mathsf{card(X)}$. However, the value of $\mathsf{card(X)}$ may changes, depends on d_i. If d_i implies some nodes in $L - K$ being members of A, then the value of $\mathsf{card(X)}$ decreases as many as $|(V_L - V_K) \cap V_j|$. Hence, the value of $\mathsf{card(A)}$ in $\rho_g(G)$ is the same as the value of $\mathsf{card(A) + |(V_L - V_K) \cap V_j|}$ in $\rho_{g^*}(H)$.

5. If $c = \exists_{\mathsf{E}}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{m}}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form,
   the proof is analogous to point 10

6. If $c = c_1 \vee c_2$, $\rho_g(G) \vDash c^\alpha$ iff $\rho_g(G) \vDash c_1^\alpha$ or $\rho_g(G) \vDash c_2^\alpha$. From the assumption, $\rho_g(G) \vDash c_1^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_1, r)^\alpha$, and $\rho_g(G) \vDash c_2^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_2, r)^\alpha$. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

7. If $c = c_1 \wedge c_2$, $\rho_g(G) \vDash c^\alpha$ iff $\rho_g(G) \vDash {}^\beta c_1^\alpha$ and $\rho_g(G)^\beta \vDash c_2^\alpha$ for some assignment β. From the assumption, $\rho_g(G) \vDash {}^\beta c_1^\alpha$ implies $\rho_{g^*}(H) \vDash {}^\beta\mathrm{Adj}_{\mathsf{MSO}}(c_1, r)^\alpha$, and $\rho_g(G) \vDash {}^\beta c_2^\alpha$ implies $\rho_{g^*}(H) \vDash {}^\beta\mathrm{Adj}_{\mathsf{MSO}}(c_2, r)^\alpha$. This means, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c_1, r)^\alpha \wedge \mathrm{Adj}_{\mathsf{MSO}}(c_2, r)^\alpha$. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c, r)$.

8. If $c = \neg c_1$, $\rho_g(G) \vDash c^\alpha$ implies $\rho_g(G) \vDash {}^\beta c_1^\alpha$ is false for some assignment β. From assumption, it can imply $\neg\rho_{g^*}(H) \vDash {}^\beta\mathrm{Adj}_{\mathsf{MSO}}(c_1, r)^\alpha$, such that $\rho_{g^*}(H) \vDash \mathrm{Adj}_{\mathsf{MSO}}(c, r)^\alpha$.

□

Lemma 6.12. Let us consider a host graph G, a GP 2 conditional rule schema $\langle r, \Gamma \rangle$ with $r = \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $g : L^\alpha \rightarrow G$ for some label assignment α_L, and a precondition d. Let us denote by c the condition $\mathrm{Lift}_{\mathsf{MSO}}(d, r^\vee)$. Then,

$$\rho_g(G) \vDash c^\alpha \text{ implies } \rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1})^\alpha$$

Proof. Here, we prove the lemma by induction on lifted form where *prop* denotes the statement $\rho_g(G) \vDash c^\alpha$ implies $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1})^\alpha$.
Base case.

1. if $c = \mathsf{true}$ or $c = \mathsf{false}$, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$ and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = c$. Hence, *prop* holds.

2. if c is predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$, or $\mathsf{atom(x)}$ for some list variable x, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$ and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = c$. Hence, *prop* holds.

3. if c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_1$ where each f_1 and f_2 are terms representing a list and neither contains free node/edge variable, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$ and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = c$. Hence, *prop* holds.

4. if c is a Boolean operation $f_1 = f_2$ or $f_1 \neq f_2$ where each f_1 and f_2 are terms representing a node (or edge) and neither contains free node/edge variable or node/edge constant, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$ and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = c$. Hence, *prop* holds.

5. if c is a Boolean operation $f_1 \diamond f_2$ for $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ and some terms f_1 and f_2 representing integers and neither contains free node/edge variable or any set variables, this means that c is a first-order formula. Since we have shown that *prop* holds for first-order lifted formula c, *prop* holds.

6. if c is a Boolean operation $\mathsf{x} \in \mathsf{X}$ for a bounded set variable X and bounded edge variable x, or a bounded set variable X and a bounded node variable x, $\mathsf{x} = \mathsf{s(y)}$ or $\mathsf{x} = \mathsf{t(y)}$ for some bounded edge variable y, then $\mathrm{Adj}_{\mathsf{MSO}}(c, r) = c$ and $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = c$. Hence, *prop* holds.

Inductive case. Assume that for conditions c_i over L (for $i = 1, \ldots, s$ where $s \geq 2^n$ and $s \geq 2^m$) in lifted form, it is true that $\rho_g(G) \vDash c_i^\alpha$ iff $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_i, r), r^{-1})^\alpha$. Then,

1. if $c = \exists_{\mathsf{I}}\mathsf{x}(\mathsf{c_1}$ for some condition c_1 over L in lifted form,
   $\rho_g(G) \vDash c^\alpha$ implies that there exists a list i such that c_1^α is true in $\rho_g(G)$ when we substitute i for x. From the assumption, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1})^\alpha$ is true in $\rho_{g^*}(H)$ when we substitute i for x. Hence, *prop* holds.

2. if $c = \exists_{\mathsf{v}} \mathsf{x} \, (\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1})$,
   $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = \exists_{\mathsf{v}} \mathsf{x} \big(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(\mathsf{c_1}, \mathsf{r}), \mathsf{r}^{-1})\big)$. From the assumption, $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1}))$ implies $\rho_g(G) \vDash c_1$ so that *prop* holds.

3. if $c = \exists_{\mathsf{e}} \mathsf{x} \, (\bigwedge_{\mathsf{i}=1}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1})$,
   $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = \exists_{\mathsf{e}} \mathsf{x} \big(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(\mathsf{c_1}, \mathsf{r}), \mathsf{r}^{-1})\big)$. From the assumption, $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1}))$ implies $\rho_g(G) \vDash c_1$ so that *prop* holds.

4. if $c = \exists_{\mathsf{V}} \mathsf{X} (\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{n}}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$,

5. If $c = \exists_{\mathsf{E}} \mathsf{X} (\bigwedge_{\mathsf{i}=1}^{2^{\mathsf{m}}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form,
   the proof is analogous to point 10

6. If $c = c_1 \vee c_2$,
   $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1}) \vee \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_2, r), r^{-1})$.
   From assumption, $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1})$ iff $\rho_g(G) \vDash c_1 \vee c_2 = c$.

7. If $c = c_1 \wedge c_2$,
   $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1}) \wedge \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_2, r), r^{-1})$.
   From assumption, $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1})$ iff $\rho_g(G) \vDash c_1 \wedge c_2 = c$.

8. If $c = \neg c_1$,
   $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1}) = \neg \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c_1, r), r^{-1})$. From assumption, $\rho_g(G) \vDash \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Adj}_{\mathsf{MSO}}(c, r), r^{-1})$ iff $\rho_g(G) \vDash \neg c_1 = c$.

□

As in the example in Section 6.1, we can construct a right application condition from a conjunction of Adj, Spec, Dang, and the given right-application condition.

Definition 6.13 (Shifting)**.** Let us consider a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, and a precondition c. Right application condition w.r.t. w, denoted by $\mathrm{Shift}(c, w)$, is defined as:

$$\mathrm{Shift}_{\mathsf{MSO}}(w) = \mathrm{Adj}_{\mathsf{MSO}}(ac_L, r) \wedge ac_R \wedge \mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1})$$

□

Example 6.6. Let us consider d_1, d_2, d_3 as defined in Example 6.5, also r_3 of Figure 5.3. Let $w_1 = \langle r_3, d_1, \mathsf{true} \rangle$, $w_1 = \langle r_3, d_2, \mathsf{true} \rangle$, and $w_1 = \langle r_3, d_3, \mathsf{true} \rangle$. Then, based on Definition 6.13,

$$
\begin{aligned}
\text{a)}\quad \mathrm{Shift}_{\mathsf{MSO}}(w_1) \;=\;& \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x}\neq 2\wedge \mathsf{x}\neq 3\wedge \mathsf{m_v}(\mathsf{x})\neq \mathsf{none})\\
&\wedge \mathsf{m_v}(2)=\mathsf{grey}\wedge \mathsf{m_v}(3)=\mathsf{grey}\wedge \mathsf{l_v}(2)=\mathsf{a}\wedge \mathsf{l_v}(3)=\mathsf{a}\\
&\wedge \mathsf{indeg}(2)=0\wedge \mathsf{indeg}(3)=0\wedge \mathsf{outdeg}(2)=0\wedge \mathsf{outdeg}(3)=0\\
\text{b)}\quad \mathrm{Shift}_{\mathsf{MSO}}(w_2) \;=\;& \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x}\neq 2\wedge \mathsf{x}\neq 3\wedge \mathsf{x}\in\mathsf{X}\wedge \mathsf{m_v}(\mathsf{x})\neq \mathsf{none})\wedge \mathsf{card}(\mathsf{X})\geq 1)\\
&\wedge \mathsf{m_v}(2)=\mathsf{grey}\wedge \mathsf{m_v}(3)=\mathsf{grey}\wedge \mathsf{l_v}(2)=\mathsf{a}\wedge \mathsf{l_v}(3)=\mathsf{a}\\
&\wedge \mathsf{indeg}(2)=0\wedge \mathsf{indeg}(3)=0\wedge \mathsf{outdeg}(2)=0\wedge \mathsf{outdeg}(3)=0\\
\text{c)}\quad \mathrm{Shift}_{\mathsf{MSO}}(w_3) \;=\;& \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x}\neq 2\wedge \mathsf{x}\neq 3\\
&\qquad\wedge((\mathsf{x}\notin\mathsf{X}\wedge \mathsf{m_v}(\mathsf{x})=\mathsf{grey})\vee(\mathsf{m_v}(\mathsf{x})\neq\mathsf{grey}\wedge\mathsf{x}\in\mathsf{X})))\\
&\quad\wedge\ \mathsf{card}(\mathsf{X})=2^{*}\mathsf{n})\\
&\wedge \mathsf{m_v}(2)=\mathsf{grey}\wedge \mathsf{m_v}(3)=\mathsf{grey}\wedge \mathsf{l_v}(2)=\mathsf{a}\wedge \mathsf{l_v}(3)=\mathsf{a}\\
&\wedge \mathsf{indeg}(2)=0\wedge \mathsf{indeg}(3)=0\wedge \mathsf{outdeg}(2)=0\wedge \mathsf{outdeg}(3)=0
\end{aligned}
$$

Proposition 6.14 (Right-application condition for MSO formulas)**.** Let us consider a host graph G, a generalised rule $w = \langle r, ac_L, ac_R\rangle$ an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$, an injective morphism $g : L^{\alpha} \rightarrow G$ for some label assignment α_L, and a precondition d. Then for host graphs H such that $G \Rightarrow_{w,g,g^*} H$ with an right morphism $g^* : R^{\beta} \rightarrow H$ where $\beta_R(i) = \alpha_L(i)$ for every variable i in L such that i in R, and for every node (or edge) i where $m_L^V(i) = m_R^V(i) = \mathtt{any}$ (or $m_L^E(i) = m_R^E(i) = \mathtt{any}$),

$$\rho_{g^*}(H) \vDash (\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(d, w)), r)^{\beta} \text{ iff } \rho_{g^*}(H) \vDash (\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(d, w), r), ac_R\rangle))^{\beta}$$

Proof. From the semantics of conjunction, we know that $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(d, w)), r)^{\beta}$ is implied by $\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(d, w)), r\rangle), ac_R))^{\beta}$, so now we show that $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(d, w)), r)^{\beta}$ implies $\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(d, w)), r), ac_R\rangle))^{\beta}$. From the proof of Proposition 5.13, we know that $\rho_{g^*}(H) \vDash ac_R^{\beta} \wedge \mathrm{Spec}(R)^{\beta} \wedge\ \mathrm{Dang}(r^{-1})^{\beta}$. □

6.4 From right-application condition to postcondition

The right-application condition we obtain from transformation Shift is strong enough to express properties of the replacement graph of any resulting graph. To turn the condition obtained from Shift to a postcondition, we can use Post we defined in the previous Chapter (see Definition 5.14) because we only need to turn a condition over right-hand graph to a postcondition. In $\mathrm{Lift}_{\mathsf{MSO}}$ or $\mathrm{Shift}_{\mathsf{MSO}}$, we never have a set of nodes/edges as a constant. Hence, we will have set variables as what we have in the precondition, so that the process of turning the right-application condition to a postcondition must be the same with the one in FOL.

Example 6.7. Let s_1, s_2, s_3 (resp.) be $\mathrm{Shift}_{\mathsf{MSO}}(\langle r_3, \mathrm{Lift}_{\mathsf{MSO}}(c_1, \mathsf{copy}^{\vee}), \mathsf{true}\rangle)$, $\mathrm{Shift}_{\mathsf{MSO}}(\langle r_3, \mathrm{Lift}_{\mathsf{MSO}}(c_2, \mathsf{copy}^{\vee}), \mathsf{true}\rangle)$, and $\mathrm{Shift}_{\mathsf{MSO}}(\langle r_3, \mathrm{Lift}_{\mathsf{MSO}}(c_3, \mathsf{copy}^{\vee}), \mathsf{true}\rangle)$ as obtained

in Example 6.6 for c_1, c_2, c_3 defined in Example 6.2 and $\mathsf{copy} = \langle r_3, \Gamma_3 \rangle$ of Figure 5.3. Then, based on Definition 5.14,

a) $\mathrm{Post}(s_1) = \exists_{\mathsf{v}}\mathsf{y}, \mathsf{z}(\exists_{\mathsf{l}}\mathsf{a}($
$\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none})$
$\wedge \mathsf{m}_{\mathsf{v}}(\mathsf{y}) = \mathsf{grey} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{z}) = \mathsf{grey} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{y}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{z}) = \mathsf{a}$
$\wedge \mathsf{indeg}(\mathsf{y}) = 0 \wedge \mathsf{indeg}(\mathsf{z}) = 0 \wedge \mathsf{outdeg}(\mathsf{y}) = 0 \wedge \mathsf{outdeg}(\mathsf{z}) = 0))$

b) $\mathrm{Post}(s_2) = \exists_{\mathsf{v}}\mathsf{y}, \mathsf{z}(\exists_{\mathsf{l}}\mathsf{a}($
$\exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z} \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none}) \wedge \mathsf{card}(\mathsf{X}) \geq 1)$
$\wedge \mathsf{m}_{\mathsf{v}}(\mathsf{y}) = \mathsf{grey} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{z}) = \mathsf{grey} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{y}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{z}) = \mathsf{a}))$
$\wedge \mathsf{indeg}(\mathsf{y}) = 0 \wedge \mathsf{indeg}(\mathsf{z}) = 0 \wedge \mathsf{outdeg}(\mathsf{y}) = 0 \wedge \mathsf{outdeg}(\mathsf{z}) = 0$

c) $\mathrm{Post}(s_3) = \exists_{\mathsf{v}}\mathsf{y}, \mathsf{z}(\exists_{\mathsf{l}}\mathsf{a}($
$\exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z}$
$\wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))$
$\wedge\ \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})$
$\wedge \mathsf{m}_{\mathsf{v}}(\mathsf{y}) = \mathsf{grey} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{z}) = \mathsf{grey} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{y}) = \mathsf{a} \wedge \mathsf{l}_{\mathsf{v}}(\mathsf{z}) = \mathsf{a}$
$\wedge \mathsf{indeg}(\mathsf{y}) = 0 \wedge \mathsf{indeg}(\mathsf{z}) = 0 \wedge \mathsf{outdeg}(\mathsf{y}) = 0 \wedge \mathsf{outdeg}(\mathsf{z}) = 0))$

To obtain a closed MSO formula from the obtained right-application condition, we only need to variablise the node/edge constants in the right-application condition, then put an existential quantifier for each free variable in the resulting FO formula.

Theorem 6.15 (Post)**.** Let us consider a precondition c and a conditional rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$. Then, $\mathrm{Post}(\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c, r^{\vee}), \mathsf{true}\rangle))$ is a strongest liberal postcondition w.r.t. c and r.

Proof. The proof is really similar to the proof of Theorem 5.16. However, instead of using Proposition 5.8, Lemma 5.10, Lemma 5.11, and Proposition 5.13, we use (resp.) Proposition 6.7, Lemma 6.11, Lemma 6.12, and Proposition 6.14. □

Now, we can obtain a strongest liberal postcondition over a given precondition and conditional rule schema, where the precondition is a closed monadic second-order formula. Here, we use the notation $\mathrm{Slp}(c, r)$ for a strongest liberal postcondition that is obtained from the construction we obtained here in this chapter.

Definition 6.16 ($\mathrm{Slp}(c, r)$, $\mathrm{Slp}(c, r^{-1})$)**.** Let us consider a conditional rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$ and a precondition c, which is a closed monadic second-order formula. A strongest liberal precondition over c and r, also over c and r^{-1}, (resp.) defined as:

$$\mathrm{Slp}(c, r) = \mathrm{Post}(\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c, \langle r, \Gamma^{\vee}, \mathsf{true}\rangle), \mathsf{true}\rangle)).$$

$$\mathrm{Slp}(c, r^{-1}) = \mathrm{Post}(\mathrm{Shift}_{\mathsf{MSO}}(\langle r, \mathrm{Lift}_{\mathsf{MSO}}(c, \langle r, \mathsf{true}, \Gamma^{\vee}\rangle), \Gamma^{\vee}\rangle)).\square$$

6.5 Complexity of a strongest liberal postcondition

Weakest liberal preconditions produced in [1, 17] can produce unwieldy expressions. Poskitt in [1] mentions that in the worst-case, a factorial blow-up can result in obtaining a weakest liberal precondition. Similarly, our approach may blow-up the size of condition when we compute a strongest liberal postcondition.

In our approach, the transformation Split gives the worst blow-up because it considers all possibilities of variables in the given precondition expressing an element in the matching. Also, we need to consider all possibilities of nodes (or edges) in the matching to become a member of node (or edge) sets that are represented by node (or set) variables in the given precondition.

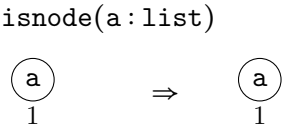


FIGURE 6.2: Rule schema `isnode`

Example 6.8. Let us consider the rule `isnode` of Figure 6.2. Let $\mathrm{P}(n)$ for $n = 1, 2, 3, \ldots$ be formula $\exists_v x_1, \ldots, x_n(\mathsf{l}_v(x_1) < 1 \wedge \ldots \wedge \mathsf{l}_v(x_n) < n)$. Table 6.4 shows us the obtained $\mathrm{Slp}(\mathrm{P}(n),\texttt{isnode})$ for some $n \in \{1, 2, 3, 4\}$. From the table, we can see that we have an exponential blow-up with respect to the number of node variables we have in the precondition.

TABLE 6.4: Strongest liberal postcondition over $\mathrm{P}(n)$ and `isnode`

$\mathrm{Slp}(\mathrm{P}(n),\texttt{isnode}) = \exists_v y(\exists_l a(\mathsf{m}_v(y) = \mathsf{none} \wedge \mathsf{l}_v(y) = a \wedge \neg\mathsf{root}(y) \wedge \mathrm{S}(n)))$	
n	$\mathrm{S}(n)$
1	$a < 1 \vee \exists_v x_1(x_1 \neq y \wedge \mathsf{l}_v(x_1) < 1)$
2	$a < 2 \vee \exists_v x_2(x_2 \neq y \wedge a < 1 \wedge \mathsf{l}_v(x_2) < 2)$ $\vee \exists_v x_1(x_1 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge a < 2) \vee \exists_v x_2(x_2 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2)))$
3	$a < 3 \vee \exists_v x_3(x_3 \neq y \wedge a < 2 \wedge \mathsf{l}_v(x_3) < 3)$ $\vee \exists_v x_2(x_2 \neq y \wedge ((a < 3 \wedge \mathsf{l}_v(x_2) < 2) \vee \exists_v x_3(x_3 \neq y \wedge a < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_3) < 3)))$ $\vee \exists_v x_1(x_1 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge a < 3) \vee \exists_v x_3(x_3 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge a < 2 \wedge \mathsf{l}_v(x_3) < 3)$ $\vee \exists_v x_2(x_2 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge a < 3)$ $\vee \exists_v x_3(x_3 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_3) < 3)))))$
4	$a < 4 \vee \exists_v x_4(x_4 \neq y \wedge a < 3 \wedge \mathsf{l}_v(x_4) < 4)$ $\vee \exists_v x_3(x_3 \neq y \wedge ((a < 4 \wedge \mathsf{l}_v(x_3) < 3) \vee \exists_v x_4(x_4 \neq y \wedge a < 2 \wedge \mathsf{l}_v(x_3) < 3 \wedge \mathsf{l}_v(x_4) < 4)))$ $\vee \exists_v x_2(x_2 \neq y \wedge ((a < 4 \wedge \mathsf{l}_v(x_2) < 3) \vee \exists_v x_4(x_4 \neq y \wedge a < 3 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_4) < 4)$ $\vee \exists_v x_3(x_3 \neq y \wedge ((a < 4 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_3) < 3)$ $\vee \exists_v x_4(x_4 \neq y \wedge a < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_3) < 3 \wedge \mathsf{l}_v(x_4) < 4)))))$ $\vee \exists_v x_1(x_1 \neq y \wedge ((a < 4 \wedge \mathsf{l}_v(x_1) < 1) \vee \exists_v x_4(x_4 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge a < 3 \wedge \mathsf{l}_v(x_4) < 4)$ $\vee \exists_v x_3(x_3 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge a < 4 \wedge \mathsf{l}_v(x_3) < 3)$ $\vee \exists_v x_4(x_4 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge a < 2 \wedge \mathsf{l}_v(x_3) < 3 \wedge \mathsf{l}_v(x_4) < 4)$ $\vee \exists_v x_2(x_2 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge a < 4)$ $\vee \exists_v x_4(x_4 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge a < 3 \wedge \mathsf{l}_v(x_4) < 4)$ $\vee \exists_v x_3(x_3 \neq y \wedge ((\mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2) < 2 \wedge \mathsf{l}_v(x_3) < 3 \wedge a < 4)$ $\vee \exists_v x_4(x_4 \neq y \wedge \mathsf{l}_v(x_1) < 1 \wedge \mathsf{l}_v(x_2)$ $\wedge \mathsf{l}_v(x_3) < 3 \wedge \mathsf{l}_v(x_4) < 4)))))))))$

In the construction of a strongest liberal postcondition, transformation Split forms disjunction of all way variables in the precondition have connection with elements in the possible matching. If we have n variables in the precondition, the worst-case scenario would be where the variables are bounded by nested quantifiers, and where $\mathrm{Val}_{\mathsf{MSO}}$ does not simplify the condition obtained from Split.

From Table 6.4 of Example 6.8, we can see that exponential growth might limit us in computing Slp manually since it will take us too much space, and there are many brackets to be used so that human error is likely to occur.

Formal Statement 1. In the worst-case, the construction of a strongest liberal postcondition can result in an exponential blow-up with respect to the number of node variables we have in the precondition.

Now, let us consider the general case, by considering a precondition c with n node variables, m edge variables, p node set variables, and q edge set variables. Also, let L be the left-hand graph of the given rule schema and $V_L = \{v_1, \ldots, v_u\}$ and $E_L = \{e_1, \ldots, e_w\}$.

Now let us recall the definition of Split (Definition 5.3). For each node variable, the transformation forms a disjunction from all possible ways of the variable expressing nodes in the left-hand graph. Every node variable x may express $v_1, \ldots, v_u$, or none of them. Hence, for each variable, the formula is repeated $u + 1$ times. Hence, we need to check all $(u + 1)^n$ possibilities for all node variables, which result in an exponential blow-up.

For each edge variable y, we need to check all possibilities where the variables express edges in the match (if any). However, for edge variables, we also need to consider whether an edge represented by a variable is incident to a node in the match or not. Hence, for each edge variable, we repeat the given precondition $(w + 1 + (u + 1)^2)$ times, such that for all edge variables, we check $(w + 1 + (u + 1)^2)^m$ possibilities. In other words, it also results in an exponential blow-up.

Every node (or edge) in the match may also be a member of a node (or edge) set that is represented by a node (or edge) set variable. Hence, for each node (or edge) set variable, we need to consider each possible subset of V_L (or E_L) being a subset of the set variable. For this, we have 2^u (or 2^w) cases to consider for each set. Hence, in total, we have 2^{up} (or 2^{wq}) cases, so that it also gives an exponential blow-up in the result.

The condition that is obtained by transformation Split may get simplified by $\mathrm{Val}_{\mathsf{MSO}}$. However, in the worst-case, where no function or predicate in the condition expresses the obvious value with respect to L, we still get the exponential blow-up in the obtained postcondition.

Quantifiers we have may be nested with mixed type (e.g. node quantifier inside edge quantifier). However, since each kind of variable contributes in exponential blow-up, it will still give us exponential blow-up in the end.

The transformation Shift also gives another blow-up, but not as much as we have in Split. In the transformation Shift, we only need to add some conditions that express the specification of the right-hand graph of the given rule schema. Hence, it is a linear blow-up with respect to the number of elements in the right-hand graph.

6.6 Summary

This chapter shows how we can obtain a strongest liberal postcondition over a rule schema and a precondition, which is a closed monadic second-order formula. We extend the construction we have in the previous chapter to cover formulas with cardinality function, element operator, and quantifiers over set variables. The approach is basically similar, but presenting the formal definition and proof for the approach is trickier than what we present in the previous chapter. This is because the deletion of a node may affect the number of nodes in the set of nodes represented in the given precondition.

Here, we also argue that the construction may result in an exponential blow-up. This is due to the need of checking all possibilities of each node/edge variables expressing nodes/edges in a possible match of the rule schema application.

Chapter 7

Graph program verification

This chapter presents proof calculi we can use to verify graph programs. Here, we define two proof calculi: semantic and syntactic, wherein partial correctness calculus is considered. The semantic proof calculus uses assertions in semantic definition, while the syntactic proof calculus uses monadic second-order formulas as assertions.

7.1 Semantic Proof Calculus

In this section, we define a proof calculus, in the sense of partial correctness, where assertions are arbitrary functions from a graph to a Boolean value (true or false).

Definition 7.1 (Assertions)**.** A *semantic assertion* is a function $a : \mathcal{G}(\mathbb{L}) \to \{\mathsf{true}, \mathsf{false}\}$. If for a graph G, $a(G) = \mathsf{true}$, we say that G *satisfies* a and we write $G \vDash a$. □

In this thesis, we only focus on partial correctness. For a graph program P and assertions c and d, triple $\{c\}\, P\, \{d\}$ is partially correct iff for all graph satisfying c, $H \in [\![P]\!]G$ implies $H \vDash d$.

Definition 7.2 (Partial correctness [14])**.** A graph program P is *partially correct* with respect to a precondition c and a postcondition d, denoted by $\vDash \{c\}\ P\ \{d\}$ if for every host graph G and every graph H in $[\![P]\!]G$, $G \vDash c$ implies $H \vDash d$. □

To prove that $\vDash \{c\}\ P\ \{d\}$ holds for some assertions c, d, and a graph program P, we use two methods: 1) finding a strongest liberal postcondition w.r.t c and P and prove that the strongest liberal postcondition implies d, and 2) using proof rules for graph programs, create a proof tree to show the partial correctness. The first method has been used in classical programming [53, 55], while the second has been used in graph programming [1] but without the special command `break`.

In the previous chapter, we have defined a strongest liberal postcondition w.r.t. a precondition and a conditional rule schema. In this chapter, we extend the definition from conditional rule schemata to graph programs. In addition, we also introduce a weakest liberal precondition over a graph program.

Definition 7.3 (Strongest liberal postconditions)**.** A condition d is a *liberal postcondition* w.r.t. a precondition c and a graph program P, if for all host graphs G and H,

$$G \models c \text{ and } H \in [\![P]\!]G \text{ implies } H \models d.$$

A *strongest liberal postcondition* w.r.t. c and P, denoted by $\mathrm{SLP}(c, P)$, is a liberal postcondition w.r.t. c and P that implies every liberal postcondition w.r.t. c and P. □

Definition 7.4 (Weakest liberal preconditions)**.** A condition c is a *liberal precondition* w.r.t. a postcondition d and a graph program P, if for all host graphs G and H,

$$G \models c \text{ and } H \in [\![P]\!]G \text{ implies } H \models d.$$

A *weakest liberal precondition* w.r.t. d and P, denoted by $\mathrm{WLP}(P, d)$, is a liberal precondition w.r.t. d and P that is implied by every liberal precondition w.r.t. d and P. □

Lemma 7.5. Let us consider a graph program P and a precondition c. Let d be a liberal postcondition w.r.t. c and P. Then d is a strongest liberal postcondition w.r.t. c and P if and only if for every graph H satisfying d, there exists a host graph G satisfying c such that $H \in [\![P]\!]G$.

Proof.
(If).
Assuming it is true that for every graph H satisfying d, there exists a host graph G satisfying c such that $H \in [\![P]\!]G$. Let H be a host graph satisfying d. From the assumption, there exists a graph G such that $G \models c$ and $H \in [\![P]\!]G$. Since $H \in [\![P]\!]G$, $H \models a$ for all liberal postcondition a over c and P. Hence, $H \models d$ implies $H \models a$ for all liberal postcondition a over c, P such that d is a strongest postcondition w.r.t. c and P
(Only if).
Assume that it is not true that for every host graph H, $H \models d$ implies there exists a host graph G satisfying c such that $H \in [\![P]\!]G$. We show that a graph satisfying d can not imply the graph satisfying all liberal postcondition w.r.t r and P. From the assumption, there exists a host graph H such that every host graph G does not satisfy c or $H \notin [\![P]\!]G$. In the case of $H \notin [\![P]\!]G$, we clearly can not guarantee characteristic of H w.r.t. P. Then for the case where G does not satisfy c, we also can not guarantee the satisfaction of any liberal postcondition a over c in H because a is dependent of c. Hence, we can not guarantee that H satisfying all liberal postcondition w.r.t. r and c. □

Lemma 7.6. Let us consider a graph program P and a postcondition d. Let c be a liberal precondition w.r.t. P and d. Then c is a weakest liberal precondition w.r.t. P and d if and only if for every graph G $G \vDash c$ if and only if for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$.

Proof. (If).
Suppose that $G \vDash c$ iff for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$. It implies for all host graphs H, $G \vDash c$ and $H \in [\![P]\!]G$ implies $H \vDash d$. From Definition 7.4, c is a liberal precondition. Let a be a liberal precondition w.r.t. P and d as well. From Definition 7.4, for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$, and from the premise, $G \vDash c$. Hence, c is a weakest liberal precondition.
(Only if).
Suppose that c is a weakest liberal precondition. From Definition 7.4, if $G \vDash c$ then $H \in [\![P]\!]G$ implies $H \vDash d$. Let a be a liberal precondition w.r.t P and d. From Definition 7.4, $G \vDash a$ implies for all H, $H \in [\![P]\!]G$ implies $H \vDash d$. Since for all a, $G \vDash a$ must imply $G \vDash c$, then $H \in [\![P]\!]G$ implies $H \vDash d$ must imply $G \vDash c$ as well. □

SLP and WLP for a loop $P!$ is not easy to construct because $P!$ may get stuck or diverge. In [17], the divergence is represented by infinite formulas while in [55], it is represented by a recursive equation that is not well-defined. In this thesis, for simplicity we only consider strongest liberal postconditions over loop-free graph programs.

For the conditional commands `if/try – then – else`, the execution of the command depends on the existence of a proper host graph as a result of executing a graph program. In [1], there is an assertion representing a condition that must be satisfied by a graph such that there exists a path to successful execution, and there is also an assertion representing a condition that must be satisfied by a host graph such that there exist a path to a failure. Here, we define assertion SUCCESS for the former and FAIL for the latter.

Definition 7.7 (Assertion SUCCESS)**.** For a graph program P, SUCCESS(P) is the predicate on host graphs where for all host graph G,

$$G \vDash \text{SUCCESS}(P) \text{ if and only if there exists a host graph } H \text{ with } H \in [\![P]\!]G.$$

□

Definition 7.8 (Assertion FAIL)**.** Let us consider a graph program P. FAIL(P) is the predicate on host graphs where for all host graph G,

$$G \vDash \text{FAIL}(P) \text{ if and only if fail} \in [\![P]\!]G. \quad \square$$

Note that for a graph program C, FAIL(C) does not necessarily equivalent to ¬SUCCESS(C), e.g. if $C = \{\texttt{nothing}, \texttt{add}\}; \texttt{zero}$ where `nothing` is the rule schema where the left and right-hand graphs are the empty graph, `add` is the rule schema where the left-hand graph is the empty graph and the right-hand graph is a single 0-labelled unmarked and unrooted node, and `zero` is a rule schema that match with the a 0-labelled unmarked and unrooted node. For a host graph G where there is no 0-labelled unmarked unrooted node, there is a derivation $\langle C, G\rangle \to^* H$ for some host graph H but also a derivation $\langle C, G\rangle \to^* \texttt{fail}$ such that $G \vDash$ SUCCESS(C) and $G \vDash$ FAIL(C).

Having a strongest liberal postcondition over a loop-free program P w.r.t a precondition c allows us to prove that the triple $\{c\}\, P\, \{d\}$ for an assertion d is partially correct. That is, by showing that d is implied by the strongest liberal postcondition.

Computing SLP($c, \mathcal{R}$) for a set of rule schemata $\mathcal{R}$ basically should be formef from the disjunction of all strongest liberal postcondition w.r.t c and each rule schema in $\mathcal{R}$. If the rule set is empty, then SLP($c, \mathcal{R}$) should be false since there is nothing to disjunct. Computing SLP($c, P; Q$) should be constructed by having SLP(c, P) and then find strongest liberal postcondition w.r.t. Q and the resulting formula.

The equation for program composition can be defined the same with the equation for program composition in [53, 55]. However, for `if` – `then` – `else` command, the command `if` in classical programming is followed by an assertion while in graph programs it is followed by a graph program. Hence, instead of checking the truth value of the assertion on the input graph, we should check if the satisfaction of SUCCESS and FAIL of the associated program on the input graph. Then for `try` – `then` – `else` command, which does not exist in classical programming, we need a equation for the command based on its similarity with `if` – `then` – `else`.

The execution of if/try commands yields two possibilities of results, so we need to check the strongest liberal postcondition for both possibilities and form the disjunction from them. For the graph program `if` C `then` P `else` Q, P can be executed if SUCCESS(C) holds and Q can be executed if FAIL(C) holds. Similarly for `try` C `then` P `else` Q, $C; P$ can be executed if SUCCESS(C) holds and Q can be executed if FAIL(C) holds.

Proposition 7.9 (Strongest liberal postcondition for loop-free programs)**.** Let us consider a precondition c and a loop-free program S. Then, the following holds:

1. If S is a set of rule schemata $\mathcal{R} = \{r_1, \ldots, r_n\}$,
$$\text{SLP}(c, \mathcal{R}) = \begin{cases} \text{SLP}(c, r_1) \vee \ldots \vee \text{SLP}(c, r_n) & ,\text{, if } n > 0, \\ \mathsf{false} & \text{, otherwise} \end{cases}$$
2. For loop-free programs C, P, and Q,

(i) If $S = P$ `or` Q,
$\mathrm{SLP}(c, S) = \mathrm{SLP}(c, P) \vee \mathrm{SLP}(c, Q)$

(ii) If $S = P; Q$,
$\mathrm{SLP}(c, S) = \mathrm{SLP}(\mathrm{SLP}(c, P), Q)$

(iii) If $S =$ `if` C `then` P `else` Q,
$\mathrm{SLP}(c, S) = \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$

(iv) If $S =$ `try` C `then` P `else` Q,
$\mathrm{SLP}(c, S) = \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), C; P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$

Proof. Here, we show that the proposition holds by induction on loop-free programs.
Base case.

1. If $S = \mathcal{R} = \{\}$,
   For all host graph G, $G \nRightarrow$ such that every condition is a liberal postcondition w.r.t. c and $\mathcal{R}$, false is the strongest among all because we know that there is no graph can satisfy `false`.
2. If $S = \mathcal{R} = \{r_1, \ldots, r_n\}$ where $n > 0$,
   (a)
$$\begin{aligned} H \vDash \mathrm{SLP}(c, \mathcal{R}) &\overset{\text{L7.5}}{\Leftrightarrow} \exists G. G \Rightarrow_{\mathcal{R}} H \wedge G \vDash c \\ &\Leftrightarrow \exists G. (G \Rightarrow_{r_1} H \vee \ldots \vee G \Rightarrow_{r_n} H) \wedge G \vDash c \\ &\Leftrightarrow (\exists G. G \Rightarrow_{r_1} H \wedge G \vDash c) \vee \ldots \vee (\exists G. G \Rightarrow_{r_n} H \wedge G \vDash c) \\ &\overset{\text{L7.5}}{\Leftrightarrow} H \vDash \mathrm{SLP}(c, r_1) \vee \ldots \vee \mathrm{SLP}(c, r_n) \end{aligned}$$

Inductive case. Assume the proposition holds for loop-free programs C, P, and Q.

1. If $S = P$ `or` Q,
$$\begin{aligned} H \vDash \mathrm{SLP}(c, S) &\overset{\text{L7.5}}{\Leftrightarrow} \exists G. H \in [\![P \texttt{ or } Q]\!] G \wedge G \vDash c \\ &\Leftrightarrow \exists G. (H \in [\![P]\!] G \vee H \in [\![Q]\!] G) \wedge G \vDash c \\ &\Leftrightarrow (\exists G. H \in [\![P]\!] G \wedge G \vDash c) \vee (\exists G. H \in [\![Q]\!] G \wedge G \vDash c) \\ &\overset{\text{L7.5}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c, P) \vee \mathrm{SLP}(c, Q) \end{aligned}$$
2. If $S = P; Q$,
$$\begin{aligned} H \vDash \mathrm{SLP}(c, S) &\overset{\text{L7.5}}{\Leftrightarrow} \exists G.\ H \in [\![P; Q]\!] G \wedge G \vDash c \\ &\Leftrightarrow \exists G, G'.\ G' \in [\![P]\!] G \wedge H \in [\![Q]\!] G' \wedge G \vDash c \\ &\overset{\text{L7.5}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{SLP}(c, P) \wedge H \in [\![Q]\!] G' \\ &\overset{\text{L7.5}}{\Leftrightarrow} H \vDash \mathrm{SLP}(\mathrm{SLP}(c, P), Q) \end{aligned}$$
3. If $S =$ `if` C `then` P `else` Q,
$$\begin{aligned} H \vDash \mathrm{SLP}(c, S) &\overset{\text{L7.5}}{\Leftrightarrow} \exists G.\ G \vDash c \wedge H \in [\![\texttt{if } C \texttt{ then } P \texttt{ else } Q]\!] G \\ &\Leftrightarrow \exists G.\ G \vDash c \wedge ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!] G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G)) \\ &\Leftrightarrow (\exists G.\ G \vDash c \wedge \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!] G) \vee (\exists G.\ G \vDash c \wedge \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G) \\ &\overset{\text{L7.5}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q) \end{aligned}$$

4. If $S = \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,

$$
\begin{aligned}
H \vDash \mathrm{SLP}(c, S) & \\
&\overset{\text{L7.5}}{\Leftrightarrow} \quad \exists G.\ G \vDash c \wedge H \in [\![\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!] G \\
&\Leftrightarrow \quad (\exists G, G'.\ G \vDash c \wedge G' \in [\![C]\!] G \wedge H \in [\![P]\!] G') \vee (\exists G.\ G \vDash c \wedge \mathrm{FAIL}(C) \wedge H \in \\
&\qquad [\![Q]\!] G) \\
&\overset{\text{L7.5}}{\Leftrightarrow} \quad (\exists G'.\ G' \vDash \mathrm{SLP}(c, C) \wedge H \in [\![P]\!] G') \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q) \\
&\overset{\text{L7.5}}{\Leftrightarrow} \quad G \vDash \mathrm{SLP}(\mathrm{SLP}(c, C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)
\end{aligned}
$$

□

To prove the triple $\{c\}\ P\ \{d\}$ is partially correct for a graph program P, we only need to show that $\mathrm{SLP}(c, P)$ implies d. However for graph programs P containing a loop, obtaining the assertion $\mathrm{SLP}(c, P)$ is not easy. Alternatively, we can create a proof tree (see Definition 2.26) with proof rules to show that $\{c\}\ P\ \{d\}$ is partially correct. Before we define the proof rules for partial correctness, we define predicate Break which shows relation between a graph program and assertions.

Definition 7.10 (Predicate Break)**.** Let us consider a graph program P and assertions c and d. $\mathrm{Break}(c, P, d)$ is the predicate defined by:

$\mathrm{Break}(c, P, d)$ holds iff for all derivations $\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle, G \vDash c$ implies $H \vDash d$.

□

Intuitively, when $\mathrm{Break}(c, P, d)$ holds, the execution of `break` that yields to termination of $P!$ will result in a graph satisfying d.

Lemma 7.11. Let us consider a graph program P with invariant c. If P does not contain the command `break`, then the following triple holds:

$$\{c\}\ P!\ \{c \wedge \mathrm{FAIL}(P)\}$$

Proof. If P does not contain the command `break`, then the derivation $\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle$ must not exist for any host graphs G and H. Hence, $\mathrm{Break}(c, P, d)$ is true for any c and d. Hence, $\mathrm{Break}(c, P, \mathrm{false})$ must be true. Since c in an invariant, $\{c\}\ P\ \{c\}$ is true. If $\langle P!, G\rangle \to^* H$ for some host graph H, from the semantics of graph programs, $\langle P!, G\rangle \to \langle P!, H\rangle \to^+ \texttt{fail}$. H must satisfy c because c is the invariant of P, and H must satisfy $\mathrm{FAIL}(P)$ because $fail \in [\![P]\!] H$. Hence, the triple holds. □

Definition 7.12 (Semantic partial correctness proof rules)**.** The semantic partial correctness proof rules for core commands, denoted by SEM, is defined in Figure 7.1, where c, d, and d' are any assertions, r is any conditional rule schema, $\mathcal{R}$ is any set of rule schemata, and C, P, and Q are any graph programs. □

$$[\text{ruleapp}]_{\text{slp}}\ \frac{}{\{c\}\ r\ \{\text{SLP}(c,r)\}}$$

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{WLP}(r,d)\}\ r\ \{d\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\}\ \text{for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P;Q\ \{d\}}$$

$$[\text{cons}]\ \frac{c \text{ implies } c' \quad \{c'\}\ P\ \{d'\} \quad d' \text{ implies } d}{\{c\}\ P\ \{d\}}$$

$$[\text{if}]\ \frac{\{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\} \quad \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge \text{SUCCESS}(C)\}\ C;P\ \{d\} \quad \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{alap}]\ \frac{\{c\}\ P\ \{c\} \quad \text{Break}(c,P,d)}{\{c\}\ P!\ \{(c \wedge \text{FAIL}(P)) \vee d\}}$$

FIGURE 7.1: Calculus SEM of semantic partial correctness proof rules

The inference rule [ruleset] tells us about the application of a set of rule schemata $\mathcal{R}$. The rule set $\mathcal{R}$ is applied to a graph by nondeterministically choose an applicable rule schema from the set and apply it to the input graph. Hence, to derive a triple about $\mathcal{R}$, we need to prove the same triple for each rule schema inside $\mathcal{R}$.

The inference rule [comp] is similar to [comp] in traditional programming. In executing $P;Q$, the graph program Q is not executed until after the execution of P has terminated. So to show a triple about $P;Q$, we need to prove a triple about each P and Q and show that they are connected to some midpoint such that the midpoint is satisfied after the execution of P and before the execution of Q.

Like in conventional Hoare logic [56], the rule [cons] is aimed to strengthen the precondition and weaken the postcondition, or to replace the condition to another condition that semantically equivalent but syntactically different. To show that c' can be strengthened to c, we only need to show that c implies c', and to weaken d' to d, we need to show that d' implies d.

The assertions SUCCESS and FAIL are needed to prove a triple about if command. Recall that in the execution of `if` C `then` P `else` Q, the program C is first executed on a copy of G. If it terminates and yields a proper graph as a result, P is executed on G. If C terminates and results in a fail state, then Q is executed on G.

Similarly, for a triple about try command, we use the two assertions. But for `try` C `then` P `else` Q, C is not executed on a copy of G, but G itself. When the execution of C on G terminates and yields a proper graph, P is executed on the result graph. Hence, the difference with [if] is located in the first of the premises, where we use the sequential composition of C and P.

As in traditional programming, we need an invariant to show a triple about loop $P!$. When we have proven the existence of an invariant for P, the invariant will hold after any number of successful executions of P. If $P!$ terminates, from the semantics of "!" we know that the last execution of P either yields a fail state (see [Loop$_2$] of Figure 2.10), such that FAIL(P) must hold, or executing the command `break` (see [Loop$_3$] of Figure 2.10). In the former case, we know that the invariant and FAIL(P) must hold from the semantics of GP 2. Then in the latter case, we use Break(c, P, d) which is defined in Definition 7.10. The triple for loops is then captured by the rule [alap], so this rule set covers nested loops and breaks as well now.

7.2 Syntactical Proof Calculus

Section 7.1 introduces us to the semantic partial correctness calculus, where assertions are functions that map graphs to Boolean value. Now, we consider monadic second-order formulas as the functions. In this section we define the construction of SLP, SUCCESS, and FAIL in monadic second-order formulas. In addition, we also define the syntactic version of partial correctness proof rules where possible (it will turn out that this is not always can be done). First, we define the monadic second-order formula App(r) which should represent the monadic second-order formula of SUCCESS(r).

Definition 7.13 (App(r))**.** Let us consider a conditional rule schema $r : \langle L \leftarrow K \rightarrow R,\ \Gamma \rangle$. The formula App($r$) is defined as

$$\mathrm{App}(r) = \mathrm{Post}(\mathrm{Spec}(L) \wedge \mathrm{Dang}(r) \wedge \Gamma).$$

□

The definition of Post(c), Spec(L), and Dang(r) for a condition c, rule graph L, and rule schema r, can be found in Definition 5.14, 3.23, and 2.8 respectively.

Lemma 7.14. Let us consider a conditional rule schema $r : \langle L \leftarrow K \rightarrow R,\ \Gamma \rangle$, and a host graph G,

$$G \vDash \mathrm{SUCCESS}(r) \text{ if and only if } G \vDash \mathrm{App}(r).$$

Proof. (If).
From the definition of Post (see Definition 5.14) and Fact 3.1, we know that $G \vDash \mathrm{App}(r)$ implies $G \vDash \mathrm{Var}(\mathrm{Spec}(L))$, such that from Lemma 3.27, we know that there exists injective morphism $g : L^\alpha \rightarrow G$ for some label assignment α_L. Then from Lemma 3.26, $G \vDash \mathrm{App}(r)$ implies $\rho_g(G) \vDash \mathrm{Dang}(r)$ and $\rho_g(G) \vDash \Gamma^\alpha$. From Observation 5.1, $\rho_g(G) \vDash \mathrm{Dang}(r)$ implies g satisfies the dangling condition, and $\rho_g(G) \vDash \Gamma^\alpha$ clearly implies $\Gamma^{\alpha,g}$ is satisfied by G. Hence,

from the definition of conditional rule schema application, we know that $G \Rightarrow_{r,g} H$ for some host graph H such that $G \vDash \mathrm{SUCCESS}(r)$.
(Only if).
$G \vDash \mathrm{SUCCESS}(r)$ implies $G \Rightarrow H$ for some host graph H, which implies the existence of injective morphism $g : L^{\alpha} \to G$ for some label assignment α_L such that g satisfies the dangling condition and $G \vDash \Gamma^{\alpha,g}$. The existence of the injective morphism implies $G \vDash \mathrm{Var}(\mathrm{Spec}(L))$ from Lemma 3.27, the satisfaction of the dangling condition implies $\rho_g(G) \vDash \mathrm{Dang}(r)$, and the $G \vDash \Gamma^{\alpha,g}$ implies $\rho_g(G) \vDash \Gamma$. Hence, $\rho_g(G) \vDash \mathrm{Spec}(L)$ since $L^{\alpha} \to \rho_g(G)$ is inclusion (see Proposition 3.24). Hence, $\rho_g(G) \vDash \mathrm{Spec}(L) \wedge \mathrm{Dang}(r) \wedge \Gamma$ so that from Lemma 3.26, $G \vDash \mathrm{App}(r)$. □

Defining a monadic second-order formula for $\mathrm{SUCCESS}(r)$ with a rule schema r is easier than defining MSO formula for $\mathrm{SUCCESS}(P)$ with an arbitrary loop-free program P. This is because we need to express properties of the initial graph after checking the existence of derivations. To determine the properties of the initial graph, we introduce the condition $\mathrm{Pre}(P, c)$ for a postcondition c and a loop-free program P. Intuitively, $\mathrm{Pre}(P, c)$ expresses the properties of the initial graph such that we can assert the existence of a host graph H such that $H \vDash c$ and $H \in [\![P]\!]G$. For an example, if there exists host graphs G' and H for a given host graph G and rule schemata r_1 and r_2 such that $G \Rightarrow_{r_1} G' \Rightarrow_{r_2} H$ and $H \vDash \mathsf{true}$ (which also means that $G \vDash \mathrm{SUCCESS}(P)$), then G' should satisfy $\mathrm{Pre}(r_2, \mathsf{true})$ and G should satisfy $\mathrm{Pre}(r_1, \mathrm{Pre}(r_2, \mathsf{true}))$ such that $\mathrm{Pre}(r_1, \mathrm{Pre}(r_2, \mathsf{true}))$ can be considered as $\mathrm{SUCCESS}(r_1; r_2)$ in monadic second-order formula. For more general cases, see Definition 7.15. In the definition, $(r^{\vee})^{-1}$ refers to the inverse of the generalised r (see Definition 3.19).

Definition 7.15 (Slp, Success, Fail, Pre of a loop-free program)**.** Let us consider a condition c and a loop-free program S. The monadic second-order formulas $\mathrm{Slp}(c, S)$, $\mathrm{Pre}(c, S)$, $\mathrm{Success}(S)$, and $\mathrm{Fail}(S)$ are defined inductively:

1. If S is a set of rule schemata $\mathcal{R} = \{r_1, \ldots, r_n\}$,
   (a) $\mathrm{Slp}(c, S) = \begin{cases} \mathrm{Post}(c, r_1^{\vee}) \vee \ldots \vee \mathrm{Post}(c, r_n^{\vee}) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$
   (b) $\mathrm{Pre}(S, c) = \begin{cases} \mathrm{Post}(c, (r_1^{\vee})^{-1}) \vee \ldots \vee \mathrm{Post}(c, (r_n^{\vee})^{-1}) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$
   (c) $\mathrm{Success}(S) = \begin{cases} \mathrm{App}(r_1) \vee \ldots \vee \mathrm{App}(r_n) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$
   (d) $\mathrm{Fail}(S) = \begin{cases} \neg(\mathrm{App}(r_1) \vee \ldots \vee \mathrm{App}(r_n)) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$
2. For loop-free programs C, P, and Q,

(i) If $S = P$ `or` Q,
  (a) $\text{Slp}(c, S)$=$\text{Slp}(c, P) \vee \text{Slp}(c, Q)$
  (b) $\text{Pre}(S, c)$=$\text{Pre}(P, c) \vee \text{Pre}(Q, c)$
  (c) $\text{Success}(S)$ =$\text{Success}(P) \vee \text{Success}(Q)$
  (d) $\text{Fail}(S)$ =$\text{Fail}(P) \vee \text{Success}(Q)$

(ii) If $S = P; Q$,
  (a) $\text{Slp}(c, S)$=$\text{Slp}(\text{Slp}(c, P), Q)$
  (b) $\text{Pre}(S, c)$=$\text{Pre}(P$,$\text{Pre}(Q, c))$
  (c) $\text{Success}(S)$ =$\text{Pre}(P, \text{Success}(Q))$
  (d) $\text{Fail}(S)$ =$\text{Fail}(P) \vee \text{Pre}(P$,$\text{Fail}(Q))$

(iii) If $S =$ `if` C `then` P `else` Q,
  (a) $\text{Slp}(c, S)$=$\text{Slp}(c \wedge \text{Success}(C), P) \vee \text{Slp}(c \wedge \text{Fail}(C), Q)$
  (b) $\text{Pre}(S, c)$=$(\text{Success}(C) \wedge \text{Pre}(P, c)) \vee (\text{Fail}(C) \wedge \text{Pre}(Q, c))$
  (c) $\text{Success}(S)$ = $(\text{Success}(C) \wedge \text{Success}(P) \vee (\text{Fail}(C) \wedge \text{Success}(Q))$
  (d) $\text{Fail}(S)$=$(\text{Success}(C) \wedge \text{Fail}(P)) \vee (\text{Fail}(C) \wedge \text{Fail}(Q))$

(iv) If $S =$ `try` C `then` P `else` Q,
  (a) $\text{Slp}(c, S)$=$\text{Slp}(c \wedge \text{Success}(C), C; P) \vee \text{Slp}(c \wedge \text{Fail}(C), Q)$
  (b) $\text{Pre}(S, c)$ =$\text{Pre}(C, \text{Pre}(P, c)) \vee (\text{Fail}(C) \wedge \text{Pre}(Q, c))$
  (c) $\text{Success}(S)$ =$\text{Pre}(\text{C}$,$\text{Success}(P)) \vee (\text{Fail}(C) \wedge \text{Success}(Q))$
  (d) $\text{Fail}(S)$ =$\text{Pre}(\text{Fail}(P), C)) \vee (\text{Fail}(C) \wedge \text{Fail}(Q))$

□

For a precondition c and a loop-free program S, $\text{Slp}(c, S)$ is basically constructed based on Proposition 7.9. For $\text{Pre}(S, c)$, since we want to know the property of the initial graph based on c that is satisfied by the final graph and S, it works similar with constructing a weakest liberal precondition from a given postcondition and a program. Here we use [17] as a reference. However, in the reference the conditional part of `if – then – else` command contains an assertion instead of a graph program such that if C is an assertion, following the setting in [17] we will have $\text{Pre}(C, c) = C \implies \text{Pre}(P, c) \wedge \neg C \implies \text{Pre}(Q, c)$. The difference between assertions and graph programs as condition of a conditional program is, the satisfaction of the assertion on the initial graph implies that Q can not be executed, while in our case, $G \vDash \text{Success}(C)$ does not always imply that Q can not be executed. Hence, we change the equation to what we have in the definition above.

$\text{Success}(S)$ should express the existence of a proper graph as a final result, which means it should express the property of the initial graph based on S and the final graph satisfying `true`. This is exactly what $\text{Pre}(\text{true}, S)$ should express. Finally, $\text{Fail}(S)$ should express the property of the initial graph where failure is a result of the execution of S. Since we can yield failure anywhere is the subprogram of S, we need to disjunct all possibilities.

Theorem 7.16 (Slp, Pre, Success, and Fail)**.** For all condition c and loop-free program S, the following holds:

(a) $\mathrm{Slp}(c, S)$ is a strongest liberal postcondition w.r.t. c and S
(b) For all host graph G, $G \vDash \mathrm{Pre}(S, c)$ if and only if there exists host graph H such that $H \in [\![S]\!]G$ and $H \vDash c$
(c) $G \vDash \mathrm{Success}(S)$ if and only if $G \vDash \mathrm{SUCCESS}(S)$
(d) $G \vDash \mathrm{Fail}(S)$ if and only if $G \vDash \mathrm{FAIL}(S)$

Proof. Here, we prove the theorem by induction on loop-free graph programs.
Base case.

1. For $\mathcal{R} = \{\}$,
   (a) for all host graph G, $G \not\Rightarrow$ such that every condition is a liberal postcondition w.r.t. c and $\mathcal{R}$, false is the strongest among all because we know that there is no graph can satisfy `false`
   (b) Statement (b) is valid because nothing satisfies false.
   (c) Both $G \vDash \mathrm{Success}(\mathcal{R})$ and $G \vDash \mathrm{SUCCESS}(\mathcal{R})$ always false such that $G \vDash \mathrm{Success}(\mathcal{R})$ iff $G \vDash \mathrm{SUCCESS}(\mathcal{R})$ holds.
   (d) Similarly, this point holds because both $G \vDash \mathrm{Fail}(\mathcal{R})$ and $G \vDash \mathrm{FAIL}(\mathcal{R})$ always true.

2. If $S = \mathcal{R} = \{r_1, \ldots, r_n\}$ where $n > 0$,
   (a) $$\begin{aligned} H \vDash \mathrm{SLP}(c, \mathcal{R}) &\overset{\text{P7.9}}{\Leftrightarrow} H \vDash \mathrm{SLP}(c, r_1) \vee \ldots \vee \mathrm{SLP}(c, r_n) \\ &\overset{\text{T5.16}}{\Leftrightarrow} H \vDash \mathrm{Post}(c, r_1^\vee) \vee \ldots \vee \mathrm{Post}(c, r_n^\vee) \end{aligned}$$
   (b) $$\begin{aligned} &\exists H.H \in [\![\mathcal{R}]\!]G \wedge H \vDash c \\ &\Leftrightarrow \exists H.(G \Rightarrow_{r_1} H \vee \ldots \vee G \Rightarrow_{r_n} H) \wedge H \vDash c \\ &\Leftrightarrow (\exists H.G \Rightarrow_{r_1} H \wedge H \vDash c) \vee \ldots \vee (\exists H.G \Rightarrow_{r_n} H \wedge H \vDash c) \\ &\overset{\text{D3.18}}{\Leftrightarrow} (\exists H.G \Rightarrow_{r_1^\vee} H \wedge H \vDash c) \vee \ldots \vee (\exists H.G \Rightarrow_{r_n^\vee} H \wedge H \vDash c) \\ &\overset{\text{L3.22}}{\Leftrightarrow} (\exists H.H \Rightarrow_{(r_1^\vee)^{-1}} H \wedge H \vDash c) \vee \ldots \vee (\exists H.H \Rightarrow_{(r_n^\vee)^{-1}} G \wedge H \vDash c) \\ &\overset{\text{T5.16}}{\Leftrightarrow} G \vDash \mathrm{Post}(c, (r_1^\vee)^{-1}) \vee \ldots \vee \mathrm{Post}(c, (r_n^\vee)^{-1}) \end{aligned}$$
   (c) $$\begin{aligned} H \vDash \mathrm{SUCCESS}(\mathcal{R}) &\Leftrightarrow \exists H.H \in [\![\mathcal{R}]\!]G \\ &\Leftrightarrow \exists H.G \Rightarrow_{r_1} H \vee \ldots \vee G \Rightarrow_{r_n} H \\ &\Leftrightarrow (\exists H.G \Rightarrow_{r_1} H) \vee \ldots \vee (\exists H.G \Rightarrow_{r_n} H) \\ &\overset{\text{D7.7}}{\Leftrightarrow} G \vDash \mathrm{SUCCESS}(r_1) \vee \ldots \vee \mathrm{SUCCESS}(r_n) \\ &\overset{\text{L7.14}}{\Leftrightarrow} G \vDash \mathrm{App}(r_1) \vee \ldots \vee \mathrm{App}(r_n) \end{aligned}$$
   (d) $$\begin{aligned} G \vDash \mathrm{FAIL}(\mathcal{R}) &\Leftrightarrow \mathrm{fail} \in [\![\mathcal{R}]\!]G \\ &\Leftrightarrow (\neg\exists H.G \Rightarrow_{r_1} H) \wedge \ldots \wedge (\neg\exists H.G \Rightarrow_{r_n} H) \\ &\overset{\text{D7.7}}{\Leftrightarrow} G \vDash \neg(\mathrm{SUCCESS}(r_1) \vee \ldots \wedge \mathrm{SUCCESS}(r_n)) \\ &\overset{\text{L7.14}}{\Leftrightarrow} G \vDash \neg(\mathrm{App}(r_1) \vee \ldots \vee \mathrm{App}(r_n)) \end{aligned}$$

Inductive case. Assume (a), (b), (c), and (d) hold for loop-free programs C, P, and Q.

1. If $S = P \texttt{ or } Q$,
   (a) $H \vDash \mathrm{SLP}(c, S) \overset{\text{P7.9}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c, P) \vee \mathrm{SLP}(c, Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Slp}(c, P) \vee \mathrm{Slp}(c, Q)$

   (b) $\exists H.H \in [\![S]\!]G \wedge H \vDash c \Leftrightarrow \exists H.(H \in [\![P]\!]G \vee H \in [\![Q]\!]G) \wedge H \vDash c$
   $\Leftrightarrow (\exists H.H \in [\![P]\!]G \wedge H \vDash c) \vee (\exists H.H \in [\![Q]\!]G \wedge H \vDash c)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, c) \vee \mathrm{Pre}(Q, c)$

   (c) $G \vDash \mathrm{SUCCESS}(S) \Leftrightarrow \exists H.H \in [\![P \texttt{ or } Q]\!]G$
   $\Leftrightarrow \exists H.H \in [\![P]\!]G \vee H \in [\![Q]\!]G$
   $\overset{\text{D7.7}}{\Leftrightarrow} G \vDash \mathrm{SUCCESS}(P) \vee \mathrm{SUCCESS}(Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Success}(P) \vee \mathrm{Success}(Q)$

   (d) $G \vDash \mathrm{FAIL}(S) \Leftrightarrow \mathrm{fail} \in [\![P \texttt{ or } Q]\!]G$
   $\Leftrightarrow \mathrm{fail} \in [\![P]\!]G \vee \mathrm{fail} \in [\![Q]\!]G$
   $\overset{\text{D7.8}}{\Leftrightarrow} G \vDash \mathrm{FAIL}(P) \vee \mathrm{FAIL}(Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Fail}(P) \vee \mathrm{Fail}(Q)$

2. If $S = P; Q$,
   (a) $H \vDash \mathrm{SLP}(c, S) \overset{\text{P7.9}}{\Leftrightarrow} H \vDash \mathrm{SLP}(\mathrm{SLP}(c, P), Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} H \vDash \mathrm{Slp}(\mathrm{Slp}(c, P), Q)$

   (b) $\exists H.H \in [\![S]\!]G \wedge H \vDash c \Leftrightarrow \exists H, G'.\ G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G' \wedge H \vDash c$
   $\overset{\text{Ind.}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{Pre}(Q, c) \wedge G' \in [\![P]\!]G$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, \mathrm{Pre}(Q, c))$

   (c) $G \vDash \mathrm{SUCCESS}(S) \Leftrightarrow \exists H.H \in [\![P; Q]\!]G$
   $\Leftrightarrow \exists H, G'.\ G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G'$
   $\overset{\text{D7.7}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{SUCCESS}(Q) \wedge G' \in [\![P]\!]G$
   $\overset{\text{Ind.}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{Success}(Q) \wedge G' \in [\![P]\!]G$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, \mathrm{Success}(Q))$

   (d) $G \vDash \mathrm{FAIL}(S) \Leftrightarrow \mathrm{fail} \in [\![P; Q]\!]G$
   $\Leftrightarrow \mathrm{fail} \in [\![P]\!]G \vee \exists H.\ H \in [\![P]\!]G \wedge \mathrm{fail} \in [\![Q]\!]H$
   $\overset{\text{D7.8}}{\Leftrightarrow} G \vDash \mathrm{FAIL}(P) \vee \exists H.\ H \in [\![P]\!]G \wedge H \vDash \mathrm{FAIL}(Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Fail}(P) \vee \mathrm{Pre}(P, \mathrm{Fail}(Q))$

3. If $S = \texttt{if } C \texttt{ then } P \texttt{ else } Q$,
   (a) $H \vDash \mathrm{SLP}(c, S)$
   $\overset{\text{P7.9}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Slp}(c \wedge \mathrm{Success}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{Fail}(C), Q)$

   (b) $\exists H.H \in [\![S]\!]G \wedge H \vDash c$
   $\Leftrightarrow \exists H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \wedge$
   $H \vDash c$
   $\Leftrightarrow (\exists H.\ G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G \wedge H \vDash c)$
   $\vee (\exists H.\ G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \wedge H \vDash c)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\mathrm{Success}(C) \wedge \mathrm{Pre}(P, c)) \vee (\mathrm{Fail}(C) \wedge \mathrm{Pre}(Q, c))$

(c) $G \vDash \text{SUCCESS}(S)$
$\Leftrightarrow \exists H.H \in [\![S]\!]G$
$\Leftrightarrow \exists H.\ (G \vDash \text{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
$\Leftrightarrow (\exists H.\ G \vDash \text{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (\exists H.\ G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G)))$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\text{Success}(C) \wedge \text{Success}(P)) \vee (\text{Fail}(C) \wedge \text{Success}(Q))$

(d) $G \vDash \text{FAIL}(S)$
$\Leftrightarrow \text{fail} \in [\![S]\!]G$
$\Leftrightarrow (G \vDash \text{SUCCESS}(C) \wedge \text{fail} \in [\![P]\!]G) \vee (G \vDash \text{FAIL}(C) \wedge \text{fail} \in [\![Q]\!]G)$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\text{Success}(C) \wedge \text{Fail}(P)) \vee (\text{Fail}(C) \wedge \text{Fail}(Q))$

4. If $S = \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,

(a) $H \vDash \text{SLP}(c, S)$
$\overset{\text{P7.9}}{\Leftrightarrow} G \vDash \text{SLP}(\text{SLP}(c, C), P) \vee \text{SLP}(c \wedge \text{FAIL}(C), Q)$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Slp}(\text{Slp}(c, C), P) \vee \text{Slp}(c \wedge \text{Fail}(C), Q)$

(b) $\exists H.H \in [\![S]\!]G \wedge H \vDash c$
$\Leftrightarrow (\exists H, G'.\ H \vDash c \wedge G' \in [\![C]\!]G \wedge H \in [\![P]\!]G') \vee (\exists H.\ H \vDash c \wedge \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
$\overset{\text{Ind.}}{\Leftrightarrow} (\exists G'.\ G' \vDash \text{Pre}(P, c) \wedge G' \in [\![C]\!]G) \vee (\exists H.\ G \vDash \text{Fail}(C) \wedge H \in [\![Q]\!]G)) \wedge H \vDash c)$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Pre}(C, \text{Pre}(P, c)) \vee (\text{Fail}(C) \wedge \text{Pre}(Q, c))$

(c) $G \vDash \text{SUCCESS}(S)$
$\Leftrightarrow \exists H.H \in [\![S]\!]G$
$\Leftrightarrow (\exists H, G'.\ G' \in [\![C]\!]G \wedge H \in [\![P]\!]G') \vee (\exists H.\ \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
$\overset{\text{D7.7}}{\Leftrightarrow} (\exists G'.\ G' \in [\![C]\!]G \wedge G' \vDash \text{SUCCESS}(P)) \vee (\exists H.\ \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Pre}(C, \text{Success}(P)) \vee (\text{Fail}(C) \wedge \text{Success}(Q))$

(d) $G \vDash \text{FAIL}(S)$
$\Leftrightarrow \text{fail} \in [\![S]\!]G$
$\Leftrightarrow (\exists G'.\ G' \in [\![C]\!]G \wedge \text{fail} \in [\![P]\!]G) \vee (G \vDash \text{FAIL}(C) \wedge \text{fail} \in [\![Q]\!]G)$
$\overset{\text{D7.7,7.8}}{\Leftrightarrow} G \vDash (\text{SUCCESS}(C) \wedge \text{FAIL}(P)) \vee (\text{FAIL}(C) \wedge \text{FAIL}(Q))$
$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\text{Success}(C) \wedge \text{Fail}(P)) \vee (\text{Fail}(C) \wedge \text{Fail}(Q))$

□

For any loop-free program P, we now can find the monadic second-order formula of SLP, SUCCESS, and FAIL. However, constructing SLP and SUCCESS of a loop is a challenging task because a loop may diverge. However, constructing a MSO formula for FAIL of a graph program with loops is not as challenging if we only consider some forms of graph programs. In [11], Bak introduced a class of commands that cannot fail. Hence, we can always conclude that $\text{Fail}(P) = \mathsf{false}$ if P is a command that cannot fail. Here, we introduce the class of non-failing commands.

Definition 7.17 (Non-failing commands)**.** The class of *non-failing commands* is inductively defined as follows:

Base case:

1. `break` and `skip` are non-failing commands
2. Every call of a rule schema with the empty graph as its left-hand graph is a non-failing command
3. Every rule set call $\{r_1, \ldots, r_n\}$ for $n \geq 1$ where each r_i has the empty graph as its left-hand graph, is a non-failing command
4. Every command P! is a non-failing command

Inductive case:

1. $P; Q$ is a non-failing command if P and Q are non-failing commands
2. `if` C `then` P `else` Q is a non-failing command if P and Q are non-failing commands
3. `try` C `then` P `else` Q is a non-failing command if P and Q are non-failing commands

□

Recall the inference rule [alap] of SEM (see Figure 7.1). To obtain a triple $\{c\}\ P!\ \{d\}$ for some precondition c, postcondition d, and a graph program $P!$, we need to find Fail(P). We now can construct Fail(P) if P is a loop-free program as in Definition 7.15, or if P is a non-failing command.

Now, let us consider P in the form $C; Q$. For any host graph G, fail $\in [\![C; Q]\!]G$ iff fail $\in [\![C]\!]G$ or $H \in [\![C]\!]G \wedge$ fail $\in [\![Q]\!]H$ for some host graph H, which means $G \vDash \text{FAIL}(C) \vee (\text{SUCCESS}(C) \wedge \text{FAIL}(Q))$. We can construct both Fail(C) and Success(C) if C is a loop-free program, and we can construct Fail(Q) if Q is a loop-free program or a non-failing command. Here, we introduce the class of *iteration commands* which is the class of commands where we can obtain Fail of the commands.

Definition 7.18 (Iteration commands)**.** The class of iteration commands is inductively defined as follows:

1. Every loop-free program is an iteration command
2. Every non-failing command is an iteration command
3. A command of the form $C; P$ is an iteration command if C is a loop-free program and P is an iteration command

□

If S is a loop-free program, we can construct Fail(S) as defined in Definition 7.15. Meanwhile, if S is a non-failing command, there is no graph G such that fail$\in [\![S]\!]G$ such that we can

conclude that $\text{Fail}(S) \equiv \mathsf{false}$. Finally, if S is in the form of $C;P$ for a loop-free program C and a non-failing program P, $\text{fail} \in [\![S]\!]G$ for a graph G only if $\text{fail} \in [\![C]\!]G$ (because P cannot fail), so that $\text{Fail}(S) \equiv \text{fail}(C)$.

Definition 7.19 (Fail of iteration commands)**.** Let $\text{Fail}_{\text{lf}}(C)$ denotes the formula $\text{Fail}(C)$ for a loop-free program C as defined in Definition 7.15. For any iteration command S,

$$\text{Fail}(S) = \begin{cases} \mathsf{false} & \text{if } S \text{ is a non-failing command} \\ \text{Fail}_{\text{lf}}(S) & \text{if } S \text{ is a loop-free program} \\ \text{Fail}(C) & \text{if } S = C;P \text{ for a loop-free program } C, \text{ a non-failing program } P \end{cases}$$

□

Theorem 7.20. Let us consider an iteration command S. Then,

$$G \vDash \text{Fail}(S) \text{ if and only if } G \vDash \text{FAIL}(S).$$

Proof. Here, we prove the theorem case by case.

1. If S is a non-failing command, then for any host graph G, $\text{fail} \notin [\![S]\!]G$. Hence, there is no graph satisfying $\text{FAIL}(S)$ such that $G \vDash \mathsf{false}$ iff $G \vDash \text{FAIL}(S)$ holds.
2. If S is a loop-free program, $G \vDash \text{Fail}(S)$ iff $G \vDash \text{FAIL}(S)$ holds based on Theorem 7.16.
3. If S is in the form $C;P$ for a loop-free program C and non-failing command P, then

$$\begin{aligned} G \vDash \text{FAIL}(C;P) \quad &\text{iff} \quad \text{fail} \in [\![C;P]\!]G \\ &\text{iff} \quad \text{fail} \in [\![C]\!]G \vee \exists G'.G' \in [\![C]\!]G \wedge \text{fail} \in [\![P]\!]G' \\ &\text{iff} \quad \text{fail} \in [\![C]\!]G \\ &\text{iff} \quad G \vDash \text{FAIL}(C) \\ &\text{iff} \quad G \vDash \text{Fail}(C) \end{aligned}$$

□

Now let us consider the proof calculus SEM. There is the assertion $\text{SUCCESS}(C)$ and $\text{FAIL}(C)$ where C is the condition of a branching statement, and $\text{FAIL}(S)$ for a loop body S. Since we are only able to construct $\text{Success}(C)$ for a loop-free program C and $\text{FAIL}(S)$ for an iteration command S, we do not define the syntactic version for arbitrary graph programs. Hence, we require a loop-free program as the condition of every branching statement and an iteration command as every loop body. For the axiom $[\text{ruleapp}]_{\text{wlp}}$, we follow the construction in [26] where a weakest liberal precondition can be constructed using the construction of Slp.

Definition 7.21 (Control programs)**.** A *control command* is a command where the condition of every branching command is loop-free and every loop body is an iteration command. Similarly, a graph program is a *control program* if all its command are control commands. □

$$[\text{ruleapp}]_{\text{slp}}\ \frac{}{\{c\}\ r\ \{\text{Slp}(c,r)\}}$$

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\neg\text{Slp}(\neg d, r^{-1})\}\ r\ \{d\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\}\ \text{for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P;Q\ \{d\}}$$

$$[\text{cons}]\ \frac{c \text{ implies } c' \quad \{c'\}\ P\ \{d'\} \quad d' \text{ implies } d}{\{c\}\ P\ \{d\}}$$

$$[\text{if}]\ \frac{\{c \wedge \text{Success}(C)\}\ P\ \{d\} \quad \{c \wedge \text{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge \text{Success}(C)\}\ C;P\ \{d\} \quad \{c \wedge \text{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{alap}]\ \frac{\{c\}\ S\ \{c\} \qquad \text{Break}(c,S,d)}{\{c\}\ S!\ \{(c \wedge \text{Fail}(S)) \vee d\}}$$

FIGURE 7.2: Calculus SYN of syntactic partial correctness proof rules

Lemma 7.22. Let us consider a conditional rule schema r and a closed monadic second-order formula d. Let $\text{Wlp}(r,d) = \neg\text{Slp}(\neg d, r^{-1})$. Then for all host graphs G,

$$G \vDash \text{Wlp}(r,d) \text{ if and only if } G \vDash \text{WLP}(r,d).$$

Proof.

$$\begin{array}{lll}
G \vDash \text{Wlp}(r,d) & \text{iff} & G \vDash \neg\text{Post}(\neg d, (r^{\vee})^{-1}) \\
& \text{iff} & \neg(\exists H, g, g^*.H \Rightarrow_{(r^{\vee})^{-1},g^*,g} G \wedge H \vDash \neg d) \qquad \text{(Lemma 4.2)} \\
& \text{iff} & \neg(\exists H, g, g^*.G \Rightarrow_{(r^{\vee}),g,g^*} H \wedge H \vDash \neg d) \qquad \text{(Lemma 3.22)} \\
& \text{iff} & \neg(\exists H.G \Rightarrow_r H \wedge H \vDash \neg d) \qquad \text{(Def. 3.18)} \\
& \text{iff} & \forall H.G \Rightarrow_r H \text{ implies } H \vDash d) \qquad \text{(Def. implication)} \\
& \text{iff} & G \vDash \text{WLP}(r,d) \qquad \text{(Lemma 7.5)}
\end{array}$$

□

Now we know the MSO formula for $\text{WLP}(r,c)$, $\text{SLP}(c,r)$, also $\text{SUCCESS}(P)$ and $\text{FAIL}(P)$ for some form of P. Finally, we define a syntactic partial correctness proof for control programs.

Definition 7.23 (Syntactic partial correctness proof rules)**.** The syntactic partial correctness proof rules, denoted by SYN, is defined in Figure 7.2, where c, d, and d' are any conditions, r is any conditional rule schema, $\mathcal{R}$ is any set of rule schemata, C is any loop-free program, P and Q are any control commands, and S is any iteration command. Outside a loop, we treat the command `break` as a `skip`. □

7.3 Summary

This chapter defines semantic and syntactic partial correctness proof calculi for graph program verification. Here, we define both semantic and syntactic calculus because semantic calculus is more powerful since it is independent on assertion language we use, while syntactic calculus is useful for presentation, as well as in practice. The semantic proof calculus can be used to verify arbitrary graph programs by Hoare-style verification by using semantic assertions. On the other hand, syntactic proof calculus can be used to verify a class of graph programs, called control programs, by using monadic second-order formulas in Hoare-style verification. However, we can not prove the implication between assertions or the so-called predicate Break inside the proof calculi.

The proof calculi we have in this chapter is an extension of the partial correctness proof calculus introduced in [1]. We extend the calculus so that now it can handle graph programs with command `break` and nested loop in certain forms. The extension is possible because we can show how to construct a monadic second-order formula as a precondition that asserts successful execution of a loop-free program or a failing execution of so-called iteration commands. In addition, we define the predicate Break to consider graph programs containing the command `break`. The extension give us an ability to prove larger subset of GP 2 graphs that are not able to be proven by the proof calculi in [1]. Now, we are able to prove graph programs that are instroduced in [11, 18, 19], which are proven to be efficient for some classes of graphs.

We are still not able to prove all GP 2 graph programs, because we still have a limitation on commands we can have inside a loop, also in branching commands. The limitation occur because we do not have a construction for SLP over a loop.

Chapter 8

Verification case studies

This chapter presents four graph programs: `vertex-colouring` [1], `transitive-closure` [34], `2-colouring` [11], and `is-connected` [57]. Here, we use graph programs existing in the literature as case studies. For each graph programs, pre- and postconditions are provided, and we verify the graph programs with respect to the given specifications.

8.1 Vertex colouring

8.1.1 Graph program `vertex-colouring`

The first example we consider in this chapter is vertex colouring program as seen in Figure 8.1. We expect the input of the program is a graph where all nodes are unmarked and unrooted, and all edges are unmarked. From the input graph, we mark all nodes with grey, then concatenate 1 to the label of all nodes while unmarking the nodes. This concatenation represents the "colour" in vertex colouring such that a node with label `a:1` represents a node with label `a` and colour 1. Finally, to make sure that all adjacent nodes have different colour, we use `change!` to change the colour of one from every two adjacent nodes if they have the same colour.

Here, we show that the graph program `vertex-colouring` is partially correct with respect to the following pre- and postcondition:

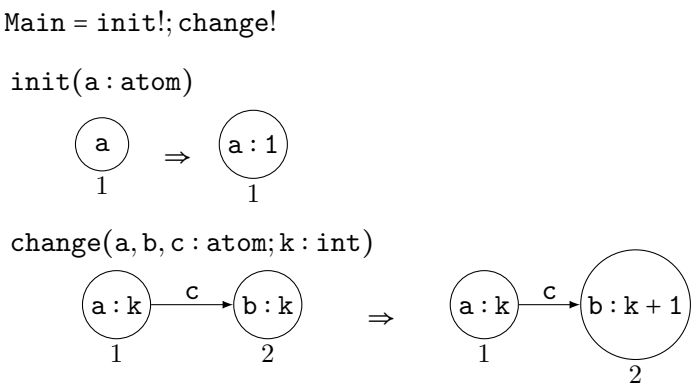


FIGURE 8.1: Graph program `vertex-colouring`

Precondition:
All nodes and edges are unmarked, all nodes are labelled with an atom, and all nodes are unrooted.
Postcondition:
All nodes are labelled with $a : i$ for some atom a and integer i; and for all distinct nodes x and y, if x and y are adjacent, x is labelled by $a : i$, and y is labelled by $b : j$ for some integers i and j, then $i \neq j$.

8.1.2 Proof tree of `vertex-colouring`

The proof is presented in the proof tree of Figure 8.2, where assertions used in the proof tree are defined in Table 8.1. Here, we use the transformation of conditions we defined in Chapter 6 to obtain strongest liberal postconditions Slp(e,`init`), and Slp(d,`change`). Let us show the process to obtain one of the strongest liberal postconditions as an example. Here, we choose Slp(d,`change`) because it is the most complicated one among the three.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\{e\}\ \texttt{init}\ \{\text{Slp(e,}\texttt{init}\text{)}\}}[\text{ruleapp}]_{\text{slp}}}{\{e\}\ \texttt{init}\ \{e\}}[\text{cons}]}{\{e\}\ \texttt{init!}\ \{e\wedge\text{Fail(}\texttt{init}\text{)}\}}[\text{alap}]}{\{\text{pre}\}\ \texttt{init!}\ \{d\}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{d\}\ \texttt{change}\ \{\text{Slp(d,}\texttt{change}\text{)}\}}[\text{ruleapp}]_{\text{slp}}}{\{d\}\ \texttt{change}\ \{d\}}[\text{cons}]}{\{d\}\ \texttt{change!}\ \{d\wedge\text{Fail(}\texttt{change}\text{)}\}}[\text{alap}]}{\{d\}\ \texttt{change!}\ \{\text{post}\}}[\text{cons}]}{\{\text{pre}\}\ \texttt{mark!}; \texttt{init!}; \texttt{change!}\ \{\text{post}\}}[\text{comp}]$$

FIGURE 8.2: Proof tree for partial correctness of `vertex_colouring`

First, we need to change the abbreviations we use in d, such that we obtain:
$\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{m_v}(\mathsf{x}) \neq \mathsf{none} \vee \mathsf{root}(\mathsf{x}) \vee \neg\exists_{\mathsf{l}}\mathsf{a}, \mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))) \wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{m_e}(\mathsf{x}) \neq \mathsf{none})$

To obtain $\text{Lift}_{\mathsf{MSO}}$(d,`change`), we first need to obtain $\text{Split}_{\mathsf{MSO}}$(d,`change`), that is:
$\neg(\mathsf{m_v}(1) \neq \mathsf{none} \vee \mathsf{root}(1) \vee \neg\exists_{\mathsf{l}}\mathsf{a}, \mathsf{i}(\mathsf{l_v}(1) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a})))$

TABLE 8.1: Conditions inside proof tree of `vertex_colouring`

MSO formulas
pre $\equiv \forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge \mathsf{atom}(\mathsf{l}_v(x))) \wedge \forall_E x(\mathsf{m}_e(x) = \mathsf{none})$
post $\equiv$ d$\wedge\forall_v x, y(x \neq y \wedge \mathsf{edge}(x,y) \Rightarrow \exists_l a, b, i, j(\mathsf{atom}(a) \wedge \mathsf{atom}(b) \wedge \mathsf{int}(i) \wedge \mathsf{int}(j)$ $\wedge\, \mathsf{l}_v(x) = a : i \wedge \mathsf{l}_v(y) = b : j \wedge i \neq j))$
d $\equiv \forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge \exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a))) \wedge \forall_e x(\mathsf{m}_e(x) = \mathsf{none})$
e $\equiv \forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge (\mathsf{atom}(\mathsf{l}_v(x) \vee \exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a))))$ $\wedge\, \forall_e x(\mathsf{m}_e(x) = \mathsf{none})$
Fail(`init`) $\equiv \neg\exists_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge \mathsf{atom}(\mathsf{l}_v(x)))$
Fail(`change`) $\equiv \neg\exists_v x, y(x \neq y \wedge \mathsf{edge}(x,y,\mathsf{none}) \wedge \mathsf{m}_v(x) = \mathsf{none} \wedge \mathsf{m}_v(y) = \mathsf{none}$ $\wedge\, \neg\mathsf{root}(x) \wedge \neg\mathsf{root}(y) \wedge \exists_l a, b, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{l}_v(y) = b : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a) \wedge \mathsf{atom}(b)))$
Slp(e,`init`)$\equiv$ $\exists_v y(\mathsf{m}_v(y) = \mathsf{none} \wedge \neg\mathsf{root}(y) \wedge \exists_l a(\mathsf{l}_v(y) = a : 1 \wedge \mathsf{atom}(a))$ $\wedge\forall_v x(x = y \vee ((\mathsf{m}_v(x) = \mathsf{none} \vee \mathsf{m}_v(x) = \mathsf{grey}) \wedge \neg\mathsf{root}(x)$ $\wedge\, (\mathsf{m}_v(x) = \mathsf{none} \Rightarrow \exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a)))))$ $\wedge\forall_e x(\mathsf{m}_e(x) = \mathsf{none})$
Slp(d,`change`)$\equiv$ $\exists_v y, z(y \neq z \wedge \mathsf{m}_v(y) = \mathsf{none} \wedge \mathsf{m}_v(z) = \mathsf{none} \wedge \neg\mathsf{root}(y) \wedge \neg\mathsf{root}(z)$ $\wedge\, \exists_l a, b, k(\mathsf{m}_v(y) = a : k \wedge \mathsf{m}_v(z) = b : k+1 \wedge \mathsf{int}(k) \wedge \mathsf{atom}(a) \wedge \mathsf{atom}(b)) \wedge \mathsf{edge}(y,z,\mathsf{none})$ $\wedge\, \forall_v x(x = y \vee x = z \vee (\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge \exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a)))))$ $\wedge\forall_e x(\mathsf{m}_e(x) = \mathsf{none})$

$\wedge\neg(\mathsf{m}_v(2) \neq \mathsf{none} \vee \mathsf{root}(2) \vee \neg\exists_l a, i(\mathsf{l}_v(2) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a)))$

$\wedge\neg\exists_v x(x \neq 1 \wedge x \neq 2 \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x) \vee \neg\exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a))))$

$\wedge(\neg(\mathsf{m}_e(e_1) \neq \mathsf{none}) \wedge \neg\exists_e x(x \neq e_1 \wedge \mathsf{m}_e(x) \neq \mathsf{none}))$

such that $\mathrm{Val}(\mathrm{Split}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}),\texttt{change})$ is:

$\neg\exists_v x(x \neq 1 \wedge x \neq 2 \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x) \vee \neg\exists_l a, i(\mathsf{l}_v(x) = a : i \wedge \mathsf{int}(i) \wedge \mathsf{atom}(a))))$

$\wedge\neg\exists_e x(x \neq e_1 \wedge \mathsf{m}_e(x) \neq \mathsf{none})$

because from the rule `change`, both node 1 and 2 are unmarked, unrooted, and labelled with a concanation of an atom and integer so that the first two lines can be simplified to `true`.

Since there is no rule application condition for `change`, then $\Gamma \equiv \mathsf{true}$. Also, there is no node that gets deleted by `change` such that Dang(`change`)$\equiv \mathsf{true}$. Hence,
$\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}) = \mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}),\texttt{change})$.

Next, since the rule `change` does not delete any node and not adding any new node, then $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}),\texttt{change}) = \mathrm{Split}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}),\texttt{change})$, such that
$\mathrm{Shift}_{\mathsf{MSO}}(\langle \texttt{change}, \mathrm{Lift}_{\mathsf{MSO}}(d, \texttt{change}), \mathsf{true}\rangle)$ is:
$\mathrm{Val}_{\mathsf{MSO}}(\mathrm{Split}_{\mathsf{MSO}}(\mathrm{d},\texttt{change}),\texttt{change}) \wedge \mathsf{s}(e_1) = 1 \wedge \mathsf{t}(e_1) = 2 \wedge \mathsf{m}_e(e_1) = \mathsf{none}$
$\wedge\, \mathsf{m}_v(1) = \mathsf{none} \wedge \mathsf{m}_v(2) = \mathsf{none} \wedge \neg\mathsf{root}(1) \wedge \neg\mathsf{root}(2)$
$\wedge \mathsf{m}_v(1) = a : k \wedge \mathsf{m}_v(2) = a : k+1 \wedge \mathsf{int}(k) \wedge \mathsf{atom}(a)$.

Finally, we can obtain Slp(d,`change`) as in Table 8.1 by transformation Post; that is by changing all 1s to y, 2s to z, and bound y, z, a, b, and k with existential quantifiers.

8.1.3 Proof of implications

From the proof tree, we can see that the proof rule [cons] is used several times. As mentioned before, the implication used in the proof rule need to be proven outside the calculus. Hence, we also provide the proof of implications to support the partial correctness proof.

1. *Proof of* pre *implies* e
   From the meaning of conjunction, $\mathsf{atom}(\mathsf{l_v}(\mathsf{x}))$ implies
   $\mathsf{atom}(\mathsf{l_v}(\mathsf{x})) \vee \exists_{\mathsf{l}}\mathsf{a}(\mathsf{l_v}(\mathsf{y}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))$. Hence, pre *implies* e.

2. *Proof of* Slp(e,`init`) *implies* e
   From the semantic of conjunction, Slp(e, `init`) implies
   $\forall_{\mathsf{v}}\mathsf{x}((\mathsf{m_v}(\mathsf{y}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{y}) \wedge \exists_{\mathsf{l}}\mathsf{a}(\mathsf{l_v}(\mathsf{y}) = \mathsf{a} : 1 \wedge \mathsf{atom}(\mathsf{a}))) \wedge \forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_e}(\mathsf{x}) = \mathsf{none})$.
   Since $\exists_{\mathsf{l}}\mathsf{a}(\mathsf{l_v}(\mathsf{y}) = \mathsf{a} : 1 \wedge \mathsf{atom}(\mathsf{a}))$ implies $\exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))$, also from the semantic of conjunction we know that $\exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))$ implies
   $\mathsf{atom}(\mathsf{l_v}) \vee \exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))$,
   Slp(e, `init`) implies e.

3. *Proof of* e $\wedge$ Fail(`init`) *implies* d
   e states that all nodes are unmarked and unrooted, and labelled with an atom or concatenation of an atom and an integer. Also, that all edges are unmarked. Fail(`init`) stated that there is no unmarked unrooted node that is labelled with an atom. Hence, e $\wedge$ Fail(`init`) implies
   $\forall_{\mathsf{v}}\mathsf{x}(\mathsf{m_v}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge \exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))) \wedge \forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_e}(\mathsf{x}) = \mathsf{none})$,
   which is equal to *d*.

4. *Proof of* Slp(d,`change`) *implies* d
   Similar to point 2, Slp(d,`change`) implies
   $\forall_{\mathsf{v}}\mathsf{x}((\mathsf{m_v}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge \exists_{\mathsf{l}}\mathsf{a},\mathsf{k}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{k} \wedge \mathsf{int}(\mathsf{k}) \wedge \mathsf{atom}(\mathsf{a})))$
   $\vee\ (\mathsf{m_v}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge \exists_{\mathsf{l}}\mathsf{a},\mathsf{k}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{k}+1 \wedge \mathsf{int}(\mathsf{k}) \wedge \mathsf{atom}(\mathsf{a})))$
   $\vee\ (\mathsf{m_v}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge \exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))))$
   $\wedge\forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_e}(\mathsf{x}) = \mathsf{none})$, which clearly implies
   $\forall_{\mathsf{v}}\mathsf{x}(\mathsf{m_v}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge \exists_{\mathsf{l}}\mathsf{a},\mathsf{i}(\mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{atom}(\mathsf{a}))) \wedge \forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_e}(\mathsf{x}) = \mathsf{none})$.

5. *Proof of* d $\wedge$ Fail(`change`) *implies* post
   Since post=d$\wedge\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{edge}(\mathsf{x},\mathsf{y}) \Rightarrow \exists_{\mathsf{l}}\mathsf{a},\mathsf{b},\mathsf{i},\mathsf{j}(\mathsf{atom}(\mathsf{a}) \wedge \mathsf{atom}(\mathsf{b}) \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{int}(\mathsf{j})$
   $\wedge\, \mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{l_v}(\mathsf{y}) = \mathsf{b} : \mathsf{j} \wedge \mathsf{i} \neq \mathsf{j}))$,
   we need to proof that Fail(`change`) implies
   $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{edge}(\mathsf{x},\mathsf{y}) \Rightarrow \exists_{\mathsf{l}}\mathsf{a},\mathsf{b},\mathsf{i},\mathsf{j}(\mathsf{atom}(\mathsf{a}) \wedge \mathsf{atom}(\mathsf{b}) \wedge \mathsf{int}(\mathsf{i}) \wedge \mathsf{int}(\mathsf{j})$
   $\wedge\, \mathsf{l_v}(\mathsf{x}) = \mathsf{a} : \mathsf{i} \wedge \mathsf{l_v}(\mathsf{y}) = \mathsf{b} : \mathsf{j} \wedge \mathsf{i} \neq \mathsf{j}))$. From the semantics of quantifiers, we know that

Fail(change)$\equiv \forall_v x, y(x \neq y \wedge \mathsf{edge}(x,y) \Rightarrow \neg\exists_l a, b, i(\mathsf{atom}(a) \wedge \mathsf{atom}(b) \wedge \mathsf{int}(i)$
$\wedge\, l_v(x) = a : i \wedge l_v(y) = b : j)).$

The right hand side, with the knowledge that all nodes are labelled with the concatenation of an atom and an integer, implies
$\exists_l a, b, i, j(\mathsf{atom}(a) \wedge \mathsf{atom}(b) \wedge \mathsf{int}(i) \wedge \mathsf{int}(j) \wedge l_v(x) = a : i \wedge l_v(y) = b : j \wedge i \neq j))$. Hence, d $\wedge$ Fail(change) *implies* post.

8.1.4 Comparison with E-conditions

In [1], there is a proof for the same graph program (`vertex-colouring` of Figure 8.1). The proof can be seen in Figure 8.3.

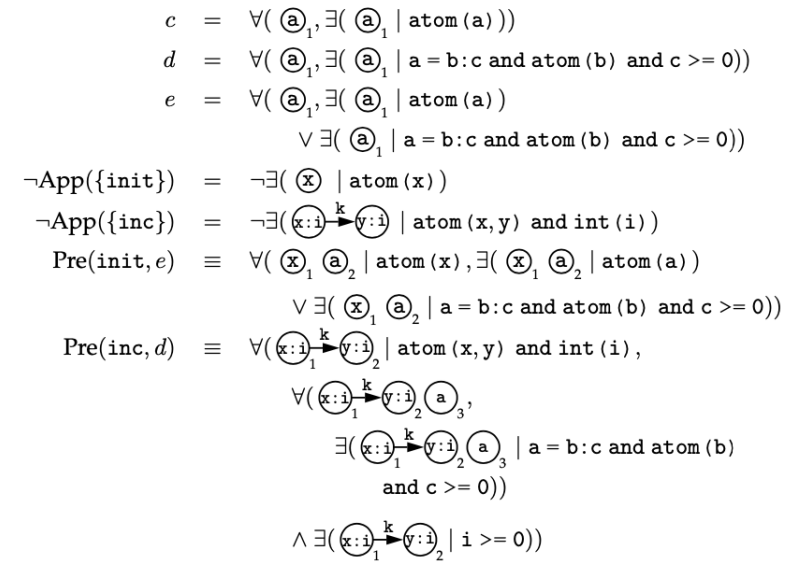


FIGURE 8.3: The partial correctness proof of `vertex-colouring` with E-condition [1]

Note that in the proof of Figure 8.3, there is an assumption that the input graph will never contain a marked node so that morphisms with marked nodes are not considered in the proof. If it is considered, the conditions would be much longer.

In this case, there may be no significant difference between the readability of MSO formulas we use in Table 8.1 and the E-conditions in Figure 8.3. However, Pre(`inc`,d) may not be easy to comprehend by some reader because the same construction gets repeated. Such repetition does not occur in MSO.

8.2 Transitive closure

8.2.1 Graph program `transitive-closure`

The second example we use is a graph program to compute transitive closure of a graph, as can be seen in Figure 8.4. The program takes a graph where all nodes are unmarked and unrooted and all edges are unmarked as an input. Then, the rule `link` adds an edge from node x to node z if they is a node y such that there exists an edge from x to y and from y to z and there is no edge from x to z. This process is executed repeatedly until there is no match for the rule.

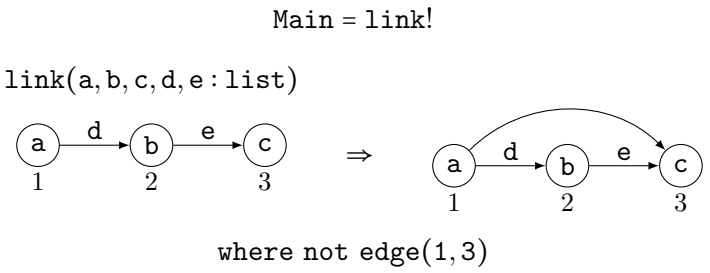


FIGURE 8.4: Graph program `transitive-closure`

Here, we want to show that the program `transitive-closure` is partially correct with respect to the following pre- and postcondition.

Precondition:
All nodes and edges are unmarked, and all nodes are unrooted.
Postcondition:
All nodes and edges are unmarked, and all nodes are unrooted. Also, for all distinct nodes x and y, if there is a directed path from x to y then there is an edge from x to y.

8.2.2 Proof tree of `transitive-closure`

The proof of the partial correctness of `transitive-closure` is given by the proof tree of Figure 8.5, while the assertions that are used in the proof tree are defined in Table 8.2.

To obtain Slp(pre,`link`) as in Table 8.2, we first find the form of pre without abbreviation, that is:
$\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none} \vee \mathsf{root}(\mathsf{x})) \wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none})$.
Then, we use the steps defined in Chapter 6, that are:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{\text{pre}\}\ \texttt{link}\ \{\text{Slp(pre,}\texttt{link}\text{)}\}}[\text{ruleapp}]_{\text{slp}}}{\{\text{pre}\}\ \texttt{link}\ \{\text{pre}\}}[\text{cons}]}{\{\text{pre}\}\ \texttt{link!}\ \{\text{pre}\wedge\text{Fail(}\texttt{link}\text{)}\}}[\text{alap}]}{\{\text{pre}\}\ \texttt{link!}\ \{\text{post}\}}[\text{cons}]$$

FIGURE 8.5: Proof tree for partial correctness of ttttransitive-closure

TABLE 8.2: Conditions inside proof tree of `transitive-closure`

MSO formulas
$\text{pre} \equiv \forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x)) \wedge \forall_e x(\mathsf{m}_e(x) = \mathsf{none})$
$\text{post} \equiv \text{pre} \wedge \forall_v x, y(x \neq y \wedge \mathsf{path}(x,y) \Rightarrow \mathsf{edge}(x,y))$
$\text{Fail(}\texttt{link}\text{)} \equiv \neg\exists_v x, y, z(x \neq y \wedge x \neq z \wedge z \neq y \wedge \mathsf{edge}(x,y,\mathsf{none}) \wedge \mathsf{edge}(y,z,\mathsf{none}) \wedge \neg\mathsf{edge}(x,z))$
$\text{Slp(pre,}\texttt{link}\text{)} \equiv$ $\exists_v u, v, w(\mathsf{edge}(u,v,\mathsf{none}) \wedge \mathsf{edge}(v,w,\mathsf{none}) \wedge \mathsf{edge}(u,w,\mathsf{none})$ $\wedge\ \mathsf{m}_v(u) = \mathsf{none} \wedge \mathsf{m}_v(v) = \mathsf{none} \wedge \mathsf{m}_v(w) = \mathsf{none} \wedge \neg\mathsf{root}(u) \wedge \neg\mathsf{root}(v) \wedge \neg\mathsf{root}(w)$ $\wedge\ \forall_v x(x = u \vee x = v \vee x = w \vee (\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x)))$ $\wedge\ \exists_e a, b, c(\mathsf{s}(a) = u \wedge \mathsf{t}(a) = v \wedge \mathsf{s}(b) = v \wedge \mathsf{t}(b) = w \wedge \mathsf{s}(c) = u \wedge \mathsf{t}(c) = w$ $\wedge\ \forall_e y(y = a \vee y = b \vee y = c \vee (\mathsf{s}(y) \neq u \wedge \mathsf{t}(y) \neq w)))$ $\wedge \forall_e x(\mathsf{m}_e(x) = \mathsf{none})$

1. Obtain $\text{Lift}_{\mathsf{MSO}}$(pre,`link`)

   (a) From Definition 6.2, $\text{Split}_{\mathsf{MSO}}$(pre,`link`) is:
   $\neg(\mathsf{m}_v(1) \neq \mathsf{none} \vee \mathsf{root}(1)) \wedge\ \neg(\mathsf{m}_v(2) \neq \mathsf{none} \vee \mathsf{root}(2)) \wedge\ \neg(\mathsf{m}_v(3) \neq \mathsf{none} \vee \mathsf{root}(3))$
   $\wedge \neg\exists_v x(x \neq 1 \wedge x \neq 2 \wedge x \neq 3 \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x)))$
   $\wedge \neg(\mathsf{m}_e(e_1) \neq \mathsf{none}) \wedge \neg(\mathsf{m}_e(e_2) \neq \mathsf{none}) \wedge \neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{m}_e(x) \neq \mathsf{none})$.
   Such that $\text{Val}(\text{Split}_{\mathsf{MSO}}(\text{pre,}\texttt{link}\text{),}\texttt{link}\text{)}$ is:
   $\wedge \neg\exists_v x(x \neq 1 \wedge x \neq 2 \wedge x \neq 3 \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x)))$
   $\wedge \neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{m}_e(x) \neq \mathsf{none})$.

   (b) The rule application condition for `link` is $\Gamma = \neg\mathsf{edge}(1,3)$, which is equivalent to $\neg\exists_e x(s(x) = 1 \wedge t(x) = 3)$. Hence, $\text{Split}_{\mathsf{MSO}}(\Gamma\text{,}\texttt{link}\text{)}$ is
   $\neg(\mathsf{s}(e_1) = 1 \wedge \mathsf{t}(e_1) = 3) \wedge\ \neg(\mathsf{s}(e_2) = 1 \wedge \mathsf{t}(e_2) = 3)$
   $\wedge \neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{s}(x) = 1 \wedge \mathsf{t}(x) = 3)$
   such that $\text{Val}(\text{Split}_{\mathsf{MSO}}(\Gamma\text{,}\texttt{link}\text{),}\texttt{link}\text{)}$ is:
   $\neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{s}(x) = 1 \wedge \mathsf{t}(x) = 3)$.

   (c) There is no node that gets deleted by `link`, such that $\text{Dang(}\texttt{link}\text{)} = \mathsf{true}$ such that $\text{Lift}_{\mathsf{MSO}}$(pre,`link`) is:
   $\neg\exists_v x(x \neq 1 \wedge x \neq 2 \wedge x \neq 3 \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x)))$
   $\wedge \neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{m}_e(x) \neq \mathsf{none})$
   $\wedge \neg\exists_e x(x \neq e_1 \wedge x \neq e_2 \wedge \mathsf{s}(x) = 1 \wedge \mathsf{t}(x) = 3)$

2. Obtain $\text{Shift}_{\mathsf{MSO}}$(pre,`link`)

(a) From Definition 6.10, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{pre},\texttt{link}))$ is:
$\wedge \neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge \mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none} \vee \mathsf{root}(\mathsf{x})))$
$\wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e}_1 \wedge \mathsf{x} \neq \mathsf{e}_2 \wedge \mathsf{x} \neq \mathsf{e}_3 \wedge \mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none})$
$\wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e}_1 \wedge \mathsf{x} \neq \mathsf{e}_2 \wedge \mathsf{x} \neq \mathsf{e}_3 \wedge \mathsf{s}(\mathsf{x}) = 1 \wedge \mathsf{t}(\mathsf{x}) = 3)$

(b) From Definition 6.13, $\mathrm{Shift}_{\mathsf{MSO}}(\mathrm{pre},\texttt{link})$ is:
$\wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e}_1 \wedge \mathsf{x} \neq \mathsf{e}_2 \wedge \mathsf{x} \neq \mathsf{e}_3 \wedge \mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none})$
$\wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{x} \neq \mathsf{e}_1 \wedge \mathsf{x} \neq \mathsf{e}_2 \wedge \mathsf{x} \neq \mathsf{e}_3 \wedge \mathsf{s}(\mathsf{x}) = 1 \wedge \mathsf{t}(\mathsf{x}) = 3)$
$\wedge \mathsf{m}_{\mathsf{v}}(1) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(2) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(3) = \mathsf{none} \wedge \neg\mathsf{root}(1) \wedge \neg\mathsf{root}(2) \wedge \neg\mathsf{root}(3)$
$\wedge \mathsf{edge}(1,2,\mathsf{none}) \wedge \mathsf{edge}(2,3,\mathsf{none}) \wedge \mathsf{edge}(1,3,\mathsf{none}).$

3. Obtain Slp(pre,`link`)
Here, we only need to change all node and edge identifiers to fresh variables, then bound all free variables by existential quantifiers. Hence, we obtain Slp(pre,`link`) as written in Table 8.2.

8.2.3 Proof of implications

Again, in the proof tree we use the proof rule [cons] several times so we need to provide the proof of implications used in the proof rule when it is necessary (i.e. when it is not obvious). Here, we show the proof of implications we use in the proof tree, that are: Slp(pre,`link`) implies pre and pre∧Fail(`link`) implies post.

1. *Proof of* Slp(pre,`link`) *implies* pre
Let us consider the subformula of Slp(pre,`link`):
$\mathsf{m}_{\mathsf{v}}(\mathsf{u}) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{v}) = \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{w}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{u}) \wedge \neg\mathsf{root}(\mathsf{v}) \wedge \neg\mathsf{root}(\mathsf{w})$
$\wedge\forall_{\mathsf{v}}\mathsf{x}(\mathsf{x} = \mathsf{u} \vee \mathsf{x} = \mathsf{v} \vee \mathsf{x} = \mathsf{w} \vee (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))).$
From $\mathsf{x} = \mathsf{u}$ and $\mathsf{m}_{\mathsf{v}}(\mathsf{u}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{u})$, $\mathsf{x} = \mathsf{v}$ and $\mathsf{m}_{\mathsf{v}}(\mathsf{v}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{v})$, also $\mathsf{x} = \mathsf{w}$ and $\mathsf{m}_{\mathsf{v}}(\mathsf{w}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{w})$, we can say that the subformula above implies
$\forall_{\mathsf{v}}\mathsf{x}((\mathsf{m}_{\mathsf{v}}(\mathsf{u}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{u}))$
$\vee (\mathsf{m}_{\mathsf{v}}(\mathsf{v}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{v}))$
$\vee (\mathsf{m}_{\mathsf{v}}(\mathsf{w}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{w}))$
$\vee (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))),$
which clearly implies $\forall_{\mathsf{v}}\mathsf{x}(\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))$. Hence, Slp(pre,`link`) implies pre.
2. *Proof of* pre∧Fail(`link`) *implies* c
Note that pre∧Fail(`link`) implies
$\mathrm{pre}\wedge\forall_{\mathsf{v}}\mathsf{x},\mathsf{y},\mathsf{z}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z} \wedge \mathsf{y} \neq \mathsf{z} \wedge \mathsf{edge}(\mathsf{x},\mathsf{y}) \wedge \mathsf{edge}(\mathsf{y},\mathsf{z}) \Rightarrow \mathsf{edge}(\mathsf{x},\mathsf{z})).$
It is obvious that the formula above implies pre, so that to prove that it implies post, we only need to show that it also implies $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{path}(\mathsf{x},\mathsf{y}) \Rightarrow \mathsf{edge}(\mathsf{x},\mathsf{y}))$.
From Lemma 3.10, we know that the predicate $\mathsf{x} \neq \mathsf{y} \wedge \mathsf{path}(\mathsf{x},\mathsf{y})$ can be expressed by:

$\exists_{\mathsf{E}}\mathsf{X}(\exists_{\mathsf{e}}\mathsf{u},\mathsf{v}(\mathsf{u}\in\mathsf{X}\wedge\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{u})=\mathsf{x}\wedge\mathsf{t}(\mathsf{v})=\mathsf{y})$
$\wedge\ \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u}\in\mathsf{X}\Rightarrow(\mathsf{t}(\mathsf{u})=\mathsf{y}\vee\exists_{\mathsf{e}}\mathsf{v}(\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{v})=\mathsf{t}(\mathsf{u}))))$,
which is equivalent to $\exists_{\mathsf{I}}\mathsf{n}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n}))$, where $\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})$ is the formula:
$\exists_{\mathsf{V}}\mathsf{X}(\mathsf{card}(\mathsf{X})=\mathsf{n}\wedge\exists_{\mathsf{e}}\mathsf{u},\mathsf{v}(\mathsf{u}\in\mathsf{X}\wedge\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{u})=\mathsf{x}\wedge\mathsf{t}(\mathsf{v})=\mathsf{y})$
$\wedge\ \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u}\in\mathsf{X}\Rightarrow(\mathsf{t}(\mathsf{u})=\mathsf{y}\vee\exists_{\mathsf{e}}\mathsf{v}(\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{v})=\mathsf{t}(\mathsf{u}))))$.
From the meaning of implication, we know that $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\exists_{\mathsf{I}}\mathsf{n}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n}))\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y}))$ is true iff $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y}))$ is true for all possible n.
Assuming that $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y},\mathsf{z}(\mathsf{x}\neq\mathsf{y}\wedge\mathsf{x}\neq\mathsf{z}\wedge\mathsf{y}\neq\mathsf{z}\wedge\mathsf{edge}(\mathsf{x},\mathsf{y})\wedge\mathsf{edge}(\mathsf{y},\mathsf{z})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{z}))$ is true, we show by induction on n that $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y}))$ is true for any n as well.
Base case (n=0),
If $n=0$, it is obvious that $\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})$ is false because
$\mathsf{card}(\mathsf{X})=0\wedge\exists_{\mathsf{e}}\mathsf{u},\mathsf{v}(\mathsf{u}\in\mathsf{X}\wedge\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{u})=\mathsf{x}\wedge\mathsf{t}(\mathsf{v})=\mathsf{y})$ is equivalent to false. Hence,
$\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y})$ is true for any x,y.
Base case (n=1),
$\mathsf{P}(\mathsf{x},\mathsf{y},1)$ implies $\mathsf{card}(\mathsf{X})=1\wedge\exists_{\mathsf{e}}\mathsf{u},\mathsf{v}(\mathsf{u}\in\mathsf{X}\wedge\mathsf{v}\in\mathsf{X}\wedge\mathsf{s}(\mathsf{u})=\mathsf{x}\wedge\mathsf{t}(\mathsf{v})=\mathsf{y})$.
If $\mathsf{P}(\mathsf{x},\mathsf{y},1)$ is true, then since the cardinality of X is 1, the source of the edge in X must be x and its target must be y.
Hence, $\mathsf{edge}(\mathsf{x},\mathsf{y})$ is true.
Inductive case.
Assuming for some $k>1$, $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{k})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y}))$ is true. We show that
$\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{k}+1)\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{y}))$ is true as well.
If $\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{k}+1)$ is true, then $\exists_{\mathsf{e}}\mathsf{u}(\mathsf{u}\in\mathsf{X}\wedge\mathsf{t}(\mathsf{u})=\mathsf{y})$ is true. Let u be the edge in X whose target is y and let s be the source of u. Then from the hypothesis assumption, $\mathsf{P}(\mathsf{x},\mathsf{u},\mathsf{k})$ implies $\mathsf{edge}(\mathsf{x},\mathsf{u})$, which means
$\mathsf{P}(\mathsf{x},\mathsf{u},\mathsf{k})\wedge\mathsf{edge}(\mathsf{u},\mathsf{y})\Rightarrow\mathsf{edge}(\mathsf{x},\mathsf{u})\wedge\mathsf{edge}(\mathsf{u},\mathsf{y})$. Since $x\neq y$, from the premise we know that $\mathsf{edge}(\mathsf{x},\mathsf{y})$ is true as well.

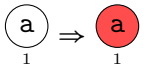
8.3 Unrooted 2-colouring

8.3.1 Graph program `2-colouring`

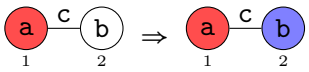
Next, we consider the graph program `2-colouring` of Figure 8.6. This program takes a graph whose all nodes and edges are unmarked and all nodes are unrooted. The execution of the program starts with marking one node into red, and repeatedly mark unmarked adjacent nodes with different colour (blue or red), then repeat the procedure in case the graph is not a connected graph. Finally, the program checks if there exists two adjacent nodes that has the same colour. If so, then the program yields fail.

```
Main = (init; Colour!)!; if Illegal then fail
Colour = {col_blue, col_red}
Illegal = {ill_blue, ill_red}
```

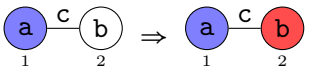
```
init(a : list)
```



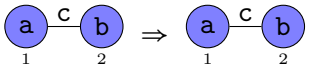

```
col_blue(a, b, c : list)
```




```
col_red(a, b, c : list)
```




```
ill_blue(a, b, c : list)
```




```
ill_red(a, b, c : list)
```

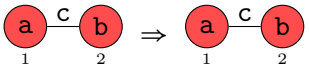


FIGURE 8.6: Graph program `2-colouring`

Here, we show two verifications of the program. One case is where the input graph is two-colourable, and the other case is where the input graph is non-two-colourable. Hence we have two specifications, namely pre- and postconditions A and B as written below. In both specifications, we argue that the postcondition is a stongest postcondition with respect to the given precondition.

Precondition A:
All nodes and edges are unmarked, and all nodes are unrooted. Also, the graph is two-colourable (the graph is bipartite).
Postcondition A:
All nodes are marked with blue or red, there is no rooted node, all edges are unmarked, and each two adjacent nodes are marked with different colour.

Precondition B:
All nodes and edges are unmarked, and all nodes are unrooted. Also, the graph is non-two-colourable (not a bipartite graph).
Postcondition B:
False.

Recall that in partial correctness calculus, a triple $\{pre\}\ C\ \{post\}$ is correct if the execution of program C on a graph satisfying *pre* results in a graph then the graph must satisfy *post*. It does not give us information about the case where the execution yields failure, get stuck, or diverge,

A program may get stuck or diverge when there is a loop body that does not terminate. For the program `2-colouring`, if we consider the command `Colour!`, it must be terminate because it reduces the number of unmarked node. Since `init` also reduces unmarked node, the command `(init; Colour!)!` must be terminate as well. Hence, the program `2-colouring` must be terminate.

Since the program `2-colouring` must be terminate, proving the triple {Precondition B} `2-colouring` {Postcondition B} is correct would mean that if `2-colouring` is executed on a graph satisfying Precondition B, then either it yields failure or it result in a graph satisfying false. Since there is no graph satisfying false, {Precondition B} `2-colouring` {Postcondition B} is correct would mean that the execution of `2-colouring` on a graph satisfying Precondition B will yield failure.

Now let us consider the triple {Precondition A} `2-colouring` {Postcondition A}. Proving the triple is partially correct will only give us information about the case where the execution of `2-colouring` on a graph satisfying Precondition A yields a graph, but not the case where the execution yields failure. However, we argue argue in Section 8.3.2.1 that the execution of the program on a graph satisfying Precondition A will not yield fail.

8.3.2 Case A: 2-colourable input graph

8.3.2.1 Proof tree of `2-colouring` (A)

The proof is presented in the proof tree of Figure 8.7, where assertions used in the proof tree are defined in Table 8.3. In the table, we use predicates $\mathsf{adj}(\mathsf{x},\mathsf{y})$ and $\mathsf{set}(\mathsf{x})$ which is defined below:
$\mathsf{adj}(\mathsf{x},\mathsf{y}) = \mathsf{edge}(\mathsf{x},\mathsf{y}) \vee \mathsf{edge}(\mathsf{y},\mathsf{x})$
$\mathsf{set}(\mathsf{x}) = (\mathsf{x} \in \mathsf{X} \vee \mathsf{x} \in \mathsf{Y}) \wedge \neg(\mathsf{x} \in \mathsf{X} \wedge \mathsf{x} \in \mathsf{Y})$

As mentioned above, with the specification in case A, we cannot have the information about failing case in the execution of `2-colouring` on a graph satisfying Precondition A. However, if we consider the semantics of the program, $\mathsf{fail} \in [\![\texttt{2-colouring}, G]\!]$ for some graph G iff there exist a graph H such that $< (\texttt{init};\texttt{Colour!})!, G >\rightarrow^* H$ and H satisfies Success(`Illegal`). Since `(init;Colour!)!` is a loop, it cannot fail. Also, we argue that it can not get stuck or diverge because both `init` and `Colour` reduce unmarked node which are used in the matching of every rules in the loop body. Hence, we can conclude that $\mathsf{fail} \in [\![\texttt{2-colouring}]\!]$ iff the output graph of the loop satisfies Success(`Illegal`).

From the proof tree we can see that the triple $\{pre\}$ $(\texttt{init};\texttt{Colour!})!$ $\{post\}$ is partially correct. Since we have convince ourselves that the loop cannot fail, get stuck or diverge,

TABLE 8.3: Conditions in the proof tree of `2-colouring` (A)

$pre \equiv$ $\forall_V x(m_v(x) = \mathsf{none} \wedge \neg \mathsf{root}(x)) \wedge \forall_E x(m_e(x) = \mathsf{none})$ $\wedge \exists_V X, Y(\forall_v x(\mathsf{set}(X))) \wedge \neg \exists_v x, y(x \neq y \wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \mathsf{adj}(x, y)))$
$post \equiv$ $\forall_V x(\neg \mathsf{root}(x)) \wedge \forall_E x(m_e(x) = \mathsf{none})$ $\wedge \exists_V X, Y(\forall_v x(\mathsf{set}(x) \wedge (x \in X \Rightarrow m_v(x) = \mathsf{red}) \wedge (x \in Y \Rightarrow m_v(x) = \mathsf{blue}))$ $\wedge \neg \exists_v x, y(x \neq y \wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \mathsf{adj}(x, y))$
$c \equiv$ $\forall_v x((m_v(x) = \mathsf{none} \vee m_v(x) = \mathsf{blue} \vee m_v(x) = \mathsf{red}) \wedge \neg \mathsf{root}(x) \wedge \forall_e x(m_e(x) = \mathsf{none})$ $\wedge \exists_V X, Y(\forall_v x(\mathsf{set}(x)) \wedge \forall_v x, y(x = y \vee (x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y) \Rightarrow \neg(\mathsf{adj}(x, y)))))$ $\wedge \forall_v x((x \in X \Rightarrow m_v(x) \neq \mathsf{blue} \wedge (x \in Y \Rightarrow m_v(x) \neq \mathsf{red})))$
$\mathrm{Slp}(c, \mathtt{init}) \equiv$ $\exists_v z(m_v(z) = \mathsf{red} \wedge l_v(z) = a \wedge \neg \mathsf{root}(z)$ $\wedge \forall_v x(x = z \vee (m_v(x) = \mathsf{none} \vee m_v(x) = \mathsf{blue} \vee m_v(x) = \mathsf{red}) \wedge \neg \mathsf{root}(x)))$ $\wedge \forall_e x(m_e(x) = \mathsf{none})$ $\wedge \exists_V X, Y(\mathsf{set}(z) \wedge \forall_v x(x = z \vee \mathsf{set}(x))$ $\wedge \forall_v x, y(x \neq z \wedge y \neq x \wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \Rightarrow \neg \mathsf{adj}(x, y))$ $\wedge \forall_v x, y(x = z \vee (x \in X \Rightarrow m_v(x) \neq \mathsf{blue}) \wedge (x \in Y \Rightarrow m_v(x) \neq \mathsf{red}))$ $\wedge (z \in X \wedge z \notin Y \Rightarrow \forall_v y(y \neq z \wedge y \in X \Rightarrow \neg \mathsf{adj}(y, z)))$ $\wedge (z \notin X \wedge z \in Y \Rightarrow \forall_v y(y \neq z \wedge y \in Y \Rightarrow \neg \mathsf{adj}(y, z)))$
$\mathrm{Slp}(c, \mathtt{Colour}) \equiv$ $\exists_v a, b(a \neq b \wedge (\mathsf{edge}(a, b, \mathsf{none}) \vee \mathsf{edge}(b, a, \mathsf{none})) \wedge \neg \mathsf{root}(a) \wedge \neg \mathsf{root}(b)$ $\wedge ((m_v(a) = \mathsf{red} \wedge m_v(b) = \mathsf{blue}) \vee (m_v(a) = \mathsf{blue} \wedge m_v(b) = \mathsf{red}))$ $\wedge \forall_v x(x = a \vee x = b \vee (\neg \mathsf{root}(x) \wedge (m_v(x) = \mathsf{none} \vee m_v(x) = \mathsf{red} \vee m_v(x) = \mathsf{blue})))$ $\wedge \forall_e x(m_v(x) = \mathsf{none})$ $\wedge \exists_V X, Y(\mathsf{set}(a) \wedge \mathsf{set}(b) \wedge \neg(a \in X \wedge b \in X) \wedge \neg(a \in Y \wedge b \in Y)$ $\wedge (a \in X \wedge b \notin X \wedge a \notin Y \wedge b \in Y$ $\Rightarrow \forall_v x(x = a \vee x = b \vee \mathsf{set}(x))$ $\wedge \forall_v x, y((x = a \wedge y = b) \vee (x = b \wedge y = a)$ $\vee (x = y \vee ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y) \Rightarrow \neg \mathsf{adj}(x, y))))$ $\wedge m_v(a) \neq \mathsf{blue} \wedge m_v(b) \neq \mathsf{red}$ $\wedge \forall_v x(x = a \vee x = b \vee ((x \in X \Rightarrow m_v(x) \neq \mathsf{blue}) \wedge (x \in Y \Rightarrow m_v(x) \neq \mathsf{red}))))$ $\wedge (a \notin X \wedge b \in X \wedge a \in Y \wedge b \notin Y$ $\Rightarrow \forall_v x(x = a \vee x = b \vee \mathsf{set}(x))$ $\wedge \forall_v x, y((x = a \wedge y = b) \vee (x = b \wedge y = a)$ $\vee (x = y \vee ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y) \Rightarrow \neg \mathsf{adj}(x, y))))$ $\wedge m_v(a) \neq \mathsf{red} \wedge m_v(b) \neq \mathsf{blue}$ $\wedge \forall_v x(x = a \vee x = b \vee ((x \in X \Rightarrow m_v(x) \neq \mathsf{blue}) \wedge (x \in Y \Rightarrow m_v(x) \neq \mathsf{red})))))$
$\mathrm{Fail}(\mathtt{Colour})$ $\equiv \neg \exists_e x((((m_v(s(x)) = \mathsf{red} \vee m_v(s(x)) = \mathsf{blue}) \wedge m_v(t(x)) = \mathsf{none})$ $\vee ((m_v(t(x)) = \mathsf{red} \vee m_v(t(x)) = \mathsf{blue}) \wedge m_v(s(x)) = \mathsf{none})) \wedge \neg \mathsf{root}(s(x)) \wedge \neg \mathsf{root}(t(x)))$
$\mathrm{Fail}(\mathtt{init}; \mathtt{Colour!}) \equiv \neg \exists_v x(m_v(x) = \mathsf{none} \wedge \neg \mathsf{root}(x))$
$\mathrm{Fail}(\mathtt{Illegal})$ $\equiv \neg \exists_e x(s(x) \neq t(x) \wedge ((m_v(s(x)) = \mathsf{red} \wedge m_v(t(x)) = \mathsf{red}) \vee (m_v(s(x)) = \mathsf{blue} \wedge m_v(t(x)) = \mathsf{blue})))$
$\mathrm{Success}(\mathtt{Illegal})$ $\equiv \exists_e x(s(x) \neq t(x) \wedge ((m_v(s(x)) = \mathsf{red} \wedge m_v(t(x)) = \mathsf{red}) \vee (m_v(s(x)) = \mathsf{blue} \wedge m_v(t(x)) = \mathsf{blue})))$

$$[\text{comp})\ \frac{\text{Subtree I} \qquad \text{Subtree II}}{\{\ \mathit{pre}\ \}\ \texttt{2colouring}\ \{\ \mathit{post}\ \}}$$

where subtree I is:

$$\dfrac{\dfrac{\dfrac{}{\{\ c\ \}\ \texttt{init}\ \{\ \mathrm{Slp}(c,\texttt{init})\ \}}[\text{ruleapp}]_{\text{slp}}}{\{\ c\ \}\ \texttt{init}\ \{\ c\ \}}[\text{cons}] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\ c\ \}\ \texttt{Colour}\ \{\ \mathrm{Slp}(c,\texttt{Colour})\ \}}[\text{ruleapp}]_{\text{slp}}}{\{\ c\ \}\ \texttt{Colour}\ \{\ c\ \}}[\text{cons}]}{\{\ c\ \}\ \texttt{Colour!}\ \{\ c \wedge \mathrm{Fail}(\texttt{Colour})\ \}}[\text{alap}]}{\{\ c\ \}\ \texttt{Colour!}\ \{\ c\ \}}[\text{cons}]}{\{\ c\ \}\ \texttt{init;Colour!}\ \{\ c\ \}}\text{comp}$$
$$\dfrac{\dfrac{\{\ c\ \}\ (\texttt{init;Colour!})!\ \{\ c \wedge \mathrm{Fail}(\texttt{init;Colour!})\ \}}{\{\ \mathit{pre}\ \}\ (\texttt{init;Colour!})!\ \{\ \mathit{post}\ \}}[\text{cons}]}{}[\text{alap}]$$

and subtree II is:

$$\dfrac{\dfrac{\dfrac{}{\{\ \mathit{post} \wedge \mathrm{Success}(\texttt{Ill})\ \}\ \texttt{fail}\ \{\ \mathsf{false}\ \}}[\text{fail}]}{\{\ \mathit{post} \wedge \mathrm{Success}(\texttt{Ill})\ \}\ \texttt{fail}\ \{\ \mathrm{post}\ \}}[\text{cons}] \qquad \dfrac{\dfrac{}{\{\ \mathit{post} \wedge \mathrm{Fail}(\texttt{Ill})\ \}\ \texttt{skip}\ \{\ \mathit{post} \wedge \mathrm{Fail}(\texttt{Illegal})\ \}}[\text{skip}]}{\{\ \mathit{post} \wedge \mathrm{Fail}(\texttt{Ill})\ \}\ \texttt{skip}\ \{\ \mathrm{post}\ \}}[\text{cons}]}{\{\ \mathit{post}\ \}\ \texttt{if Illegal then fail!}\ \{\ \mathit{post}\ \}}[\text{if}]$$

FIGURE 8.7: Proof tree for partial correctness of `2-colouring` (A)

then it must produce a graph, which must satisfy *post*. We have proven that *post* implies ¬Fail(`Illegal`), so that we can conclude that the execution of the program on a graph satisfying Precondition A cannot fail and must resulting a graph satisfying Postcondition B).

In this section, we also use the transformation of conditions as defined in Chapter 6. For an illustration, here we show how we derive the formula Slp(c, `init`) as an example.

To obtain Slp(c,`init`) as in Table 8.3, we first find the form of c without abbreviation, that is:

$$\neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{blue} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{red}) \vee\ \mathsf{root}(\mathsf{x})) \wedge \neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none})$$
$$\wedge\ \exists_{\mathsf{V}}\mathsf{X},\mathsf{Y}(\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red})))$$

Then, we use the steps defined in Chapter 6, that are:

1. Obtain $\mathrm{Lift}_{\mathsf{MSO}}$($c$,`init`)

   (a) From Definition 6.2, $\mathrm{Split}_{\mathsf{MSO}}$(c,`init`) is:
   $$\neg((\mathsf{m}_{\mathsf{v}}(1) \neq \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(1) \neq \mathsf{blue} \wedge \mathsf{m}_{\mathsf{v}}(1) \neq \mathsf{red}) \vee \mathsf{root}(1))$$
   $$\wedge\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{blue} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{red}) \vee\ \mathsf{root}(\mathsf{x}))$$
   $$\wedge\neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none}) \wedge \exists_{\mathsf{V}}\mathsf{X},\mathsf{Y}($$
   $$(1 \in \mathsf{X} \wedge 1 \in \mathsf{Y} \Rightarrow$$
   $$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
   $$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$

$$\wedge (1 \in \mathsf{X} \wedge 1 \notin \mathsf{Y} \Rightarrow$$
$$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$
$$\wedge (1 \notin \mathsf{X} \wedge 1 \in \mathsf{Y} \Rightarrow$$
$$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$
$$\wedge (1 \notin \mathsf{X} \wedge 1 \notin \mathsf{Y} \Rightarrow$$
$$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$

Such that $\mathrm{Val}(\mathrm{Split}_{\mathsf{MSO}}(\mathrm{c},\texttt{init}),\texttt{init})$ is:

$$\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{x} \neq 1 \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{none} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{blue} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{red}) \vee\ \mathsf{root}(\mathsf{x}))$$
$$\wedge\neg\exists_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) \neq \mathsf{none}) \wedge \exists_{\mathsf{V}}\mathsf{X},\mathsf{Y}($$
$$\wedge (1 \in \mathsf{X} \wedge 1 \notin \mathsf{Y} \Rightarrow$$
$$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$
$$\wedge (1 \notin \mathsf{X} \wedge 1 \in \mathsf{Y} \Rightarrow$$
$$\neg\exists_{\mathsf{v}}\mathsf{x}(\neg\mathsf{set}(\mathsf{x})) \wedge \neg\exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \wedge (\mathsf{adj}(\mathsf{x},\mathsf{y})))))$$
$$\wedge\ \neg\exists_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue} \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red}))))$$

(b) The rule application condition for `init` is $\Gamma = \mathsf{true}$

(c) There is no node that gets deleted by `init`, such that $\mathrm{Dang}(\texttt{init}) = \mathsf{true}$ such that $\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init}) = \mathrm{Val}(\mathrm{Split}_{\mathsf{MSO}}(\mathrm{c},\texttt{init}),\texttt{init})$

2. Obtain $\mathrm{Shift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init})$

   (a) From Definition 6.10, $\mathrm{Adj}_{\mathsf{MSO}}(\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init})) = \mathrm{Lift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init})$ because there is no node or edge that gets deleted, and no node or edge that gets added by the rule.

   (b) From Definition 6.13, $\mathrm{Shift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init})$ is
   $\mathrm{Lift}_{\mathsf{MSO}}(\mathrm{c},\texttt{init}) \wedge \mathsf{m}_{\mathsf{v}}(1) = \mathsf{red} \wedge \neg\mathsf{root}(1) \wedge \mathsf{l}_{\mathsf{v}}(1) = \mathsf{a}$

3. Obtain $\mathrm{Slp}(\mathrm{c},\texttt{init})$
   Here, we only need to change all node and edge identifiers to fresh variables, then bound all free variables by existential quantifiers. Hence, we obtain $\mathrm{Slp}(\mathrm{c},\texttt{init})$ as written in Table 8.3.

8.3.2.2 Proof of implications

1. Proof of $\mathrm{Slp}(c,\texttt{init})$ implies c
   $\mathrm{Slp}(c,\texttt{init})$ implies all nodes except the node that is represented by z are marked with blue or red or unmarked, and all nodes are unrooted. However, $\mathrm{Slp}(c,\texttt{init})$ also implies z is red and unrooted as well, so that these implies $\forall_{\mathsf{v}}x((\mathsf{m}_{\mathsf{v}}(x) = none \vee \mathsf{m}_{\mathsf{v}}(x) = red \vee \mathsf{m}_{\mathsf{v}}(x) = blue) \wedge \neg root(x))$. Also, it is obvious that $\mathrm{Slp}(c,\texttt{init})$ implies

$\forall_e x(\mathsf{m_e}(x) = none)$ from the meaning of disjunction.

Slp(c,`init`) also implies the existence of set X and Y such that if x does not equal z, then the last line of c is true, and if $x = z$, the last line of c is true as well due to the last two lines of Slp(c,`init`).

2. Proof of Slp(c, `Colour`) implies c

   Slp(c, `Colour`) asserts that the node represented by a and b are either red or blue, and both nodes are unrooted. Hence, together with
   $\forall_v x(\mathsf{x = a} \vee \mathsf{x = b} \vee (\neg\mathsf{root(x)} \wedge (\mathsf{m_v(x) = none} \vee \mathsf{m_v(x) = red} \vee \mathsf{m_v(x) = blue})))$ of
   Slp(c, `Colour`), it implies that all nodes are unrooted, and either unmarked, red, or blue. This means that Slp(c, `Colour`) implies $\forall_v x((\mathsf{m_v(x) = none} \vee \mathsf{m_v(x) = blue} \vee \mathsf{m_v(x) = red}) \wedge \neg\mathsf{root(x)} \wedge \forall_e x(\mathsf{m_e(x) = none})$ since $\forall_e x(\mathsf{m_e(x) = none})$ is implied as well based on the meaning of conjunction.

   For some set of nodes X and Y, note that Slp(c, `Colour`) implies $a \neq b$, $\mathsf{adj(a, b)}$, and $\neg(\mathsf{a} \in \mathsf{X} \wedge \mathsf{b} \in \mathsf{Y}) \wedge \neg(\mathsf{a} \in \mathsf{Y} \wedge \mathsf{b} \in \mathsf{Y})$ such that they imply if $(x = a \wedge y = b) \vee (x = b \wedge y = a)$ is true then $(\mathsf{x = y} \vee (\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \Rightarrow \neg\mathsf{adj(x, y)})$ is true as well. Hence, from the meaning of conjunction and implication, Slp(c, `Colour`) implies $\forall_v \mathsf{x, y}((\mathsf{x = y} \vee (\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y}) \Rightarrow \neg\mathsf{adj(x, y)}))$. Similarly, because Slp(c, `Colour`) implies $\mathsf{set(a)} \wedge \mathsf{set(b)}$ and also $\forall_v x(\mathsf{x = a} \vee \mathsf{x = b} \vee \mathsf{set(x)})$, Slp($c$, `Colour`) also implies $\forall_v x(\mathsf{set(x)})$.

   Slp(c, `Colour`) also asserts that in the case where $\mathsf{x} \neq \mathsf{a} \wedge \mathsf{x} \neq \mathsf{b}$, then if $\mathsf{x} \in \mathsf{X}$ then $\mathsf{m_v(x)} \neq \mathsf{blue}$ and if $\mathsf{x} \in \mathsf{Y}$ then $\mathsf{m_v(x)} \neq \mathsf{red}$, while also asserts that if $\mathsf{a} \in \mathsf{X}$ then $\mathsf{m_v(a)} \neq \mathsf{blue}$, $\mathsf{b} \in \mathsf{X}$ then $\mathsf{m_v(b)} \neq \mathsf{blue}$, $\mathsf{a} \in \mathsf{Y}$ then $\mathsf{m_v(a)} \neq \mathsf{red}$, and $\mathsf{b} \in \mathsf{Y}$ then $\mathsf{m_v(b)} \neq \mathsf{red}$. Hence, Slp($c$, `Colour`) also implies $\forall_v x(\mathsf{x} \in \mathsf{X} \Rightarrow$
   $\mathsf{m_v(x)} \neq \mathsf{blue} \wedge \mathsf{x} \in \mathsf{Y} \Rightarrow \mathsf{m_v(y)} \neq \mathsf{red})$.

3. Proof of *pre* implies c

   pre obviously implies the first two lines of c. Then, since *pre* implies that all nodes are unmarked, then all nodes in X or Z must not be coloured so that the last line of c is also implied.

4. Proof of $c \wedge$Fail(`init`) implies *post*

   c implies that all nodes are unmarked or marked with blue or red, while Fail(`init` implies that all nodes are marked. Hence, it implies that all nodes are blue or red. Since c implies all nodes in X are not blue and all nodes in Y are not red, these implies all nodes in X are red and all nodes in Y are blue. Hence, $c \wedge$Fail(`init`) implies *post*.

$$[\text{comp})\ \frac{\text{Subtree I} \qquad \text{Subtree II}}{\{\ pre\ \}\ \texttt{2colouring}\ \{\ post\ \}}$$

where subtree I is:

$$\dfrac{\dfrac{\dfrac{}{\{\ c\ \}\ \texttt{init}\ \{\ \mathrm{Slp}(c,\texttt{init})\ \}}[\text{ruleapp}]_{\text{slp}}}{\{\ c\ \}\ \texttt{init}\ \{\ c\ \}}[\text{cons}] \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\{\ c\ \}\ \texttt{Colour}\ \{\ \mathrm{Slp}(c,\texttt{Colour})\ \}}[\text{ruleapp}]_{\text{slp}}}{\{\ c\ \}\ \texttt{Colour}\ \{\ c\ \}}[\text{cons}]}{\{\ c\ \}\ \texttt{Colour!}\ \{\ c\ \}}[\text{alap}]}{\{\ c\ \}\ \texttt{Colour!}\ \{\ c \wedge \mathrm{Fail}(\texttt{Colour})\ \}}[\text{cons}]}{\{\ c\ \}\ \texttt{init;Colour!}\ \{\ c\ \}}\text{comp}$$
$$\dfrac{\dfrac{\{\ c\ \}\ (\texttt{init;Colour!})!\ \{\ c \wedge \mathrm{Fail}(\texttt{init;Colour!})\ \}}{\{\ pre\ \}\ (\texttt{init;Colour!})!\ \{\ d\ \}}[\text{cons}]}{}[\text{alap}]$$

and subtree II is:

$$\dfrac{\dfrac{\dfrac{}{\{\ d \wedge \mathrm{Success}(\texttt{Ill})\ \}\ \texttt{fail}\ \{\ \mathsf{false}\ \}}[\text{fail}]}{\{\ d \wedge \mathrm{Success}(\texttt{Ill})\ \}\ \texttt{fail}\ \{\ \mathrm{post}\ \}}[\text{cons}] \quad \dfrac{\dfrac{}{\{\ d \wedge \mathrm{Fail}(\texttt{Ill})\ \}\ \texttt{skip}\ \{\ d \wedge \mathrm{Fail}(\texttt{Illegal})\ \}}[\text{skip}]}{\{\ d \wedge \mathrm{Fail}(\texttt{Ill})\ \}\ \texttt{skip}\ \{\ \mathrm{post}\ \}}[\text{cons}]}{\{\ post\ \}\ \texttt{if Illegal then fail!}\ \{\ post\ \}}[\text{if}]$$

FIGURE 8.8: Proof tree for partial correctness of `2-colouring` (B)

8.3.3 Case B: non-2-colourable input graph

8.3.3.1 Proof tree of `2-colouring` (B)

The proof is presented in the proof tree of Figure 8.8, where assertions used in the proof tree are defined in Table 8.4, where assertions Fail used in the proof tree are the same as what written in Table 8.3. In the table, we use predicates adj and set as defined in the previous section.

8.3.3.2 Proof of implications

1. *Proof of* Slp(c,`init`) *implies* c
   The subformula $\forall_{\mathsf{v}}\mathsf{x}(\mathsf{x} = \mathsf{z} \vee (\neg\mathsf{root}(\mathsf{x}) \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue})))$ asserts that all nodes that is not represented by x are unrooted and either unmarked, blue, or red. However, Slp(c,`init`) also implies that the node re[resented by z is unrooted and red so that Slp(c,`init`) implies
   $\forall_{\mathsf{v}}\mathsf{x}(\neg\mathsf{root}(\mathsf{x}) \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue}))$.
   By the meaning of conjunction, it is obvious that Slp(c,`init`) also implies
   $\forall_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) = \mathsf{none})$.
   In addition, Slp(c,`init`) also implies the non-existence of set of nodes X and Y such that each node (including the node represented by z) is in X or Y but not both, and

TABLE 8.4: Conditions in the proof tree of `2-colouring` (B)

MSO formulas
pre ≡ $\forall_V x(m_v(x) = \text{none} \wedge \neg\text{root}(x)) \wedge \forall_E x(m_e(x) = \text{none})$ $\wedge\neg\exists_V X, Y(\forall_v x(\text{set}(x)) \wedge \neg\exists_v x, y(x \neq y \wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \text{adj}(x, y)))$
post≡ False
$c \equiv$ $\forall_v x((m_v(x) = \text{none} \vee m_v(x) = \text{blue} \vee m_v(x) = \text{red}) \wedge \neg\text{root}(x) \wedge \forall_e x(m_e(x) = \text{none})$ $\wedge\forall_V X, Y(\forall_v x(\text{set}(x)) \wedge \forall_v x((x \in X \Rightarrow m_v(x) \neq \text{blue}) \wedge (x \in Y \wedge m_v(x) \neq \text{red}))$ $\Rightarrow \exists_v x, y((x \neq z \wedge (x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \text{adj}(x, y)))$
$d \equiv \neg\text{Fail}(\texttt{Illegal})$
$\text{Fail}(\texttt{Illegal}) \equiv$ $\neg\exists_v x, y(\neg\text{root}(x) \wedge \neg\text{root}(y) \wedge ((m_v(x) = \text{red} \wedge m_v(y) = \text{red}) \vee (m_v(x) = \text{blue} \wedge m_v(y) = \text{blue}))$ $\wedge(\text{edge}(x, y, \text{none}) \vee \text{edge}(y, x, \text{none})))$
$\text{Success}(\texttt{Illegal}) \equiv \neg\ \text{Fail}(\texttt{Illegal})$
$\text{Fail}(\texttt{init; Colour!}) \equiv \neg\exists_v x(m_v(x) = \text{none} \wedge \neg\text{root}(x))$
$\text{Fail}(\texttt{Colour}) \equiv$ $\neg\exists_v x, y(\neg\text{root}(x) \wedge \neg\text{root}(y) \wedge ((m_v(x) = \text{red} \wedge m_v(y) = \text{none}) \vee (m_v(x) = \text{blue} \wedge m_v(y) = \text{none}))$ $\wedge(\text{edge}(x, y, \text{none}) \vee \text{edge}(y, x, \text{none})))$
$\text{Slp}(c, \texttt{init}) \equiv$ $\exists_v z(m_v(z) = \text{red} \wedge \neg\text{root}(z) \wedge \forall_e x(m_v(x) = \text{none})$ $\wedge \forall_v x(x = z \vee (\neg\text{root}(x) \wedge (m_v(x) = \text{none} \vee m_v(x) = \text{blue} \vee m_v(x) = \text{red})))$ $\wedge \neg\exists_V X, Y(((z \in X \wedge z \notin Y) \vee (z \in Y \wedge z \notin X)) \wedge \forall_v x(x = z \vee \text{set}(x))$ $\wedge \neg\exists_v x, y(x \neq z \wedge y \neq z \wedge y \neq z \wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \text{adj}(x, y))$ $\wedge \forall_v x(x = z \vee ((x \in X \Rightarrow m_v(x) \neq \text{blue}) \wedge (x \in Y \Rightarrow m_v(x) \neq \text{red}))$ $\wedge\ (z \in X \wedge z \notin Y \Rightarrow m_v(z) \neq \text{blue} \wedge \neg\exists_v x(x \in X \wedge \text{adj}(x, z)))$ $\wedge\ (z \notin X \wedge z \in Y \Rightarrow m_v(z) \neq \text{red} \wedge \neg\exists_v x(x \in Y \wedge \text{adj}(x, z))))$
$\text{Slp}(c, \texttt{Colour}) \equiv$ $\exists_v p, q(p \neq q \wedge (\text{edge}(p, q, \text{none}) \vee \text{edge}(q, p, \text{none}))$ $\wedge ((m_v(p) = \text{red} \wedge m_v(q) = \text{blue}) \vee (m_v(p) = \text{blue} \wedge m_v(q) = \text{red})) \wedge \neg\text{root}(p) \wedge \neg\text{root}(q)$ $\wedge \forall_v x(x = p \vee x = q \vee (\neg\text{root}(x) \wedge (m_v(x) = \text{none} \vee m_v(x) = \text{red} \vee m_v(x) = \text{blue})))$ $\wedge \forall_e x(m_e(x) = \text{none})$ $\wedge \exists_V X, Y(((p \in X \wedge q \in Y) \vee (p \in Y \wedge q \in X))$ $\wedge \forall_v x(x = p \vee x = q \vee \text{set}(x)) \wedge (p \notin X \vee q \notin X) \wedge (p \notin Y \vee q \notin Y))$ $\wedge \forall_v x(x = p \vee x = q \vee ((x \in X \Rightarrow m_v(x) \neq \text{blue}) \wedge (x \in Y \Rightarrow m_v(x) \neq \text{red})))$ $\wedge \neg\exists_v x, y(x \neq p \wedge x \neq q \wedge y \neq p \wedge y \neq q \wedge x \neq y$ $\wedge ((x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \text{adj}(x, y))$ $\wedge (p \in X \wedge q \in Y \Rightarrow m_v(p) \neq \text{blue} \wedge m_v(q) \neq \text{red})$ $\wedge (p \in Y \wedge q \in X \Rightarrow m_v(p) \neq \text{red} \wedge m_v(q) \neq \text{blue}))$

all nodes in X are not blue and all nodes in Y are not red. Also, Slp(c,`init`) implies that there is no two adjacent nodes belong to the same set. This means, Slp(c,`init`) implies

$\forall_V X, Y(\forall_v x(\text{set}(x)) \wedge \forall_v x((x \in X \Rightarrow m_v(x) \neq \text{blue}) \wedge (x \in Y \wedge m_v(x) \neq \text{red}))$
$\Rightarrow \exists_v x, y((x \neq z \wedge (x \in X \wedge y \in X) \vee (x \in Y \wedge y \in Y)) \wedge \text{adj}(x, y))).$

Hence, by the meaning of conjunction and implication, Slp(c,`init`) implies c.

2. *Proof of* Slp(c,`Colour`) *implies* c

Similarly as above, Slp(c,`Colour`) implies that if we not considering the node represented by p and q, if Slp(c,`Colour`) holds then c holds. However, since Slp(c,`Colour`) implies that the nodes represented by p and q are unrooted and either blue or red, then

the formula $\forall_{\mathsf{v}}\mathsf{x}(\neg\mathsf{root}(\mathsf{x}) \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue}))$ is implied by Slp(c,`Colour`). Also, $\forall_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) = \mathsf{none})$ is implied by Slp(c,`init`) from the meaning of conjunction.

Slp(c,`Colour`) also asserts the nonexistence of set of nodes X and Y such that when $\mathsf{set}(\mathsf{p})$ and $\mathsf{set}(\mathsf{q})$ holds, if $\mathsf{p} \in \mathsf{X}$ then p is not blue and no nodes adjacent to p in X and if $\mathsf{p} \in \mathsf{Y}$ then p is not red and no nodes adjacent to p in Y (and similarly for y). Hence, c still holds when we consider the nodes represented by p and q such that Slp(c,`Colour`) implies c.

3. *Proof of pre implies c*

   By the meaning of conjunction, it is obvious that *pre* implies $\forall_{\mathsf{e}}\mathsf{x}(\mathsf{m}_{\mathsf{e}}(\mathsf{x}) = \mathsf{none})$. By the meaning of disjunction, we also know that $\forall_{\mathsf{v}}\mathsf{x}(\neg\mathsf{root}(\mathsf{x}) \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none})$ of *pre* implies $\forall_{\mathsf{v}}\mathsf{x}(\neg\mathsf{root}(\mathsf{x}) \wedge (\mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{none} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m}_{\mathsf{v}}(\mathsf{x}) = \mathsf{blue}))$. In addition, since *pre* implies that all nodes are unmarked, it implies $\forall_{\mathsf{x}}((\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{blue}) \wedge (\mathsf{x} \in \mathsf{Y} \Rightarrow \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{red}))$ (since the formula is always true if all nodes are unmarked). Hence, by the meaning of conjunction and quantifiers, *pre* implies

   $$\forall_{\mathsf{V}}\mathsf{X},\mathsf{Y}(\forall_{\mathsf{v}}\mathsf{x}(\mathsf{set}(\mathsf{x})) \wedge \forall_{\mathsf{v}}\mathsf{x}((\mathsf{x} \in \mathsf{X} \Rightarrow \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{blue}) \wedge (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{m}_{\mathsf{v}}(\mathsf{x}) \neq \mathsf{red})) \Rightarrow \exists_{\mathsf{v}}\mathsf{x},\mathsf{y}((\mathsf{x} \neq \mathsf{z} \wedge (\mathsf{x} \in \mathsf{X} \wedge \mathsf{y} \in \mathsf{X}) \vee (\mathsf{x} \in \mathsf{Y} \wedge \mathsf{y} \in \mathsf{Y})) \wedge \mathsf{adj}(\mathsf{x},\mathsf{y})))$$

   as well so that *pre* implies c by the meaning of implication.

4. *Proof of* $c \wedge$ Fail(`init; Colour!`) *implies* $\neg$Fail(`Illegal`)

   Because Fail(`init;Colour!`) asserts that there is no unmarked node that is unrooted and c asserts that all nodes are rooted and either unmarked, red, or blue, then $c \wedge$ Fail(`init; Colour!`) implies $\forall_{\mathsf{v}}x(\neg root(x) \wedge (\mathsf{m}_{\mathsf{v}}(x) = red \vee \mathsf{m}_{\mathsf{v}}(x) = blue))$, i.e. all nodes are unrooted and either blue or red. If we consider the set of nodes X and Y where all red nodes are in X and all blue nodes are in Y, c asserts that there are two adjacent nodes that are belong to the same set, hence have the same colour (either blue or red). In addition, since c also asserts that all edges are unmarked, the two adjacent nodes must be connected by an unmarked edges. Hence, $\neg$Fail(`Illegal`) must be hold if c and Fail(`init;Colour!`) is true.

8.4 Connectedness

8.4.1 Graph program `connectedness`

In this section we consider the graph program `is-connected` as seen in Figure 8.9. The program is executed by checking the existence of an unrooted node with no marks and change it to a red rooted node. The program then execute depth first-search procedure by finding unrooted node that is adjacent with the red rooted node and change the node to

red, swap the rootedness, and mark the edge between them by dashed and repeat it as long as possible. The procedure continue by searching a red node that adjacent to red unrooted node by dashed edge and change the mark of the rooted node to grey while unmarking it, and move the root to the other node, then reply the procedure. Finally, the program checks if there still exists an unmarked node. If so, then the program yields fail.

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

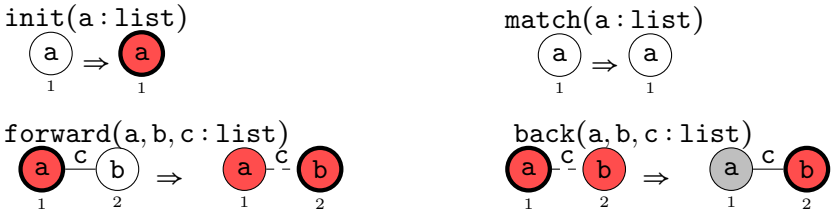


FIGURE 8.9: Graph program `is-connected`

Like in the previous section, here we give two sets of specifications. One is the case where the input graph is connected, and the other one is the case where the input graph is not connected. The first case should return an empty graph or a graph whose nodes are grey except for one red rooted node, while the latter case should return failure.

Precondition A:
All nodes and edges are unmarked, and all nodes are unrooted. Also, the graph is connected, that is, for every nodes x, y, there exists an undirected path[1] *from x to y)*
Postcondition A:
Either the graph is empty, or there is a node that is marked with red and is rooted while other nodes are grey and unrooted. All edges are unmarked, and the graph is connected.

Precondition B:
All nodes and edges are unmarked, and all nodes are unrooted. Also, there exist two distinct nodes x and y such that there is no undirected path from x to y.
Postcondition B:
False.

[1] undirected path is a path that does not consider the direction of the edges in the graph

Let us consider the semantics of the program `is-connected`. The program may get stuck or diverge if there is a loop that does not terminate. However, in this program, both `forward!` and `DFS!` must be terminate because the rules reduces the number of unmarked nodes and red nodes. Hence, the execution of `is-connected` on a graph G must either result in a proper graph, or failure.

Similiar to the previous section, here the postcondition *False* expresses that there is no graph that can be the result with the given precondition. Hence, if we prove that the triple {Precondition B} `is-connected` {Postcondition B} is partially correct, it will imply that the execution of `is-connected` on a graph satisfying Precondition B will not result in a graph, hence will yield failure. On the other hand, the triple {Precondition A} `is-connected` {Postcondition A} does not give us information about the case where the execution of `is-connected` on a graph satisfying Precondition A yields failure. However, we argue in 8.4.2.1 that such execution can not yield failure.

8.4.2 Case A: connected input graph

8.4.2.1 Proof tree of `is-connected` (A)

The proof is presented in the proof tree of Figure 8.10, where assertions used in the proof tree are defined in Table 8.5.

As mentioned above, with the specification in case A, we cannot have the information about failing case in the execution of `2-colouring` on a graph satisfying Precondition A. However, if we consider the semantics of the program, fail$\in [\![\texttt{is-connected}]\!]G$ for some graph G iff G satisfies Success(`init`) and there exist a graph H such that $< \texttt{init;DFS!}, G >\rightarrow^* H$ and H satisfies Success(`match`). Since `DFS!` is a loop, then it cannot fail. We also believe that the loop cannot get stuck or diverge. Hence, we can conclude that fail can not be in $[\![\texttt{is-connected}, G]\!]$ if when G satisfies Success(`init`) then the output graph of `init;DFS!` satisfies Success(`Illegal`).

From the proof tree we can see that the triple $\{pre\}$ `init` $\{c\}$ and $\{c\}$ `DFS!` $\{post\}$ are partially correct so that by the proof rule [comp] we can conclude that $\{pre\}$ `init;DFS!` $\{post\}$ is partially correct as well. We have proven that *post* implies ¬Fail(`match`), so that we can conclude that the execution of the program on a graph satisfying Precondition A cannot fail and must resulting a graph satisfying Postcondition B).

$$\dfrac{\text{[try]}\ \dfrac{\text{Subtree A} \qquad \dfrac{\text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{pre \wedge \text{Fail}(\texttt{init})\}}}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{post\}}}{\{\ pre\ \}\ \texttt{try init then (DFS!;Check)}\ \{\ post\ \}}}{}$$

where subtree A is:

$$\dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ pre\ \}\ \texttt{init}\ \{\ \text{Slp}(pre, \texttt{init})\ \}} \qquad \dfrac{\text{[cons]}\ \dfrac{\text{[alap]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ c\ \}\ \texttt{forward}\ \{\ \text{Slp}(c, \texttt{forward})\ \}}}{\{\ c\ \}\ \texttt{forward}\ \{\ c\ \}}}{\{\ c\ \}\ \texttt{forward!}\ \{\ c \wedge \text{Fail}(\texttt{forward})\ \}}}{\{\ c\ \}\ \texttt{forward!}\ \{\ d\ \}} \qquad \text{Subtree A1}}{\{\ c\ \}\ \texttt{DFS}\ \{\ c\ \}} \qquad \text{Break}(c, \texttt{DFS}, post)}{\{\ (c \wedge \text{Fail}(\texttt{DFS})) \vee post\ \}}}{\{\ c\ \}\ \texttt{DFS!}\ \{\ post\ \}}}{\{\ pre\ \}\ \texttt{init}\ \{\ c\ \}} \qquad \text{Subtree A2}}{\{\ pre\ \}\ \texttt{init;DFS!;Check}\ \{\ post\ \}}}{\{\ pre \wedge \text{Success}(\texttt{init})\ \}\ \texttt{init;DFS!;Check}\ \{\ post\ \}}}{}$$

for Subtree A1:

$$\dfrac{\text{[try]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ \text{Slp}(d \wedge \text{Success}(\texttt{back}), \texttt{back})\ \}}}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ c\ \}} \qquad \dfrac{\text{[cons]}\ \dfrac{\text{[break]}\ \dfrac{}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ d \wedge \text{Fail}(\texttt{back})\ \}}}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ c\ \}}}{\{\ d\ \}\ \texttt{try back else break}\ \{\ c\ \}}}{}$$

and Subtree A2:

$$\dfrac{\text{[if]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[fail]}\ \dfrac{}{\{\ \mathsf{false}\ \}\ \texttt{fail}\ \{\ \mathsf{false}\ \}}}{\{\ post \wedge \text{Success}(\texttt{match})\ \}\ \texttt{fail}\ \{\ \text{post}\ \}} \qquad \dfrac{\text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ post \wedge \text{Fail}(\texttt{match})\ \}\ \texttt{skip}\ \{\ post \wedge \text{Fail}(\texttt{match})\ \}}}{\{\ post \wedge \text{Fail}(\texttt{match})\ \}\ \texttt{skip}\ \{\ post\ \}}}{\{\ post\ \}\ \texttt{if match then fail}\ \{\ post\ \}}}{}$$

FIGURE 8.10: Proof tree for partial correctness of `is-connected` (A)

TABLE 8.5: Conditions inside proof tree of `is-connected` (A)

MSO formulas
$pre \equiv \forall_v x(m_v(x) = \text{none} \wedge \neg\text{root}(x)) \wedge \forall_E x(m_e(x) = \text{none})$ $\wedge\ \forall_V X(\forall_v x(x \in X) \vee \forall_v x(x \notin X) \vee \exists_v x, y(x \in X \wedge y \notin X \wedge \text{adj}(x, y)))$
$post \equiv \forall_v x(\text{false})$ $\vee\ (\exists_v x(m_v(x) = \text{red} \wedge \text{root}(x) \wedge \forall_v y(y = x \vee (m_v(y) = \text{grey} \wedge \neg\text{root}(y)))) \wedge \forall_e x(m_e(x) = \text{none})$ $\wedge\ \forall_V X(\forall_v x(x \in X) \vee \forall_v x(x \notin X) \vee \exists_v x, y(x \in X \wedge y \in X \wedge \text{adj}(x, y)))$
$c \equiv$ $\forall_e x(m_e(x) = \text{none} \vee m_e(x) = \text{dashed})$ $\wedge \exists_v x(\text{root}(x) \wedge m_v(x) = \text{red}$ $\wedge\ \forall_v y(x = y \vee ((m_v(y) = \text{none} \vee m_v(y) = \text{red} \vee m_v(y) = \text{grey}) \wedge \neg\text{root}(y)))$ $\wedge \forall_V X(\forall_v x(x \in X) \vee \forall_v x(x \notin X) \vee \exists_v x, y(x \in X \wedge y \notin X \wedge \text{adj}(x, y)))$ $\wedge \forall_e x(m_e(x) = \text{dashed} \Rightarrow m_v(s(x)) = \text{red} \wedge m_v(t(x)) = \text{red} \wedge (\neg\text{root}(s(x)) \vee \neg\text{root}(t(x))))$ $\wedge\ \forall_v z(m_v(z) = \text{red} \wedge \neg\text{root}(z) \Rightarrow \text{Success}(\text{back}))$ $\wedge \forall_v x(m_v(x) = \text{grey} \Rightarrow \neg\exists_v y(\text{adj}(x, y) \wedge \neg\text{root}(y) \wedge (m_v(y) = \text{none} \vee (m_v(y) = \text{red} \wedge \neg\text{root}(y)))))$
$d \equiv c \wedge \text{Fail}(\texttt{forward})$
$\text{Success}(\texttt{back}) \equiv$ $\exists_v x, y(m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{red} \wedge \neg\text{root}(y) \wedge (\text{edge}(x, y, \text{dashed}) \vee \text{edge}(y, x, \text{dashed})))$
$\text{Fail}(\texttt{back}) \equiv \neg\text{Success}(\texttt{back})$
$\text{Fail}(\texttt{init}) \equiv \neg\exists_v(m_v(x) = \text{none} \wedge \neg\text{root}(x))$
$\text{Fail}(\texttt{match}) \equiv \text{Fail}(\texttt{init})$
$\text{Success}(\texttt{match}) \equiv \neg\text{Fail}(\texttt{init})$
$\text{Fail}(\texttt{DFS}) \equiv \text{false}$
$\text{Fail}(\texttt{forward}) \equiv$ $\neg\exists_v x, y(m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge (\text{edge}(x, y, \text{none}) \vee \text{edge}(y, x, \text{none})))$
$\text{Slp}(pre, \texttt{init}) \equiv$ $\exists_v y(m_v(y) = \text{red} \wedge \text{root}(y) \wedge \forall_v x(x = y \vee (m_v(x) = \text{none} \wedge \neg\text{root}(x))))$ $\wedge \forall_e x(m_e(x) = \text{none})$ $\wedge \exists_v z(\forall_v x(x = z \vee (\text{upath}(x, z) \wedge \forall_v y(y = z \vee x = y \vee \text{upath}(x, y)))))$
$\text{Slp}(c, \texttt{forward}) \equiv$ $\exists_v a, b(\exists_e c(a \neq b \wedge m_v(a) = \text{grey} \wedge m_v(b) = \text{red} \wedge \neg\text{root}(a) \wedge \text{root}(b)$ $\wedge\ (s(c) = a \wedge t(c) = b) \vee (s(c) = b \wedge t(c) = a)) \wedge m_v(c) = \text{none}$ $\wedge\ f))$
$f \equiv$ $\forall_e x(x \neq c \Rightarrow m_e(x) = \text{none} \vee m_e(x) = \text{dashed})$ $\wedge \forall_v x(x \neq a \wedge x \neq b \Rightarrow \neg\text{root}(x) \wedge (m_v(x) = \text{red} \vee m_v(x) = \text{grey} \vee m_v(x) = \text{none}))$ $\wedge \neg\exists_V X(((a \in X \wedge b \in X) \vee (a \notin X \wedge b \notin X))$ $\wedge\ (a \in X \wedge b \in X \Rightarrow \exists_v x(x \notin X) \wedge \neg\exists_v x(x \notin X \wedge (\text{adj}(a, x) \vee \text{adj}(b, x)))$ $\wedge\ \neg\exists_v x, y(x \neq a \wedge x \neq b \wedge y \neq a \wedge y \neq b \wedge x \notin X \wedge (\text{adj}(x, y))))$ $\wedge\ (a \notin X \wedge b \notin X \Rightarrow \exists_v x(x \in X) \wedge \neg\exists_v x(x \in X \wedge (\text{adj}(a, x) \vee \text{adj}(b, x)))$ $\wedge\ \neg\exists_v x, y(x \neq a \wedge x \neq b \wedge y \neq a \wedge y \neq b \wedge x \notin X \wedge (\text{adj}(x, y)))))$ $\wedge \forall_e x(x \neq c \wedge m_e(x) = \text{dashed} \Rightarrow m_v(s(x)) = \text{red} \wedge m_v(t(x)) = \text{red} \wedge (\neg\text{root}(s(x)) \vee \neg\text{root}(t(x))))$ $\wedge\ \forall_v z(z \neq a \wedge z \neq b \wedge m_v(z) = \text{red} \wedge \neg\text{root}(z) \Rightarrow \text{Success}(\text{back}))$ $\wedge \forall_v x(x \neq a \wedge x \neq b \wedge m_v(x) = \text{grey}$ $\Rightarrow \neg\exists_v y(\text{adj}(x, y) \wedge \neg\text{root}(y) \wedge (m_v(y) = \text{none} \vee (m_v(y) = \text{red} \wedge \neg\text{root}(y)))))$
$\text{Slp}(d \wedge \text{Success}(back), \texttt{back}) \equiv$ $\exists_v a, b(\exists_e c(a \neq b \wedge m_v(a) = \text{grey} \wedge m_v(b) = \text{red} \wedge \neg\text{root}(a) \wedge \text{root}(b)$ $\wedge\ (s(c) = a \wedge t(c) = b) \vee (s(c) = b \wedge t(c) = a)) \wedge m_v(c) = \text{none}$ $\wedge\ f$ $\wedge\ \neg\exists_v y(y \neq a \wedge y \neq b \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge \text{adj}(b, y, \text{none}))$ $\wedge\ \neg\exists_v x, y(x \neq a \wedge x \neq b \wedge y \neq a \wedge y \neq b$ $\wedge\ m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge \text{adj}(b, y, \text{none})))$

8.4.2.2 Proof of implications

1. *Proof of* $pre \wedge \text{Fail}(\texttt{init})$ *implies post*
   From the meaning of conjunction, *pre* implies $\forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x))$, while Fail(`init`) implies $\forall_v x(\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x))$. From the meaning of universal quantifiers, we know that $\forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x)) \wedge \forall_v x(\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x))$ implies $\forall_v x(\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x) \wedge (\mathsf{m}_v(x) \neq \mathsf{none} \vee \mathsf{root}(x)))$, which implies $\forall_v x(\mathsf{false})$. Hence, it implies *post* from the meaning of disjunction.
2. *Proof of* $\text{Slp}(pre, \texttt{init})$ *implies* c
   From the meaning of disjunction, the formula
   $\exists_v y(\mathsf{m}_v(y) = \mathsf{red} \wedge \mathsf{root}(y) \wedge \forall_v x(x = y \vee (\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x))))$ implies
   $\exists_v x(\mathsf{root}(x) \wedge \mathsf{m}_v(x) = \mathsf{red} \wedge \forall_v y(x = y \vee \neg\mathsf{root}(y)))$. The formula also implies
   $\forall_v x(\mathsf{m}_v(x) = \mathsf{none} \vee \mathsf{m}_v(x) = \mathsf{red} \vee \mathsf{m}_v(x) = \mathsf{grey})$ because from the formula we know that the node represented by y is red and others are unmarked. The formula
   $\forall_e x(\mathsf{m}_e(x) = \mathsf{none})$ clearly implies $\forall_e x(\mathsf{m}_e(x) = \mathsf{none}) \vee \mathsf{m}_e(x) = \mathsf{dashed}$, while the formula $\forall_v x(x = 1 \vee (\mathsf{upath}(x, 1) \wedge \forall_v y(y = 1 \vee x = y \vee \mathsf{upath}(x, y))))$ implies
   $\forall_v x, y(x = y \vee \mathsf{upath}(x, y))$.
   Also, if $\text{Slp}(pre, \texttt{init})$ is true then the implications with
   $\forall_e x(\mathsf{m}_e(x) = \mathsf{dashed}$ or $(\exists_v x(\mathsf{m}_v(x) = \mathsf{red} \wedge \neg\mathsf{root}(x))$ as premise are always true because $\text{Slp}(pre, \texttt{init})$ implies $\forall_e x(\mathsf{m}_e(x) = \mathsf{none})$ which means that all edges are unmarked, and
   $\exists_v y(\mathsf{m}_v(y) = \mathsf{red} \wedge \mathsf{root}(y) \wedge \forall_v x(x = y \vee (\mathsf{m}_v(x) = \mathsf{none} \wedge \neg\mathsf{root}(x))))$ which means there is exactly one red rooted node while other nodes are unmarked and unrooted.
   Finally, $\text{Slp}(pre, \texttt{init})$ asserts that there is one rooted red node while other nodes are unmarked in addition to the existence of an undirected path between every node. Hence, if there is an unmarked node in the graph, the red rooted node must be adjacent to at least one unmarked node so that
   $\exists_v x(\mathsf{m}_v(x) = \mathsf{unmarked}) \Rightarrow \exists_v x, y(\mathsf{m}_v(x) = \mathsf{none} \wedge \mathsf{m}_v(y) = \mathsf{red} \wedge \mathsf{adj}(x, y)$ must hold.
3. *Proof of* $\text{Break}(c, \texttt{DFS}, post)$
   From the semantics of GP 2 commands, when we have the derivation $[\![DFS, G]\!] \rightarrow^* [\![\texttt{break}, H]\!]$ iff we have the derivation $[\![\texttt{forward!}, G]\!] \rightarrow^H$ for a graph H such that Fail(`back`) holds on H. From the proof tree of Figure 8.10, we know that $\{c\}$ `forward!` $\{c \wedge \text{Fail}(\texttt{forward})\}$ is correct so that H must imply $c \wedge \text{Fail}(\texttt{forward})$ as well. Hence, H must satisfy $c \wedge \text{Fail}(\texttt{forward}) \wedge \text{Fail}(\texttt{back})$ if the input graph G satisfies c.
   Let us consider the formula $\exists_v x(\mathsf{m}_v(x) = \mathsf{red} \wedge \neg\mathsf{root}(x)) \Rightarrow \text{Success}(\texttt{back})$ of c. If $c \wedge \text{Fail}(\texttt{forward}) \wedge \text{Fail}(\texttt{back})$ holds, then Success(`back`) must not hold so that from the meaning of implication, we know that $\exists_v x(\mathsf{m}_v(x) = \mathsf{red} \wedge \neg\mathsf{root}(x))$ does not hold either. By the meaning of implication and from the formula

$\forall_e x(\mathsf{m}_e(x) = \mathsf{dashed} \Rightarrow (\mathsf{m}_v(\mathsf{s}(x)) = \mathsf{red} \wedge \neg\mathsf{root}) \vee (\mathsf{m}_v(\mathsf{t}(x)) = \mathsf{red} \wedge \neg\mathsf{root}))$ of c, it implies that there is no dashed edge so that together with
$\forall_e x(\mathsf{m}_e(x) = \mathsf{none} \vee \mathsf{m}_e(x) = \mathsf{dashed})$, it implies $\forall_e x(\mathsf{m}_e(x) = \mathsf{none})$.
Now let us consider the formula
$\exists_v x(\mathsf{m}_v(x) = \mathsf{none}) \Rightarrow \exists_v x, y(\mathsf{m}_v(x) = \mathsf{red} \wedge \mathsf{m}_v(y) = \mathsf{none} \wedge \mathsf{adj}(x, y)$ of c. If there is no dashed edge and there is no red node other than a red rooted node, then from the formula we know that if an unmarked node exists then there is an unmarked node adjacent to the red unrooted node. However, Fail(`forward`) asserts the negation of that so that we can conclude that there is no unmarked node from the meaning of implication. Since we have also proven that $c \wedge$Fail(`forward`)$\wedge$Fail(`back`) implies that all nodes except one red rooted node are either unmarked or grey, then because there is no unmarked node, the following formula holds:
$\exists_v x(\mathsf{m}_v(x) = \mathsf{red} \wedge \mathsf{root}(x) \wedge \forall_v y(\mathsf{m}_v(y) = \mathsf{grey} \wedge \neg\mathsf{root}(y)))$.
Hence, $c \wedge$Fail(`forward`)$\wedge$Fail(`back`) implies *post* so that Break(c,`DFS`,*post*) holds.

4. *Proof of* Slp(c, `forward`) *implies* c
   Slp(c, `forward`) implies that if all edges that is not represented by c are unmarked or dashed, while c representing a dashed edge. Hence, Slp(c, `forward`) implies
   $\forall_e x(\mathsf{m}_e x = \mathsf{none} \vee \mathsf{m}_e x = \mathsf{dashed})$. Similarly, Slp($c$, `forward`) implies all nodes that are not represented by a and b are unrooted and either unmarked, red, or grey, where a and b representing grey unrooted node and red rooted node respectively. Hence, Slp(c, `forward`) implies $\exists_v x(\mathsf{m}_v(x) = red \wedge root(x) \wedge \forall_v y(x = y \vee (\neg root(y) \wedge (\mathsf{m}_v(y) = none \vee \mathsf{m}_v(y) = red \vee \mathsf{m}_v(y) = grey))))$.
   Slp(c, `forward`) also implies the nonexixtense of a set of nodes X such that both a and b both belong to (or not in) the set and there exists an edge that is not in (or in) the set, but there is no node outside (or in) X that is adjacent to a or b an there are no two adjacent nodes where one is in X and the other one is not in X. On the other words, $\forall_V X(\forall_v x(x \in X) \vee \forall_v x(x \notin X) \vee \exists_v x, y(x \in X \wedge y \notin X \wedge \mathsf{adj}(x, y)))$ is true if Slp(c, `forward`) is true.
   Note that Slp(c, `forward`) implies that for all edges that is not represented by c, if it is dashed then it is adjacent to a red unrooted node. However, Slp(c, `forward`) also implies that c is dashed and adjacent to red unrooted node a. Hence, Slp(c, `forward`) implies $\forall_e x(\mathsf{m}_e(x) = \mathsf{dashed} \Rightarrow \mathsf{m}_v(\mathsf{s}(x)) = \mathsf{red} \wedge \mathsf{m}_v(\mathsf{t}(x)) = \mathsf{red} \wedge (\neg\mathsf{root}(\mathsf{s}(x)) \vee \neg\mathsf{root}(\mathsf{t}(x))))$.
   Slp(c, `forward`) also implies that Success(`back`) holds, because we can use the nodes represented by a and b to satisfy it. Hence, if Slp(c, `forward`) is true then
   $\forall_v z(\mathsf{m}_v(z) = \mathsf{red} \wedge \neg\mathsf{root}(z) \Rightarrow \text{Success}(\mathsf{back}))$ always hold.
   Finally, Slp(c, `forward`) implies
   $\forall_v x(x \neq a \wedge x \neq b \wedge \mathsf{m}_v(x) = \mathsf{grey}$
   $\quad \Rightarrow \neg\exists_v y(\mathsf{adj}(x, y) \wedge \neg\mathsf{root}(y) \wedge (\mathsf{m}_v(y) = \mathsf{none} \vee (\mathsf{m}_v(y) = \mathsf{red} \wedge \neg\mathsf{root}(y)))))$
   Note that Slp(c, `forward`) also implies that both nodes represented by a and b are not

grey. Hence, the implication still holds for the case where $x = a$ or $x = b$. Hence, Slp(c, `forward`) implies

$\forall_v x(m_v(x) = \mathsf{grey} \Rightarrow \neg\exists_v y(\mathsf{adj}(x,y) \wedge \neg\mathsf{root}(y) \wedge (m_v(y) = \mathsf{none} \vee (m_v(y) = \mathsf{red} \wedge \neg\mathsf{root}(y)))))$

5. *Proof of* Slp($d \wedge$ **Success**(`back`), `back`) *implies* c
   From the meaning of conjunction, we know that Slp($d \wedge$ **Success**(`back`), `back`) implies f, and with the same reason as above (from the fact of a is grey unrooted node, b is red rooted node, and c is unmarked, we can show as in the previous point that Slp($d \wedge$ **Success**(`back`), `back`) implies c.
6. *Proof of* $post \wedge$ Success(`match`) *implies* false
   post clearly implies that all nodes are either red or grey, so there must not exist an unmarked node.

8.4.3 Case B: disconnected input graph

8.4.3.1 Proof tree of `is-connected` (B)

8.4.3.2 Proof of implications

1. Proof of $pre\wedge$ Fail(`init`) implies *post*
   From the meaning of conjunction, *pre* implies $\forall_v x(m_v(x) \wedge \neg\mathsf{root}(x))$, while Fail(`init`) stated that $\forall_v x(m_v(x) \neq \mathsf{none} \vee \mathsf{root}(x))$. By the meaning of universal quantifier, $pre\wedge$ Fail(`init`) then implies $\forall_v x(m_v(x) \wedge \neg\mathsf{root}(x)) \wedge (m_v(x) \neq \mathsf{none} \vee \mathsf{root}(x))$ which is equivalent to $\forall_v x(\mathsf{false})$. From the meaning of disjunction, it is obvious that $\forall_v x(\mathsf{false})$ implies *post*.

2. Proof of Slp(pre,`init`) implies c
   The formula $\forall_e x(m_e(x) = \mathsf{none})$ of Slp(pre,`init`) also clearly implies
   $\forall_e x(m_e(x) = \mathsf{none} \vee m_e(x) = \mathsf{dashed})$ of c by the meaning of disjunction.
   Also,
   $\wedge\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg\mathsf{adj}(x,y))$
   is directly implied by Slp(pre,`init`) from the meaning of conjunction.

   Slp(pre,`init`) stated that there exists a node that is red and rooted, while other nodes are unmarked and unrooted. It clearly implies
   $\exists_v x(\mathsf{root}(x) \wedge m_v(x) = \mathsf{red}$
   $\wedge \forall_v y(x = y \vee ((m_v(y) = \mathsf{none} \vee m_v(y) = \mathsf{red} \vee m_v(y) = \mathsf{grey}) \wedge \neg\mathsf{root}(y)))$.
   Since Slp(pre,`init`) expresses the existence of a red rooted node while other nodes are unmarked, the formula
   $\forall_v x(m_v(x) = \mathsf{grey} \Rightarrow \neg\exists_v y(\mathsf{adj}(x,y) \wedge \neg\mathsf{root}(y) \wedge (m_v(y) = \mathsf{none} \vee (m_v(y) = \mathsf{red} \wedge \neg\mathsf{root}(y)))))$
   of c is implied by Slp(pre,`init`) as well since there is no grey nodes if Slp(pre,`init`) is true.

$$\text{[try]}\ \frac{\text{Subtree A} \qquad \text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{pre \wedge \text{Fail}(\texttt{init})\}}}{\{\ pre \wedge \text{Fail}(\texttt{init})\}\ \texttt{skip}\ \{post\}}}{\{\ pre\ \}\ \texttt{try init then (DFS!;Check)}\ \{\ post\ \}}$$

where subtree A is:

$$\text{[comp]}\ \frac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ pre\ \}\ \texttt{init}\ \{\ \text{Slp}(pre, \texttt{init})\ \}}\ \text{[cons]}\ \dfrac{}{\{\ pre\ \}\ \texttt{init}\ \{\ c\ \}} \qquad \text{[cons]}\ \dfrac{\text{[alap]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[alap]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ c\ \}\ \texttt{forward}\ \{\ \text{Slp}(c, \texttt{forward})\ \}}}{\{\ c\ \}\ \texttt{forward}\ \{\ c\ \}}}{\{\ c\ \}\ \texttt{forward!}\ \{\ c \wedge \text{Fail}(\texttt{forward})\ \}} \qquad \{\ c\ \}\ \texttt{forward!}\ \{\ d\ \}}{\{\ c\ \}\ \texttt{DFS}\ \{\ c\ \}} \qquad \text{Break}(c, \texttt{DFS}, e)}{\{\ c\ \}\ \texttt{DFS!}\ \{\ (c \wedge \text{Fail}(\texttt{DFS})) \vee e\ \}}}{\{\ c\ \}\ \texttt{DFS!}\ \{\ e\ \}} \qquad \text{Subtree A1}}{\{\ pre\ \}\ \texttt{init;DFS!;Check}\ \{\ post\ \}}\ \text{[cons]}\ \frac{}{\{\ pre \wedge \text{Success}(\texttt{init})\ \}\ \texttt{init;DFS!;Check}\ \{\ post\ \}} \qquad \text{Subtree A2}$$

for Subtree A1:

$$\text{[try]}\ \frac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ \text{Slp}(d \wedge \text{Success}(\texttt{back}), \texttt{back})\ \}}\ \text{[cons]}\ \dfrac{}{\{\ d \wedge \text{Success}(\texttt{back})\ \}\ \texttt{back}\ \{\ c\ \}} \qquad \text{[cons]}\ \dfrac{\text{[break]}\ \dfrac{}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ d \wedge \text{Fail}(\texttt{back})\ \}}}{\{\ d \wedge \text{Fail}(\texttt{back})\ \}\ \texttt{break}\ \{\ c\ \}}}{\{\ d\ \}\ \texttt{try back else break}\ \{\ c\ \}}$$

and Subtree A2:

$$\text{[if]}\ \frac{\text{[cons]}\ \dfrac{\text{[fail]}\ \dfrac{}{\{\ e\ \}\ \texttt{fail}\ \{\ \text{false}\ \}}}{\{\ e \wedge \text{Success}(\texttt{match})\ \}\ \texttt{fail}\ \{\ \text{post}\ \}} \qquad \text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ \text{false}\ \}\ \texttt{skip}\ \{\ \text{false}\ \}}}{\{\ e \wedge \text{Fail}(\texttt{match})\ \}\ \texttt{skip}\ \{\ post\ \}}}{\{\ e\ \}\ \texttt{if match then fail}\ \{\ post\ \}}$$

FIGURE 8.11: Proof tree for partial correctness of `is-connected` (B)

TABLE 8.6: Conditions inside proof tree of `is-connected` (B)

MSO formulas
$pre \equiv$ $\forall_v x(m_v(x) = \text{none} \wedge \neg\text{root}(x)) \wedge \forall_e x(m_e(x) = \text{none})$ $\wedge\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg\text{adj}(x, y))$
$post \equiv \text{false}$
$c \equiv$ $\forall_e x(m_e(x) = \text{none} \vee m_e(x) = \text{dashed})$ $\wedge\exists_v x(\text{root}(x) \wedge m_v(x) = \text{red}$ $\wedge \forall_v y(x = y \vee ((m_v(y) = \text{none} \vee m_v(y) = \text{red} \vee m_v(y) = \text{grey}) \wedge \neg\text{root}(y)))$ $\wedge\forall_v x(m_v(x) = \text{grey} \Rightarrow \neg\exists_v y(\text{adj}(x, y) \wedge \neg\text{root}(y) \wedge (m_v(y) = \text{none} \vee m_v(y) = \text{red}\wedge)))$ $\wedge\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg\text{adj}(x, y)) \wedge \forall_v x(x \notin X \Rightarrow m_v(x) = \text{none})$
$d \equiv c\wedge$ Fail(`forward`)
$e \equiv$Success(`match`)
Success(`back`) $\equiv$ $\exists_v x, y(m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{red} \wedge \neg\text{root}(y) \wedge (\text{edge}(x, y, \text{dashed}) \vee \text{edge}(y, x, \text{dashed})))$
Fail(`back`) $\equiv \neg$Success(`back`)
Fail(`init`) $\equiv \neg\exists_v(m_v(x) = \text{none} \wedge \neg\text{root}(x))$
Fail(`match`) $\equiv$ Fail(`init`)
Success(`match`) $\equiv \neg$Fail(`init`)
Fail(`DFS`) $\equiv \text{false}$
Fail(`forward`) $\equiv$ $\neg\exists_v x, y(m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge (\text{edge}(x, y, \text{none}) \vee \text{edge}(y, x, \text{none})))$
Slp(pre,`init`) $\equiv$ $\exists_v a(m_v(a) = \text{red} \wedge \text{root}(a)$ $\wedge \forall_v x(x = a \vee (m_v(x) = \text{none} \wedge \neg\text{root}(x))) \wedge \forall_e x(m_e(x) = \text{none})$ $\wedge \exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg\text{adj}(x, y))))$
Slp(c,`forward`) $\equiv$ $\exists_v a, b(\exists_e c(a \neq b \wedge m_v(a) = red \wedge m_v(b) = red \wedge \neg root(a) \wedge root(b)$ $\wedge ((s(c) = a \wedge t(c) = b) \vee (s(c) = b \wedge t(c) = a)) \wedge m_e(c) = \text{dashed} \wedge f))$
$f \equiv$ $\forall_e x(x \neq c \Rightarrow m_e x = \text{none} \vee m_e x = \text{dashed})$ $\wedge\forall_v y(y \neq a \wedge y \neq b \Rightarrow \neg\text{root}(y) \wedge (m_v(y) = \text{none} \vee \neg\text{root}(y) = \text{red} \vee \neg\text{root}(y) = \text{grey}))$ $\wedge\forall_v x(x \neq a \wedge x \neq b \wedge m_v(x) = \text{grey}$ $\Rightarrow \neg\exists_v y(\text{adj}(x, y) \wedge \neg\text{root}(y) \wedge (m_v(y) = \text{none} \vee m_v(y) = \text{red})))$ $\exists_v X(a \in X \wedge b \in X \wedge \exists_v x(x \notin X) \wedge \forall_v y(y \notin X \Rightarrow \neg\text{adj}(a, y) \wedge \neg\text{adj}(b, y))$ $\wedge \forall_v x, y(x \neq a \wedge x \neq b \wedge x \in X \wedge y \notin X \Rightarrow \neg\text{adj}(x, y)) \wedge \forall_v x(x \notin X \Rightarrow m_v(x) = \text{none}))$
Slp($d \wedge$ Success($back$), `back`) $\equiv$ $\exists_v a, b(\exists_e c(a \neq b \wedge m_v(a) = grey \wedge m_v(b) = red \wedge \neg root(a) \wedge root(b)$ $\wedge ((s(c) = a \wedge t(c) = b) \vee (s(c) = b \wedge t(c) = a)) \wedge m_e(c) = \text{none} \wedge f$ $\wedge \neg\exists_v y(y \neq a \wedge y \neq b \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge \text{adj}(b, y, \text{none}))$ $\wedge \neg\exists_v x, y(x \neq a \wedge x \neq b \wedge y \neq a \wedge y \neq b$ $\wedge m_v(x) = \text{red} \wedge \text{root}(x) \wedge m_v(y) = \text{none} \wedge \neg\text{root}(y) \wedge \text{adj}(b, y, \text{none}))))$

3. Proof of Break(c,`DFS`,e)

   From the semantics of GP 2 commands, when we have the derivation $[\![DFS, G]\!] \rightarrow^* [\![\texttt{break}, H]\!]$ iff we have the derivation $[\![\texttt{forward!}, G]\!] \rightarrow^H$ for a graph H such that Fail(`back`) holds on H. From the proof tree of Figure 8.11, we know that $\{c\}$ `forward!` $\{c\wedge$Fail(`forward`)$\}$ is correct so that H must imply $c\wedge$Fail(`forward`) as well. Hence, H must satisfy $c\wedge$Fail(`forward`)$\wedge$Fail(`back`) if the input graph G satisfies c.

   $c\wedge$Fail(`forward`)$\wedge$Fail(`back`) clearly implies c. If the formula c holds, we know that the formula $\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg\text{adj}(x, y))$

$\wedge \forall_v x(x \notin X \Rightarrow m_v(x) = none)$
holds as well. Since the formula expresses the existence of an unmarked node (which must be unrooted if we consider $\exists_v x(m_v(x) = red \wedge root(x) \wedge \forall_v y(x = y \vee \neg root(x)))$ of c), Success(`match`) must be hold as well if c holds.

4. *Proof of* Slp(c,`forward`) *implies* c

Slp(c, `forward`) implies that if all edges that is not represented by c are unmarked or dashed, while c representing a dashed edge. Hence, Slp(c, `forward`) implies $\forall_e x(m_e x = none \vee m_e x = dashed)$. Similarly, Slp($c$, `forward`) implies all nodes that are not represented by a and b are unrooted and either unmarked, red, or grey, where a and b representing grey unrooted node and red rooted node respectively. Hence, Slp(c, `forward`) implies
$\exists_v x(m_v(x) = red \wedge root(x) \wedge \forall_v y(x = y \vee (\neg root(y) \wedge (m_v(y) = none \vee m_v(y) = red \vee m_v(y) = grey))))$.

Slp(c, `forward`) also implies the existence of a set of nodes X such that both a and b both belong to the set and there exists a node that is not in the set, such that no node outside X adjacent to a or b, or any other nodes in X. Also, all nodes outside X are unmarked. Note that the nodes represented by a and b are marked so that $a \in X \wedge b \in X \wedge \forall_v x(x \notin X \Rightarrow m_v(x) = none)$ implies $\forall_v x(x \notin X \Rightarrow m_v(x) = none)$. Hence, Slp($c$, `forward`) implies
$\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg adj(x, y)) \wedge \forall_v x(x \notin X \Rightarrow m_v(x) = none)$

Finally, Slp(c, `forward`) implies
$\forall_v x(x \neq a \wedge x \neq b \wedge m_v(x) = grey$
$\quad \Rightarrow \neg\exists_v y(adj(x, y) \wedge \neg root(y) \wedge (m_v(y) = none \vee (m_v(y) = red \wedge \neg root(y)))))$
Note that Slp(c, `forward`) also implies that both nodes represented by a and b are not grey. Hence, the implication still holds for the case where $x = a$ or $x = b$, so that Slp(c, `forward`) implies
$\forall_v x(m_v(x) = grey \Rightarrow \neg\exists_v y(adj(x, y) \wedge \neg root(y) \wedge (m_v(y) = none \vee (m_v(y) = red \wedge \neg root(y)))))$.

5. *Proof of* Slp($d \wedge$Success($back$),`back`) *implies* c
From the meaning of conjunction, we know that Slp($d \wedge$Success($back$),`back`) implies
$\exists_v a, b(\exists_e c(a \neq b \wedge m_v(a) = grey \wedge m_v(b) = red \wedge \neg root(a) \wedge root(b)$
$\quad \wedge ((s(c) = a \wedge t(c) = b) \vee (s(c) = b \wedge t(c) = a)) \wedge m_e(c) = none \wedge f))$.
Similarly as the previous point, f implies $\forall_e x(m_e x = none \vee m_e x = dashed)$ and also
$\exists_v x(m_v(x) = red \wedge root(x) \wedge \forall_v y(x = y \vee (\neg root(y) \wedge (m_v(y) = none \vee m_v(y) = red \vee m_v(y) = grey))))$
because node that is represented by a is unrooted and grey, node that is represented by b is red and rooted, while edge that is represented by c is unmarked. And with the same reason as the previous point, we can say that Slp($d \wedge$Success($back$),`back`) implies
$\exists_V X(\exists_v x(x \in X) \wedge \exists_v x(x \notin X) \wedge \forall_v x, y(x \in X \wedge y \notin X \Rightarrow \neg adj(x, y)) \wedge \forall_v x(x \notin X \Rightarrow m_v(x) = none)$.

Finally, similarly with the previous point, Slp($d \wedge$Success($back$),`back`) implies
$\forall_v x(x \neq a \wedge x \neq b \wedge m_v(x) = grey$
$\quad \Rightarrow \neg\exists_v y(adj(x, y) \wedge \neg root(y) \wedge (m_v(y) = none \vee (m_v(y) = red \wedge \neg root(y)))))$

when Slp(c, `forward`) also implies that node represented by b is not grey, while node that is represented by a is adjacent to a red rooted node. Hence, the implication still holds for the case where $x = a$ or $x = b$, so that Slp(d∧Success($back$),`back`) implies
$\forall_v x(\mathsf{m}_v(x) = \mathsf{grey} \Rightarrow \neg\exists_v y(\mathsf{adj}(x, y) \wedge \neg\mathsf{root}(y) \wedge (\mathsf{m}_v(y) = \mathsf{none} \vee (\mathsf{m}_v(y) = \mathsf{red} \wedge \neg\mathsf{root}(y)))))$.

8.5 Summary

In this chapter, we have shown four graph programs that are proven to be partially correct with respect to the given specifications. In the first case study, `vertex-colouring`, the specification can be expressed in first-order formulas, while specifications in the remaining case studies are not expressible by first-order formulas. The given specification for the second case study (`transitive-closure`) requires the expression of the existence of a path between two nodes. Hence, we use the predicate path, which is not a first-order predicate. The 2-colourability of a graph (or bipartiteness of a graph) and disconnectedness also cannot be expressed in first-order formulas since we need to express that the nodes can be divided into two sets with certain constraints.

Here, we only focus on partial correctness of the programs with respect to the given specifications. Hence, the proof trees in this chapter give us information in the case where the execution of the programs terminate and do not fail. However, by considering the semantics of the programs, we can argue that the program will not fail and must terminate.

For `vertex-colouring` and `transtitive-closure`, one can easily show that the programs are non-failing and terminating. For `2-colouring` and `is-connected`, we argue that the programs must terminate, and with the help of the proof trees, we show that the programs fail for one of the preconditions and do not fail for the other.

The proof for `2-colouring` and also for `disconnectedness` are long, relative to the other two programs. This is due to the existence of branching commands and nested loops. The formulas we use in the proof trees of `2-colouring` and `disconnectedness` are longer than the other two. This is due to the complexity of the loop body. In `vertex-colouring` and `transitive-closure`, the loop bodies only containing a rule set call. Hence, every invariant we need to use is an invariant over a rule set call. This is why we have simpler formulas when we proof the two programs.

As we can see in all case studies above, we need to prove the correctness of implications and the predicate Break we use in the proof trees. This is due to the calculus we use, which does not contain proof rules for proving implications between assertions, or the correctness of predicate Break. The proofs of implications we use here mostly use the meaning of

conjunctions, disjunctions, and implications. For the proof of a predicate Break, we need to use the semantics of graph programs while utilising triples we have in the proof trees.

From the four case studies we have in this chapter, we can say that our proof calculus is applicable to some graph programs. Especially, it is applicable to a graph program with depth-first search (DFS) approach, as in `connectedness`, which is not supported by the calculus introduced in [1].

Chapter 9

Soundness and completeness of the proof calculi

This chapter discusses the soundness and the relative completeness of proof calculi we defined in Chapter 7. We show proof that both proof calculi are sound, and that the semantic proof calculus is relatively complete.

9.1 Soundness

To proof the soundness, we use structural induction on proof tree as defined in Definition 9.1.

Definition 9.1 (Structural induction on proof trees)**.** Let us consider a property *Prop*. To prove that *Prop* holds for all proof trees (that are created from some proof rules) by *structural induction on proof tree* is done by:

1. Show that *Prop* holds for each axiom in the proof rules
2. For each inference rule, assuming that *Prop* holds for each premise T of the rule (i.e. upper part of the rule), show that *Prop* holds for the conclusion as well (i.e. lower part of the rule). □

When we prove that a triple $\{c\}P\{d\}$ for assertions c, d and a graph program P is partially correct by showing that $\mathrm{SLP}(c, P)$ implies d, it is obviously sound because of the definition of a strongest liberal postcondition itself. Then if c and d are monadic second-order formulas and P is a loop-free program, showing that $\mathrm{Slp}(c, P)$ implies d implies that $\{c\}P\{d\}$ is partially correct from Theorem 7.16. Then we also need to prove the soundness of proof calculus as summarised in Figure 7.1 and Figure 7.2.

Theorem 9.2 (Soundness of SEM)**.** Let us consider a graph program P and assertions c, d. Then,

$$\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\} \text{ implies } \vDash \{c\}\ P\ \{d\}.$$

Proof. To prove the soundness, we show that the implication holds for each axiom and inference rule in the proof rule w.r.t. the semantics of graph programs by structural induction on proof trees.

1. Base case :
   (a) $[\text{ruleapp}]_{\text{slp}}$. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs H, $H \vDash d$ iff $H \vDash \text{SLP}(c, r)$. Suppose that $G \vDash c$. From Definition 4.1, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash \{c\}\ P\ \{d\}$.
   (b) $[\text{ruleapp}]_{\text{wlp}}$. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs G, $G \vDash c$ iff $H \vDash \text{WLP}(r, c)$. Suppose that $G \vDash c$. From Definition 7.4, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash \{c\}\ P\ \{d\}$.
2. Inductive case.
   Assume that *Prop* holds for each premise of inference rules in Definition 7.12 for a set of rule schemata $\mathcal{R}$, assertions c, d, e, c', d', inv, host graphs G, G', H, H', and graph programs C, P, Q.
   (a) [ruleset]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \mathcal{R}\ \{d\}$ and $G \vDash c$. Since we can have a proof tree where $\{c\}\ \mathcal{R}\ \{d\}$ is the root, then $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From point 1, this means that $\vDash \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From the semantics of graph programs, $H \in [\![R]\!]G$ iff $H \in [\![r]\!]G$ for some $r \in \mathcal{R}$. Since for any $r \in \mathcal{R}$, $H \in [\![r]\!]G$ implies $H \vDash d$, $H \in [\![\mathcal{R}]\!]G$ implies $H \vDash d$ as well so that $\vDash \{c\}\ \mathcal{R}\ \{d\}$.
   (b) [comp]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P; Q\ \{d\}$ and $G \vDash c$. $\vdash_{\mathsf{SEM}} \{c\}\ P; Q\ \{d\}$, implies $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{e\}$ and $\vdash_{\mathsf{SEM}} \{e\}\ Q\ \{d\}$. From the semantic of graph programs, $H \in [\![P; Q]\!]G$ iff there exists G' such that $G' \in [\![P]\!]G$ and $H \in [\![Q]\!]G'$. In addition to the assumption, $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{e\}$ implies $G' \vDash e$, and $\vdash_{\mathsf{SEM}} \{e\}\ Q\ \{d\}$ implies $H \vDash d$ so that $\vDash \{c\}\ P; Q\ \{d\}$.
   (c) [cons]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$ and $G \vDash c$. From the inference rule, we know that $\vdash \{c'\}\ P\ \{d'\}$, c implies c' (so that $G \vDash c'$), and d' implies d. From $\vdash \{c'\}\ P\ \{d'\}$, we get that for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d'$ so that $H \vDash d$. Hence, $\vDash \{c\}\ P\ \{d\}$.
   (d) [if]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$. From $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$, we get $\vdash_{\mathsf{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ and $\vdash_{\mathsf{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From the former we know that for all host graphs H, if $G \vDash \text{SUCCESS}(C)$ and $H \in [\![P]\!]G$ then $H \vDash d$, while from the latter we know that for all host graphs H, if $G \vDash \text{FAIL}(C)$ and $H \in [\![Q]\!]G$ then $H \vDash d$. Recall that from the semantic of graph programs, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ iff $G \vDash \text{SUCCESS}(C) \wedge$

$H \in [\![P]\!]G$ or $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$. Since both $G \vDash \text{SUCCESS}(C) \wedge H \in [\![P]\!]G$ and $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$ implies $H \vDash d$, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ implies $H \vDash d$ such that $\vDash \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.

(e) [try]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$.
$\vdash_{\mathsf{SEM}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ implies $\vdash_{\mathsf{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ C; P\ \{d\}$ and $\vdash_{\mathsf{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From the former we know that for all host graphs H, if $G \vDash \text{SUCCESS}(C)$ and $H \in [\![C; P]\!]G$ then $H \vDash d$, while from the latter we know that for all host graphs H, if $G \vDash \text{FAIL}(C)$ and $H \in [\![Q]\!]G$ then $H \vDash d$. Recall that from the semantic of graph programs, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ iff $G \vDash \text{SUCCESS}(C) \wedge H \in [\![C; P]\!]G$ or $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$. Since both $G \vDash \text{SUCCESS}(C) \wedge H \in [\![C; P]\!]G$ and $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$ implies $H \vDash d$, $H \in [\![\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ implies $H \vDash d$ such that $\vDash \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.

(f) [alap]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P!\ \{d\}$ and $G \vDash c$. From $\vdash_{\mathsf{SEM}} \{c\}\ P!\ \{d\}$, we know that $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$ and $\text{Break}(c, P, d)$ holds. From $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$, we get that for all host graph H, $H \in [\![P]\!]G$ implies $H \vDash c$, while from Definition 7.10 and the true value of $\text{Break}(c, P, d)$ we know that for all hsot graphs H, $G \vDash c$ and $\langle P, G\rangle \to^* \langle \texttt{break}; \texttt{H}\rangle$ implies $H \vDash d$. From the semantic of graph programs, $H \in [\![P!]\!]G$ iff there exist derivation $\langle P, G\rangle \to^* \langle \texttt{break}; \texttt{H}\rangle$ or $\langle P!, G\rangle \to^* \langle P!, H\rangle$ and $\langle P!, H\rangle \to^+ \texttt{fail}$. The first case yields $H \vDash d$ because of $\text{Break}(c, P, d)$. Note that $\langle P!, G\rangle \to^* \langle P!, H\rangle$ is done by having (probably) multiple execution of P on host graphs, so that from $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$ we know that $H \vDash c$. Then since $\langle P!, H\rangle \to^+ \texttt{fail}$, $H \vDash \text{FAIL}(P)$ so that $H \vDash c \wedge \text{FAIL}(C)$. Hence, $H \in [\![P!]\!]G$ implies $H \vDash d \vee (c \wedge \text{FAIL}(P))$ so that $\vDash \{c\}\ P!\ \{d\}$.

□

Theorem 9.3 (Soundness of SYN)**.** Let P be a restricted graph program i.e. graph programs where for every subprogram in the form $\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, $\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, or $C!$, C is a loop-free program. Let also c and d be monadic second-order formulas. Then,

$$\vdash_{\mathsf{SYN}} \{c\}\ P\ \{d\} \text{ implies } \vDash \{c\}\ P\ \{d\}.$$

Proof. The soundness of $[\text{ruleapp}]_{\text{slp}}$ follows from Theorem 5.16 and Theorem 9.2, while the soundness of $[\text{ruleapp}]_{\text{wlp}}$ follows from Theorem 5.16 and Lemma 7.22. The soundness of [ruleset], [comp], [cons], [if], and [try] follows from Theorem 9.2 and Theorem 7.16 about defining SUCCESS and FAIL in monadic second-order formulas. Finally, the soundness of the inference rule [alap] follows from Theorem 9.2 and Theorem 7.20. □

9.2 Relative completeness

A proof calculus is complete when every valid triple can be proven to be correct by the proof calculus. However, completeness really depends on the kind of assertions we used because the ability to prove that d can be implied by c for some assertions c and d depends on language of the assertions.

Let us consider Gödel's first incompleteness theorem, which says that any consistent formal system F can be carried out is incomplete if it includes Peano arithmetic [58]. Since we can express all sentences of Peano arithmetic in (the first-order fragment) of our monadic second-order formulas, the incompleteness should follows. Because of Peano arithmetic, there can not be a complete set of inference rule for implication theorem so that instead of discussing about completeness, here we focus on relative completeness, where we can separate the incompleteness due to the axioms and inference rules from any incompleteness in deducing valid assertions [59].

In [1], relative completeness of the proof calculi are shown by showing that $\{WLP(P,d)\}\ P\ \{d\}$ can be proven by the calculi for any graph program P and assertion d. Here, we use the same approach to show that our semantic proof calculus is relative complete.

Theorem 9.4. Let us consider a proof calculus A. Then, for any assertions c, d, and graph program P, $\vDash \{c\}P\{d\}$ implies $\vdash_A \{c\}P\{d\}$ (i.e. A is relative complete) if the following hold for any graph programs S and assertions a, b, e:

(i) $\vdash_A \{WLP(S,e)\}\ S\ \{e\}$

(ii) if a implies b then $\vdash_A \{b\}\ S\ \{e\}$ implies $\vdash_A \{a\}\ S\ \{e\}$

Proof. From the definition of weakest liberal precondition (see Definition 7.4), we know that if $\vDash \{c\}\ P\ \{d\}$ then c implies $\mathrm{WLP}(P,d)$. If the premise of the theorem holds, from (i) we know that $\vdash_A \{WLP(P,d)\}\ P\ \{d\}$. Because c implies $\mathrm{WLP}(P,d)$ and $\vdash_A \{WLP(P,d)\}\ P\ \{d\}$, from (ii) we also know that $\vdash_A \{c\}\ P\ \{d\}$. □

Before showing that SEM is relative complete, because of Theorem 9.4, we first show that for any postcondition d and graph program P, we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d)\}\ P\ \{d\}$. Note that $\mathrm{WLP}(P,d)$ must exist because we always have at least one liberal precondition for any graph program, that is: true.

Lemma 9.5. Let us consider a graph program S and a postcondition d. Then,

$$\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S,d)\}\ S\ \{d\}$$

Proof. Here we prove the lemma by induction on graph programs.

Base case. If S is a (conditional) rule schema r,

$\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S,d)\}\ S\ \{d\}$ automatically follows from the axiom $[\mathrm{ruleapp}]_{\mathrm{wlp}}$.

Inductive case.

Assume that for graph programs C, P, and Q, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(C,d)\}\ C\ \{d\}$, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d)\}\ P\ \{d\}$, and $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q,d)\}\ Q\ \{d\}$.

(a) If $S = \mathcal{R}$.

If $\mathcal{R} = \{\}$, then there is no premise to prove so that we can deduce $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(\mathcal{R},d)\}\ \mathcal{R}\ \{d\}$ automatically. If $\mathcal{R} = \{r_1,\ldots,r_n\}$ for $n > 0$, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(r_1,d)\}\ r_1\ \{d\},\ldots,\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(r_n,d)\}\ r_n\ \{d\}$ from $[\mathrm{ruleapp}]_{\mathrm{slp}}$. Let e be the assertion $\mathrm{WLP}(r_1,d) \wedge \ldots \wedge \mathrm{WLP}(r_n,d)$, so that by [cons], $\vdash_{\mathsf{SEM}} \{e\}\ r_1\ \{d\},\ldots,\vdash_{\mathsf{SEM}} \{e\}\ r_n\ \{d\}$. By [ruleset] we then get that $\vdash_{\mathsf{SEM}} \{e\}\ \mathcal{R}\ \{d\}$. Then by [cons], $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(\mathcal{R},d)\}\ \mathcal{R}\ \{d\}$ because

$$
\begin{aligned}
G \vDash\ & \mathrm{WLP}(\mathcal{R},d)\\
\overset{\text{L7.6}}{\Leftrightarrow}\ & \forall H.H \in [\![\mathcal{R}]\!]G \Rightarrow H \vDash d\\
\Leftrightarrow\ & \forall H.(H \in [\![r_1]\!]G \vee \ldots \vee H \in [\![r_n]\!]G) \Rightarrow H \vDash d\\
\Leftrightarrow\ & \forall H.(H \in [\![r_1]\!]G \Rightarrow H \vDash d) \wedge \ldots \wedge (H \in [\![r_n]\!]G \Rightarrow H \vDash d)\\
\overset{\text{L7.6}}{\Leftrightarrow}\ & G \vDash \mathrm{WLP}(r_1,d) \wedge \ldots \wedge \mathrm{WLP}(r_n,d)
\end{aligned}
$$

(b) If $S = P;Q$,

From the assumption, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,\mathrm{WLP}(Q,d))\}\,P\,\{\mathrm{WLP}(Q,d)\}$ and $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q,d)\}\,Q\,\{d\}$. Then by the inference rule [comp], we get that $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,\mathrm{WLP}(Q,d))\}\ P;Q\ \{d\}$. Finally by [cons], we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P;Q,d)\}\ P;Q\ \{d\}$ because

$$
\begin{aligned}
G \vDash\ & \mathrm{WLP}(P;Q,d)\\
\overset{\text{L7.6}}{\Leftrightarrow}\ & \forall H.H \in [\![P;Q]\!]G \Rightarrow H \vDash d\\
\Leftrightarrow\ & \forall H,G'.(G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G' \Rightarrow H \vDash d\\
\Leftrightarrow\ & \forall G'.(G' \in [\![P]\!]G \Rightarrow (\forall H.H \in [\![Q]\!]G' \Rightarrow H \vDash d)\\
\overset{\text{L7.6}}{\Leftrightarrow}\ & \forall G'.(G' \in [\![P]\!]G \Rightarrow G' \vDash \mathrm{WLP}(Q,d)\\
\overset{\text{L7.6}}{\Leftrightarrow}\ & G \vDash \mathrm{WLP}(P,\mathrm{WLP}(Q,d))
\end{aligned}
$$

(c) If $S = \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q$,

Both $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d)\}\,P\,\{d\}$ and $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q,d)\}\,Q\,\{d\}$ follow from the assumption. By [cons], we have:

$\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q,d))\}\,P\,\{d\}$ and

$\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q,d) \wedge (\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(P,d))\}\, Q\, \{d\}$.

Let e denotes $(\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(P,d)) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q,d))$ so that by [cons], we have:

$\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{SUCCESS}(C)\}\, P\, \{d\}$ and $\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{FAIL}(C)\}\, Q\, \{d\}$.

By [if] we then get that $\vdash_{\mathsf{SEM}} \{e\}\, S\, \{d\}$, and finally by [cons] we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S,d)\}\, S\, \{d\}$ because

$$
\begin{array}{ll}
& G \vDash \mathrm{WLP}(\mathtt{if}\, C\, \mathtt{then}\, P\, \mathtt{else}\, Q, d) \\
\overset{\text{L7.6}}{\Leftrightarrow} & \forall H.\ H \in [\![\mathtt{if}\, C\, \mathtt{then}\, P\, \mathtt{else}\, Q]\!] G \Rightarrow H \vDash d \\
\Leftrightarrow & \forall H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!] G) \\
& \qquad\qquad \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G)) \Rightarrow H \vDash d \\
\Leftrightarrow & (\forall H.\ (G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!] G) \Rightarrow H \vDash d) \\
& \wedge (\forall H.\ (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G) \Rightarrow H \vDash d) \\
\Leftrightarrow & G \vDash \mathrm{SUCCESS}(C) \Rightarrow (\forall H.\ H \in [\![P]\!] G \Rightarrow H \vDash d) \\
& \wedge G \vDash \mathrm{FAIL}(C) \Rightarrow (\forall H.\ H \in [\![Q]\!] G \Rightarrow H \vDash d) \\
\overset{\text{L7.6}}{\Leftrightarrow} & G \vDash (\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(P,d) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q,d))
\end{array}
$$

(d) If $S = \mathtt{try}\, C\, \mathtt{then}\, P\, \mathtt{else}\, Q$,

Let e denotes $\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(C; P, d) \wedge \mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q,d)$. Similar to point (c), from the assumption we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q,d)\}\, Q\, \{d\}$, which imply $\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{FAIL}(C)\}\, Q\, \{d\}$. Also from the assumption, we have both $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(C, \mathrm{WLP}(P,d))\}\, P\, \{\mathrm{WLP}(P,d)\}$ and also $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d)\}\, P\, \{d\}$. By [comp] and [cons] as case $S = P; Q$, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(C; P, d)\}\, C; P\, \{d\}$. Then by [cons] as in $\mathtt{if} - \mathtt{then} - \mathtt{try}$ case, $\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{SUCCESS}(C)\}\, C; P\, \{d\}$ such that by the inference rule [try] we have $\vdash_{\mathsf{SEM}} \{e\}\, S\, \{d\}$. Finnaly by [cons], $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S,d)\}\, S\, \{d\}$ because

$$
\begin{array}{ll}
& G \vDash \mathrm{WLP}(\mathtt{try}\, C\, \mathtt{then}\, P\, \mathtt{else}\, Q, d) \\
\overset{\text{L7.6}}{\Leftrightarrow} & \forall H.\ H \in [\![\mathtt{try}\, C\, \mathtt{then}\, P\, \mathtt{else}\, Q]\!] G \Rightarrow H \vDash d \\
\Leftrightarrow & \forall H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![C; P]\!] G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G)) \\
& \qquad \Rightarrow H \vDash d \\
\Leftrightarrow & (\forall H.\ (G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![C; P]\!] G) \Rightarrow H \vDash d) \\
& \wedge (\forall H.\ (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!] G) \Rightarrow H \vDash d) \\
\Leftrightarrow & G \vDash \mathrm{SUCCESS}(C) \Rightarrow (\forall H.\ H \in [\![C; P]\!] G \Rightarrow H \vDash d) \\
& \wedge G \vDash \mathrm{FAIL}(C) \Rightarrow (\forall H.\ H \in [\![Q]\!] G \Rightarrow H \vDash d) \\
\overset{\text{L7.6}}{\Leftrightarrow} & G \vDash (\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(C; P, d) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q,d))
\end{array}
$$

(d) If $S = P!$,

From the assumption, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P, \mathrm{WLP}(P!, d))\}\, P\, \{\mathrm{WLP}(P!, d)\}$. By [cons] as in $P;Q$ case, we get $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P; P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$ such that by [cons] we know that $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$. Note that from Theorem 9.2, this implies $\vDash \{\mathrm{WLP}(P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$ such that for all host graphs $G_1, \ldots, G_n$, and H where $G_2 \in [\![P]\!]G_1, \ldots, G_n \in [\![P]\!]G_{n-1}$, and $\langle P, G_n\rangle \to^* \langle \mathtt{break}, H\rangle$, $G \vDash \mathrm{WLP}(P!.d)$ implies $G' \vDash \mathrm{WLP}(P!.d)$ and $H \vDash d$. Hence, $\mathrm{Break}(\mathrm{WLP}(P!.d), P, d)$ holds. Then by the inference rule [alap], we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{(\mathrm{WLP}(P!, d) \wedge \mathrm{FAIL}(P)) \vee d\}$ such that by [cons], $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{d\}$ because

$$\begin{aligned} H \vDash \mathrm{WLP}(P!, d) \wedge \mathrm{FAIL}(P) &\overset{\text{L7.6}}{\Leftrightarrow} \mathrm{fail} \in [\![P]\!]H \wedge \forall H'.\ H' \in [\![P!]\!]H \Rightarrow H' \vDash d \\ &\Rightarrow \quad H \in [\![P!]\!]H \wedge \forall H'.\ H' \in [\![P!]\!]H \Rightarrow H' \vDash d \\ &\Rightarrow \quad H \vDash d. \end{aligned}$$

□

Corollary 9.6. Let us consider a graph program P and assertions c, d. Then,

$$\vDash \{c\}\ P\ \{d\} \text{ implies } \vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}.$$

Proof. From Lemma 9.5, we know that for all $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P, d)\}\ P\ \{d\}$ and because we have the proof calculus [cons] in the calculus SEM, then the premise of Theorem 9.4 holds so that $\vDash \{c\}\ P\ \{d\}$ implies $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$. □

In Theorem 9.6, we show the relative completeness of our semantic partial correctness calculus. The proof depends on the existence of WLP.

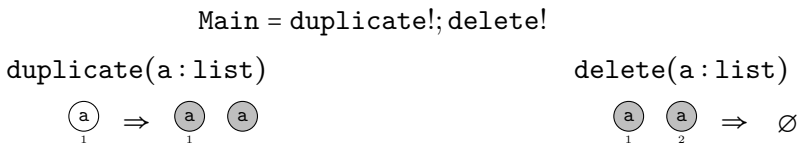


FIGURE 9.1: graph program `double`

If we consider first-order Hoare-triples, there is strong evidence that such situation can occur. For example, consider the triple $\{c\}\ P\ \{d\}$ with $c = \forall_{\mathsf{V}} \mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none} \wedge \neg\exists_{\mathsf{E}} \mathsf{y}(\mathsf{s}(\mathsf{y}) = \mathsf{x} \vee \mathsf{t}(\mathsf{y}) = \mathsf{x}))$ (all nodes are unmarked and isolated), $d = \forall_{\mathsf{V}} \mathsf{x}(\mathsf{false})$ (the graph is empty), and the program `double` of Figure 9.1.

It is obvious that $\vDash \{c\}$ `duplicate!;delete!` $\{d\}$ holds: `duplicate!` duplicates the number of nodes while marking the nodes grey, hence its result graph consists of an even number

of isolated grey nodes. Then `delete!` deletes pairs of grey nodes as long as possible, so the overall result is the empty graph. Note that "consists of an even number of isolated grey nodes" is both the strongest postcondition with respect to c and `duplicate!`, and the weakest precondition with respect to `delete!` and d.

Using SYN one can prove $\vdash \{c\}$ `duplicate!` $\{e\}$ where e expresses that all nodes are grey and isolated. However, we argue that our logic cannot express that a graph has an even number of nodes. This is because pure first-order logic (without built-in operations) cannot express this property [42] and it is likely that this inexpressiveness carries over to our logic. As a consequence, one can only prove $\vdash \{e\}$ `delete!` $\{f\}$ where f expresses that the graph contains at most one node (because otherwise `delete` would be applicable). But we cannot use SYN to prove $\vdash \{c\}$ `duplicate!;delete!` $\{d\}$.

From the example above we can see that the expressiveness of assertions play important role in relative completeness. Courcelle [45, 60] has proven that for a graph G and subsets X, Y of its vertex set V_G, the following properties are not expressible in monadic second-order logic without counting (either with set of node or set of edges quantifier):

1. The graph has even number of nodes
2. The number of nodes in a graph is a prime number
3. The graph has the same number of red nodes and grey nodes

However, we can express the three properties by the following formulas, respectively:

1. $\exists_V X(\forall_v x(x \in X) \wedge \exists_l n(\mathsf{card}(x) = 2 * n))$
2. $\exists_V X(\forall_v x(x \in X) \wedge \neg\exists_l n, m(n \neq 1 \wedge m \neq 1 \wedge \mathsf{card}(x) = n * m))$
3. $\exists_V X, Y(\forall_v x(m_v(x) = \mathsf{red} \Leftrightarrow x \in X) \wedge \forall_v x(m_v(x) = \mathsf{grey} \Leftrightarrow x \in Y) \wedge \mathsf{card}(X) = \mathsf{card}(Y))$

With the existence of function card, our formula can express more properties if we compare it with counting monadic second-order logic in [45]. However, what kind of properties can not be expressed by our formulas is still an open problem in this thesis. Hence, the relative completeness of our monadic second-order Hoare-triple is still unknown.

9.3 Summary

In this chapter, we have proven the soundness of our SEM and SYN calculi with respect to partial correctness. We prove the soundness of SEM by structural induction on a proof

tree, where the base cases are the proof rules [ruleapp]slp and [ruleapp]wlp. By using the soundness of strongest liberal postcondition we obtain from Chapter 5 and its connection with a weakest liberal precondition, we prove that the base cases also hold for SYN such that the calculus also sound.

Here, we also show the proof of the relative completeness of SEM. By considering relative completeness, we do not consider any incompleteness in deducing valid assertion (as used in the proof rule [cons]). In the proof, we can see the importance of having the proof rule $[\text{ruleapp}]_{\mathsf{wlp}}$. The proof rule is important because we need to have a triple about a loop body with its invariant as the specifications. In the proof, we show that a weakest liberal postcondition over a loop can be considered the loop body's invariance.

The proof of the soundness and correctness of SEM actually is not really different from what has been presented in [1]. However, here we have slightly different proof calculus due to the existence of the command `break`, which was not considered in the rule [alap] in [1] because the command `break` was added after the thesis was published.

On the other hand, the relative completeness of SYN remains an open problem in this thesis. However, we conjecture that the calculus is not relative complete due to our monadic second-order formula's expressiveness. We do not have concrete proof or counterexample to show this, but we have strong evidence that the first-order Hoare-triple is not relative complete due to the expressiveness of first-order formulas, as shown in Section 9.2 by using the graph program `double` of Figure 9.1 as example.

Chapter 10

Conclusions and future work

In this chapter, we summarise the findings in this thesis. Also, we propose some future work related to our findings.

10.1 Conclusions

We have defined monadic second-order formulas that can express the properties of GP 2 graphs. The syntax of the formulas is derived from the definition of GP 2 graphs and considering the syntax of GP 2 rule schema conditions. The formulas can express the structure of the graphs, labels, marks, rootedness, and conditions that may be expressed by rule schema conditions. Other than expressing properties of GP 2 graphs in general, here we also present a way to express properties of a graph with respect to its morphism with another graph. This is important in graph transformations since we are focus on graphs that have morphisms with the left-hand graph of rules.

By using the monadic second-order formulas as assertions, we are able to construct a strongest liberal postcondition with respect to a given precondition and a rule schema, which is proven to be correct. Moreover, it turns out that once we know a strongest liberal postcondition over a rule schema, we are able to obtain more assertions, that are: weakest liberal precondition with respect to a postcondition and a rule schema, assertions that express precondition for the successful execution of loop-free programs as well as failure execution of a subclass of commands, namely iteration commands.

From the obtained assertions, we present the extension of partial correctness Hoare-calculus, called syntactic proof calculus. With this calculus, we are able to verify graph programs that contain nested loops in certain forms. Other than syntactic proof calculus, in this thesis, we also present semantic proof calculus. While syntactic proof calculus requires closed monadic second-order formulas as assertions, semantic proof calculus is an extensional proof calculus

that considers assertions in a semantic way. In this thesis, both calculi are proven to be sound. Moreover, the semantic proof calculus is also proven to be relatively complete. For syntactic proof calculus, the relative completeness remains as an open problem.

The semantic proof calculus we present in this thesis can handle any graph programs, while the syntactic proof calculus has limitation to handle a subclass of programs, namely command programs. This is due to the ability to express the precondition for successful and fail execution of loops. In semantic proof calculus, we assume that assertions we use are able to express these properties. However, we do not have proof to show that these properties can be expressed in monadic second-order formulas.

In this thesis, we consider the verification of GP 2 graph programs. However, since GP 2 has many attributes (e.g. marks, expressions as labels, rooted nodes), the approaches we use here to construct a strongest liberal postcondition should be able to be used to construct a strongest liberal postcondition w.r.t. graph transformation rule (outside GP 2), as long as their attributes are covered by GP 2.

Although this thesis may contribute to advancing graph program verification, there are still many areas for improvement. Our proof calculi assume that the implications used in the proof rule [cons] can be done outside calculus. However, we have not presented any theory for theorem proving in this thesis. In addition, in this thesis, we only focus on partial correctness and not on total correctness.

10.2 Future work

We discuss several areas that can be studied based on the work of this thesis. Here we discuss problems that are still open in this thesis, and some topics that can be studied from the results presented in this thesis.

10.2.1 Theorem proving for implications between assertions

From the examples presented in Chapter 7, we showed that proving the implications that are used in the proof rule [cons] needs a lot of work so that it would be helpful to have formalism to prove the implications. Pennemann in his thesis [17] proposed a calculus that operates directly on nested conditions, which can specify first-order properties of plain graphs (not attributed as in GP 2 graphs). In the thesis, Pennemann investigated a sound and complete satisfiability algorithm that decides the tautology for a certain fragment of conditions. To tackle the termination issue in the algorithm, resolution principle [61] is used to present

deduction rules to handle conjunctive normal form. Another approach of theorem proving on nested conditions is also presented in [62], which also consider first-order properties of a graph. However, in this approach, an attributed graph is considered.

For first-order logic, satisfiability problem is known to be undecidable [17, 42]. In general, the satisfiability problem for monadic second-order logic is also undecidable [45]. However, in literature, there are studies that discuss satisfiability problem for some classes of monadic second-order logic [45–47]. Courcelle shows that if the property of interest can be expressed in monadic second-order logic with edge set quantifier, then parameterising by the combination of the tree-width of the graph of interest and the size of the formula, it can be determined in linear time whether the graph has the property [45, 48]. Moreover, Seese in [46] shows the converse, that if a set of finite, simple, undirected graphs has a decidable monadic second-order logic with edge set quantifier, then it has bounded tree-width. Also, Courcelle in [47] shows that if the class has a decidable counting monadic second-order (with additional predicate even) satisfiability problem, then it has bounded clique-width. Since some studies show the decidability of some subclasses of monadic second-order logic, we may be able to explore more about theorem proving when we consider monadic second-order logic than if we consider first-order logic.

10.2.2 Automatic construction of invariants

In the case studies we have in this thesis, an invariant over a single rule set call is obtained by using a similar method. We use a disjunction such that a strongest liberal postcondition over the invariant still implies the invariant. A formal construction technique of invariant for a rule set call, or maybe even a larger class of commands would to be an interesting topic to be investigated, because if we can construct an invariant automatically, we may be able to avoid finding the invariants manually to avoid human errors.

Verification of invariants for attributed graph transformations systems (based on nested attributed graph condition) has been discussed in [63]. We may use the approach to validate the construction. We may also investigate the verification approach to help us having the construction of invariants.

10.2.3 Monadic second-order transductions for reasoning about graph programs

The downfall of Hoare triple we use in this thesis is that we can not express the morphism between the initial and the final graph [64]. On the other hand, monadic second-order

transduction was discussed in [45, 50]. The transductions allow us to express the preservation of nodes and edges after a transformation. By expressing preservation, we may be able to express the existence of morphism $G \leftarrow D \rightarrow H$ in direct derivation $G \Rightarrow H$.

Monadic second-order transduction [45, 50] is a way to express the properties of the resulting graph based on the input graph. Transduction is defined with a so-called definition scheme. Intuitively, a definition scheme is a tuple of MSO formulas which then used in MSO transductions to define a graph G' from a graph G. In [50], graphs are defined as a tuple of domain and predicates while for host graphs, we have a tuple of domains and functions. To deal with the functions, we can view the functions as cases so that we can consider them as predicates.

After defining a definition scheme in our setting, we then can try to define monadic second-order transductions with the scheme. Intuitively, monadic second-order transduction with definition scheme $\mathcal{D}$ is an isomorphism class of relation between graphs such that G and H are related if H can be defined from G by $\mathcal{D}$.

10.2.4 Relative completeness for monadic second-order Hoare-triples

In this thesis, we have a conjecture that first-order Hoare-triples is not relative complete. To support the conjecture, we give an example of a graph program with the first-order specification. In order to prove the correctness, we need to express the evenness of nodes in the graph, which can not be expressed in first-order formulas.

The existence of even number of nodes cannot also be expressed in a monadic second-order formula without counting [42, 45]. Hence, triples whose specifications are monadic second-order formulas without counting should not be relative complete as well.

Monadic second-order formulas we defined in this thesis are expressive enough to express nodes' evenness, so the example can not work for the formulas. In fact, what kind of graph properties can not be expressed by our formulas is still an open problem in this thesis. Hence, we can not yet give a counter-example, nor proof related to monadic second-order Hoare triples.

10.2.5 Proof calculus for total correctness of monadic second-order Hoare triples

In this thesis, we only consider partial correctness so that proof of termination and non-failing must be done outside the calculus. As can be seen in Chapter 7, we show termination by

giving an argument based on the semantics of graph programs. We also use graph programs' semantics to argue that a program with certain precondition can not file, by also utilising proof-tree.

Although we can prove the total correctness of a program with our partial correctness calculus by giving additional argument on the semantics of graph programs, having the termination and non-failing proof embedded in the proof calculus would give us more advantage. Poskitt in [1] defines proof calculi for weak-total correctness and total correctness of Hoare triples with E-conditions as assertions. We can investigate how our formulas and the command `break` affect the calculus. However, the proof calculi in [1] can not handle nested loop. Having total correctness proof calculus that can handle at least the same class of programs as we have for partial correctness calculus would be interesting.

10.2.6 Proof obligation for the construction of a strongest liberal postcondition

Chapter 4 of this thesis concludes that we can construct a strong liberal postcondition over a rule schema and a precondition (in first- or monadic second-order formulas) if we can define some transformations (i.e. Lift, Shift, and Post) such that some properties hold. Since we use a similar approach as [1] to construct a strongest liberal postcondition, that is by lifting and shifting a condition; then we may be able to show that the properties we introduce in Chapter 4 of this thesis can be applied to other logic (e.g. E-condition).

References

[1] Christopher M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013. URL http://etheses.whiterose.ac.uk/4700/.

[2] Detlef Plump. The design of GP 2. In *Proc. 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.

[3] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer-Verlag, 2006.

[4] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation*. Monographs in Theoretical Computer Science. Springer-Verlag, 2015.

[5] Horst Bunke. Attributed programmed graph grammars and their application to schematic diagram interpretation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(6):574–582, 1982. doi: 10.1109/TPAMI.1982.4767310.

[6] A. Schürr, A. J. Winter, and A. Zündorf. *The PROGRES Approach: Language and Environment*, page 487–550. World Scientific Publishing Co., Inc., USA, 1999. ISBN 9810240201.

[7] C. Ermel, M. Rudolf, and G. Taentzer. *The AGG Approach: Language and Environment*, page 551–603. World Scientific Publishing Co., Inc., USA, 1999. ISBN 9810240201.

[8] 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, *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE 2011, Budapest, Hungary, October 4-7, 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.

[9] Amir Hossein Ghamarian, Maarten de Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *Int. J. Softw. Tools Technol. Transf.*, 14(1):15–40, 2012. doi: 10.1007/s10009-011-0186-x.

[10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. Grgen.net - the expressive, convenient and fast graph rewrite system. *Int. J. Softw. Tools Technol. Transf.*, 12(3-4): 263–271, 2010. doi: 10.1007/s10009-010-0148-8.

[11] Christopher Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015. URL http://etheses.whiterose.ac.uk/12586/.

[12] 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, 2001.

[13] Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2010. doi: 10.1007/978-3-642-15928-2_10.

[14] Christopher M. Poskitt and Detlef Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012. doi: 10.3233/FI-2012-708.

[15] Christopher M. Poskitt and Detlef Plump. Verifying total correctness of graph programs. In *Graph Computation Models (GCM 2012), Revised Selected Papers*, volume 61 of *Electronic Communications of the EASST*, 2013.

[16] 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, 2014. doi: 10.1007/978-3-319-09108-2_3.

[17] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2009.

[18] Graham Campbell, Brian Courtehoute, and Detlef Plump. Linear-Time Graph Algorithms in GP 2. In Markus Roggenbach and Ana Sokolova, editors, *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, volume 139 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:23, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-120-7. doi: 10.4230/LIPIcs.CALCO.2019.16. URL http://drops.dagstuhl.de/opus/volltexte/2019/11444.

[19] Brian Courtehoute and Detlef Plump. A fast graph program for computing minimum spanning trees. In *Proceedings of the Eleventh International Workshop on Graph Computation Models*, volume 330 of *Electronic Proceedings in Theoritical Computer Science*, pages 163–180, 2020. doi: 10.4204/EPTCS.330.

[20] Detlef Plump. From imperative to rule-based graph programs. *J. Log. Algebraic Methods Program.*, 88:154–173, 2017. doi: 10.1016/j.jlamp.2016.12.001.

[21] Tobias Nipkow. *Hoare Logics in Isabelle/HOL*, pages 341–367. Springer Netherlands, Dordrecht, 2002. ISBN 978-94-010-0413-8. doi: 10.1007/978-94-010-0413-8_11.

[22] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. doi: 10.1007/978-3-319-10542-0.

[23] Christine Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682, pages 45–95. Springer, 2012.

[24] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In *SETSS 2018*, volume 11430 of *LNCS*, pages 148–201. Springer, 2018. doi: 10.1007/978-3-030-17601-3_4.

[25] Gia Wulandari and Detlef Plump. Verifying graph programs with first-order logic. In *Proceedings of the Eleventh International Workshop on Graph Computation Models*, volume 330 of *Electronic Proceedings in Theoritical Computer Science*, pages 181–200, 2020. doi: 10.4204/EPTCS.330.

[26] Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:245–296, 2009.

[27] Francesc Rosselló and Gabriel Valiente. Graph transformation in molecular biology. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2005. doi: 10.1007/978-3-540-31847-7_7.

[28] Jakob Lykke Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. Chemical graph transformation with stereo-information. In *Graph Transformation - 10th International Conference, ICGT 2017 Proceedings*, volume 10373 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2017. doi: 10.1007/978-3-319-61470-0_4.

[29] Graham Campbell. Efficient graph rewriting. *CoRR*, abs/1906.05170, 2019. URL http://arxiv.org/abs/1906.05170.

[30] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In *Proceedings of the International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1979. ISBN 3-540-09525-X. URL http://dl.acm.org/citation.cfm?id=647558.730053.

[31] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. doi: 10.17/S0960129501003425.

[32] Annegret Habel and Detlef Plump. A core language for graph transformation (extended abstract), 2002.

[33] Sandra Steinert. *The Graph Programming Language GP*. PhD thesis, The University of York, 2007.

[34] Detlef Plump. The graph programming language GP. In Symeon Bozapalidis and George Rahonis, editors, *Algebraic Informatics, Third International Conference, CAI 2009, Thessaloniki, Greece, May 19-22, 2009, Proceedings*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009. doi: 10.1007/978-3-642-03564-7.

[35] 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, 2004.

[36] Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.

[37] 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. doi: 10.1007/3-540-45832-8.

[38] Ivaylo Hristakiev and Detlef Plump. Attributed graph transformation via rule schemata: Church-rosser theorem. In *Software Technologies: Applications and Foundations - STAF 2016 Collocated Workshops: DataMod, GCM, HOFM, MELO, SEMS, VeryComp, Vienna, Austria, July 4-8, 2016, Revised Selected Papers*, pages 145–160, 2016. doi: 10.1007/978-3-319-50230-4.

[39] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, third edition, 2009.

[40] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Fundamental Approaches to Software Engineering, Third Internationsl Conference, FASE 2000 Proceedings*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000. doi: 10.1007/3-540-46428-X_20.

[41] David von Oheimb. Hoare logic for java in isabelle/hol. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001. doi: 10.1002/cpe.598.

[42] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 3-540-21202-7. doi: 10.1007/978-3-662-07003-1.

[43] Zhongwan Lu. *Mathematical Logic for Computer Science*, volume 13 of *World Scientific Series in Computer Science*. World Scientific, 1989. ISBN 978-9971-50-251-5. doi: 10.1142/0388.

[44] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving, Second Edition*. Dover Publications, Inc., New York, NY, USA, 2015. ISBN 0486780821, 9780486780825.

[45] Bruno Courcelle. Monadic second-order graph transductions. In Jean-Claude Raoult, editor, *CAAP '92, 17th Colloquium on Trees in Algebra and Programming, Rennes, France, February 26-28, 1992, Proceedings*, volume 581 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1992. doi: 10.1007/3-540-55251-0.

[46] Detlef Seese. The structure of models of decidable monadic theories of graphs. *Ann. Pure Appl. Log.*, 53(2):169–195, 1991. doi: 10.1016/0168-0072(91)90054-P.

[47] Bruno Courcelle and Sang-il Oum. Vertex-minors, monadic second-order logic, and a conjecture by Seese. *J. Comb. Theory, Ser. B*, 97(1):91–126, 2007. doi: 10.1016/j.jctb.2006.04.003.

[48] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013. ISBN 978-1-4471-5558-4. doi: 10.1007/978-1-4471-5559-1.

[49] Christopher M. Poskitt and Detlef Plump. Hoare logic for graph programs. In *Proc. THEORY Workshop at Verified Software: Theories, Tools and Experiments (VS-THEORY 2010)*, 2010.

[50] Bruno Courcelle and Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2012. ISBN 0521898331, 9780521898331.

[51] Kenneth H. Rosen. *Discrete mathematics and its applications*. McGraw-Hill, seventh edition, 2012. ISBN 978-0-07-338309-5.

[52] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007. doi: 10.1007/978-3-540-74113-8.

[53] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990. ISBN 978-3-540-96957-0. doi: 10.1007/978-1-4612-3228-5.

[54] Patrick Cousot. Chapter 15 - methods and logics for proving programs. In Jan Van Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 841 – 993. Elsevier, Amsterdam, 1990. ISBN 978-0-444-88074-1. doi: https:

//doi.org/10.1016/B978-0-444-88074-1.50020-2. URL http://www.sciencedirect.com/science/article/pii/B9780444880741500202.

[55] Clifford B. Jones, A.W. Roscoe, and Kenneth R. Wood, editors. *Reflections on the Work of C.A.R. Hoare*. Springer, 2010. doi: 10.1007/978-1-84882-912-1.

[56] Krzysztof R. Apt and Ernst-Ruediger Olderog. Fifty years of hoare's logic, 2019.

[57] Brian Courtehoute Graham Campbell and Detlef Plump. Fast rule-based graph programs). *ArXiv e-prints*, arXiv:2012.11394 [cs.PL], 2020. URL https://arxiv.org/abs/2012.11394.

[58] Panu Raatikainen. Gödel's Incompleteness Theorems. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2020 edition, 2020.

[59] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978. doi: 10.1137/0207005.

[60] Bruno Courcelle. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. doi: 10.1016/0890-5401(90)90043-H.

[61] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. doi: 10.1145/321250.321253.

[62] Sven Schneider, Leen Lambers, and Fernando Orejas. Automated reasoning for attributed graph properties. *Int. J. Softw. Tools Technol. Transf.*, 20(6):705–737, 2018. doi: 10.1007/s10009-018-0496-3.

[63] Sven Schneider, Johannes Dyck, and Holger Giese. Formal verification of invariants for attributed graph transformation systems based on nested attributed graph conditions. In Fabio Gadducci and Timo Kehrer, editors, *Graph Transformation - 13th International Conference, ICGT 2020, Held as Part of STAF 2020, Bergen, Norway, June 25-26, 2020, Proceedings*, volume 12150 of *Lecture Notes in Computer Science*, pages 257–275. Springer, 2020. doi: 10.1007/978-3-030-51372-6.

[64] Gia S. Wulandari and Detlef Plump. Verifying a copying garbage collector in GP 2. In *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, pages 479–494, 2018. doi: 10.1007/978-3-030-04771-9.

Index