Verifying Graph Programs with Monadic Second-Order Logic (extended version)

Gia S. Wulandari*[1,2] and Detlef Plump[1]

[1] Department of Computer Science, University of York, UK
[2] School of Computing, Telkom University, Indonesia

Abstract. In this paper, 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 the so-called E-conditions as assertions. However, this type of assertion only able to express first-order properties of GP 2 graphs. An attempt to extend the assertion to monadic second-order logic, called M-condition, has been studied, but it is not a full MSO and does not consider attributed graph. Here, we present an approach to verify GP 2 graph programs with a standard monadic second-order logic (with counting). We show how to construct a strongest liberal postcondition with respect to a rule schema. We then show that we can prove the total correctness of graph programs in some classes, which include programs with nested loops.

1 Introduction

GP 2 is a graph programming language based on graph transformations that facilitates formal reasoning. Graphs in GP 2 are attributed: the nodes and edges can be labelled and marked, nodes may be rooted, and we can use expressions for labels of graphs in the rules so that it is practical to use [1].

Poskitt and Plump [19, 21] studied about verification of graph programs (programs that are written in GP 2) by introducing the so-called E-conditions as assertions for Hoare-style verification. E-conditions is an extension of nested conditions [8, 15] with support of expressions. By using E-conditions for assertions, we can verify a class of graph programs whose loop bodies and conditions of branching commands are rule-set calls. However, E-conditions are limited to express first-order properties of GP 2 graphs so that it can not express some important properties such as connectedness and the existence of a path. An extension of E-conditions, called M-conditions, is then introduced in [22]. M-conditions extend E-conditions by adding the path predicate to express the existence of a path. However, it only can express graphs without attributes (not GP 2 graphs).

In this paper, we introduce monadic second-order formulas to express GP 2 graphs. We define the formulas based on standard logic and GP 2 graphs and

* Supported by Indonesia Endowment Fund for Education (LPDP)

rule schema properties. We prefer to use standard logic because we think it is easier to comprehend by programmers that are not familiar with morphisms. In addition, there are some proof assistants such as Isabelle, Coq, and Z3 [2, 13, 14], that can be used for standard logic so that it might be useful for future work.

We have described in [24] how we can prove that a graph program is partially correct if we have a closed first-order formulas as specifications. In the paper, graph programs we can verify with first-order specifications are limited a class of program called control programs. Control programs allow a loop-free program to be conditions of branching commands, and allow nested loop in certain forms. Hence, we clearly have a larger class of programs that can be handle, if we compare to the work in [19].

Here, we show that we can prove that a control program is totally correct (in the sense that it must be terminate) if we have a closed monadic second-order formulas as specifications. To be able to prove the total correctness of a control program, we first show how we can construct a strongest liberal postcondition from the given precondition and rule schema. This construction then can be used to provide axioms in our proof calculus. We also use the same approach in [24] so that in this paper we show the extension we need to construct a strongest liberal postcondition over monadic second-order formulas.

The remainder of this paper is constructed as follows. A brief review of the graph programming language GP 2 can be found in Section 2. In Section 3, we introduce monadic second-order formulas that can express properties of GP 2 graphs. In Section 4, we outline the construction of a strongest liberal postcondition for a given rule schema and monadic second-order formula. Section 5 presents the total correctness proof rules of a semantic and a syntactic verification calculus, and identifies the class of programs that can be verified with the syntactic calculus. In Section 7, we demonstrate how to verify a graph program for computing transitive closure, also for checking connectedness of a graph. In Section 6, we discuss the soundness and completeness of our proof calculi. Section 8 contains a comparison of our approach with other approaches in the literature. Finally, we conclude and give some topics for future work in Section 9.

2 Graph programming language GP 2

GP 2 is a graph programming language using graph transformation systems with the double-pushout approach, which was introduced in [16]. In this section, we briefly introduce graph transformation systems in GP 2. For more detail documentation of GP 2, we refer readers to [1].

2.1 GP 2 graphs

A graph is a flexible structure in representing objects and relations between them. Objects are usually represented by nodes, while edges represent relations between them. Additional information about the objects and the relations are usually written as a label of the nodes and edges. Also, sometimes rooted nodes are used to distinguish some nodes with others.

Definition 1 (Label Alphabet). A label alphabet $\mathcal{C} = \langle \mathcal{C}_V, \mathcal{C}_E \rangle$ *is a pair comprising a set* $\mathcal{C}_V$ *of node labels and a set* $\mathcal{C}_E$ *of edge labels.* □

Definition 2 (Graph over label alphabet; class of graphs). *A graph over label alphabet* $\mathcal{C}$ is a system $G = \langle V_G, E_G, s_G, t_G, l_G, m_G, p_G \rangle$ 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 $l_G : V_G \to \mathcal{C}_V$, an edge labelling function $m_G : E_G \to \mathcal{C}_E$, and a partial rootedness function $p_G : V_G \to \{0, 1\}$. A *totally labelled graph* is a graph where its node labelling and rootedness functions are total. We then denote by $\mathcal{G}(\mathcal{C}_\perp)$ the set of all graphs over $\mathcal{C}$, and $\mathcal{G}(\mathcal{C})$ the set of all totally labelled graphs over $\mathcal{C}$. □

Graphically, in this paper, we represent a node with a circle, an edge with an arrow where its tail and head represent the source and target, respectively. The label of a node is written inside the node, while the label of an edge is written next to the arrow. The rootedness of a node v is represented by the line of the circle representing v, that is, standard circle for an unrooted node ($p(v) = 0$) and bold circle for a rooted node ($p(v) = 1$). To represent a node v with undefined rootedness ($p(v) =\perp$), we also use a standard circle. We use the same representation because nodes with undefined rootedness only exist in the interface of GP 2 rules, and the interface contains only this kind of nodes so that no ambiguity will arise even when we use the same representation.

There are two kinds of graphs in GP 2, that are 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.

Definition 3 (GP 2 labels). A set of node marks, denoted by $\mathbb{M}_V$, is the set {`none`, `red`, `blue`, `green`, `grey`}, while a set of edge marks, denoted by $\mathbb{M}_E$, is the set {`none`, `red`, `blue`, `green`, `dashed`}. A set of lists, denoted by $\mathbb{L}$, consists of all (list of) integers and strings that can be derived from the following abstract syntax:

```
𝕃         ::= empty | GraphExp | 𝕃 ‘:’ 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 $\{0, \ldots, 9\}$.

A GP 2 node label is a pair $\langle \ell^V, m^V \rangle \in \mathbb{L} \times \mathbb{M}_V$, and a *GP 2 edge label* is a pair $\langle \ell^E, m^E \rangle \in \mathbb{L} \times \mathbb{M}_E$. We then denote the set of GP 2 labels as $\mathcal{L} = \langle \mathcal{L}_V, \mathcal{L}_E \rangle$. □

The colon operator ‘:’ is used to concatenate atomic 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:

$$\begin{tikzcd}
 & & \texttt{int} & \\
\texttt{list} \arrow[r, phantom, "\supseteq"] & \texttt{atom} \arrow[ur, phantom, "\supseteq" sloped] \arrow[dr, phantom, "\supseteq" sloped] & & \\
 & & \texttt{string} \arrow[r, phantom, "\supseteq"] & \texttt{char}
\end{tikzcd}$$

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 4 (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:

```
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 $\mathcal{S}$, contains all pairs node label $\langle \ell^V, m^V \rangle \in \mathbb{E} \times (\mathbb{M}_V \cup \{\texttt{any}\})$ and edge label $\langle \ell^E, m^E \rangle \in \mathbb{E} \times (\mathbb{M}_E \cup \{\texttt{any}\})$ □

Definition 5 (GP 2 host graphs and rule graphs). A *host graph* G is a graph over $\mathcal{L}$, and a *rule graph* H is a graph over $\mathcal{S}$. A host graph (or rule graph) G has a node labelling function $l_G^V = \langle \ell_G^V, m_G^V \rangle$ such that for every node $v \in V_G$, $\ell_G^V(v)$ is defined if and only if $m_G^E(v)$ is defined. Similarly, for every edge $e \in E_G$, $\ell_G^E(e)$ is defined iff $m_G^E(e)$ is defined. □

If we consider the grammars of Definition 1 and Definition 4, it is obvious that $\mathbb{L}$ 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:

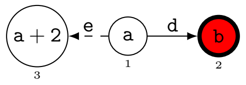
Graph ::= [Position] '|' Nodes '|' Edges
Nodes ::= '(' NodeId ['(R)'] ',' Label [',' Position] ')'
Edges ::= '(' EdgeId ['(B)']',' NodeId ',' NodeId ',' Label ')'

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 1 and Definition 4. 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.

The marks `red`, `green`, `blue` and `grey` are graphically represented by the obvious colours while `dashed` is represented by a dashed line. The wildcard mark `any` is represented by the colour magenta.

Node labels are undefined only in the interface graphs of rule schemata. This allows rules to relabel nodes. Similarly, the root function is undefined only for the nodes of interface graphs. The purpose of root nodes is to speed up the matching of rule schemata [1, 18].

Example 1 (A graph). Let G be a graph with $V_G = \{1,2,3\}, E_G = \{e1, e2\}, s_G = \{e1 \mapsto 1, e2 \mapsto 1\}, t_G = \{e1 \mapsto 2, e2 \mapsto 3\}, l_G = \{1 \mapsto \langle a, \texttt{none}\rangle, 2 \mapsto \langle b, \texttt{red}\rangle, 3 \mapsto \langle a+2, \texttt{none}\rangle\}, m_G = \{e1 \mapsto \langle d, \texttt{none}\rangle, e2 \mapsto \langle e, \texttt{dashed}\rangle\}$, and $p_G = \{1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 0\}$. Graphically, G can be seen as the following graph:



Syntactically in GP 2, G is written as follows:

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

To show a relation between graphs, which are what we do in graph transformations, we use graph morphism. 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.

Definition 6 (Graph morphisms). Given two graphs G and H. 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: $g_V \circ s_G = s_H \circ g_E$, $g_V \circ t_G = t_H \circ g_E$, $l_H(g_V(x)) = l_G(x)$, $m_H(g_E(y)) = m_G(y)$ for all $x \in V_G$ such that $l_G(x) \neq\perp$ and all $y \in E_G$ such that $m_G(y) \neq\perp$. Also, for all $v \in V_G$, such that $p_G(v) \neq\perp$ $p_H(g_V(v)) = p_G(v)$. 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 $l_H(g_V(v)) = \perp$ for all nodes v with $l_G(v) = \perp$ and $v \in r_G$ iff $g(v) \in r_H$ for all $v \in V_G$. Furthermore. we call a morphism g as an *inclusion* if $g(x) = x$ for all x in G. □

Definition 7 (Premorphisms). Given a rule graph L and a host graph 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 preserve sources, targets, and rootedness. □

2.2 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. In addition, GP 2 also allows a condition for the left-hand graph. When a condition exists, the rule is called a conditional rule schema.

Definition 8 (Rule schemata). A *rule schema* $r = \langle L \leftarrow K \to R \rangle$ comprises totally labelled rule graphs L and R, a graph K containing only unlabelled nodes with undefined rootedness, also 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 1. 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 paper, 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 variables occur in a rule schema condition must also occur in the left-hand graph of the rule schema.

Definition 9 (Conditional rule schemata). 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. $\square$

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.

A conditional rule schema $\langle L \leftarrow K \rightarrow R, \Gamma\rangle$ is applied to a host graph G in stages: (1) evaluate the expressions in L and R with respect to a premorphism $g\colon L \rightarrow G$ and a label assignment α, obtaining an instantiated rule $\langle L^{g,\alpha} \leftarrow K \rightarrow R^{g,\alpha}\rangle$; (2) check that $g\colon L^{g,\alpha} \rightarrow G$ is label preserving and that the evaluation of Γ with respect to g and α returns true; (3) construct two natural pushouts based on the instantiated rule and g.

Definition 10 (Label assignment). Consider a rule graph L and the set X of all variables occurring in L. For each $x \in X$, let $\mathrm{dom}(x)$ denotes the domain of x associated with the type of x. A *label assignment* for L is a triple $\alpha = \langle \alpha_{\mathbb{L}}, \mu_V, \mu_E\rangle$ where $\alpha_{\mathbb{L}}\colon X \rightarrow \mathbb{L}$ is a function such that for each $x \in X$, $\alpha_{\mathbb{L}}(x) \in \mathrm{dom}(x)$, and $\mu_V\colon V_L \rightarrow \mathbb{M}_V\backslash\{\texttt{none}\}$ and $\mu_E\colon E_L \rightarrow \mathbb{M}_E\backslash\{\texttt{none}\}$ are partial functions assigning a mark to each node and edge marked with `any`. $\square$

For a conditional rule schema $\langle L \leftarrow K \rightarrow R,\ \Gamma\rangle$ with the set X of all list variables in L, set Y (or Z) of all nodes (or edges) in L 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 $\mu_V(i)$, and every $m_L^E(i)$ for $i \in Z$ with $\mu_E(i)$. Then for an injective graph morphism $g : L^\alpha \rightarrow G$ 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 satisfaction of $\Gamma^{g,\alpha}$ in G is required for the application of a conditional rule schema. In addition, the application also depends on the dangling condition, which is a condition that asserts the production of a graph after node removal.

Definition 11 (Dangling condition; match). Let $r = L \leftarrow K \rightarrow R$ be a rule schema with host graphs L and R. Let also G be a host graph, and $g : L \rightarrow G$ be an injective morphism. The *dangling condition* is a condition where 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. $\square$

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 [9] for unrooted graph programming. The approach is the modified for rooted programming in [1, 3].

Definition 12 (Direct derivation; comatch). A *direct derivation* from a host graph G to a host graph H via a rule $r = \langle L \leftarrow K \rightarrow R\rangle$ consists of a natural double-pushout as in Fig. 1, where $g : L \to G$ and $g^* : R \to H$ are injective morphisms. If there exists such direct derivation, we write $G \Rightarrow_{r,g} H$, and we say that g^* is a *comatch* for r. $\Box$

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "(1)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d, "g^*"] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Fig. 1: A direct derivation for a rule $r = \langle L \leftarrow K \to R \rangle$

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[9, 11]. In [1], 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 Fig. 2 [3]. In addition, we need a natural double-pushout because we want to have invertible direct derivations.

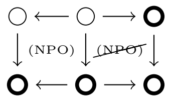


Fig. 2: Non-natural double-pushout

The natural double-pushout construction such that we have natural double-pushout is described in [1, 3], that are:

1. To obtain D, remove all nodes and edges in $g(L-K)$ from G. For all $v \in V_K$ with $l_K(v) =\perp$, define $l_D(g_V(v)) =\perp$. 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 $l_K(v) =\perp$, define $l_H(g_V(v)) = l_R(v)$. 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 13 (Conditional rule schema application). Given 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 $g : L \rightarrow G$ and a label assignment α_L such that:
(i) $g : L^{\alpha} \rightarrow G$ 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 9 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.

2.3 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 the abstract syntax of GP 2 programs, see Fig. 3, where RuleId and ProcId are identifiers that start with lower case and upper case respectively.

Other than executing a set of rule schemata, 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 Fig. 3, 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 command in the condition terminates with a result graph (that is, it neither diverges nor fails) and it is not satisfied if the execution yields failure.

```
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
```

Fig. 3: Abstract syntax of GP 2 programs

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 $(\texttt{ComSeq}\times\mathcal{G}(\mathcal{L})) \cup \mathcal{G}(\mathcal{L}) \cup (\texttt{fail}))$, 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 Fig. 4 and Fig. 5 where $\mathcal{R}$ is a rule set call; C, P, P', and Q are command sequences; and G and H are host graphs.

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

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

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

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

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

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

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

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

Fig. 4: Inference rules for core commands [17]

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

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

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

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

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

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

Fig. 5: Inference rules for derived commands [17]

Definition 14 (Semantic function [17]). The semantic function $[\![\_]\!]$: ComSeq $\to (\mathcal{G}(\mathbb{L}) \to 2^{\mathcal{G}(\mathbb{L})\cup\{fail,\bot\}})$ is defined by

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

$\square$

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 15 (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 $\mathcal{R} = \{r_1, \dots, r_n\}$, 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. $P!$. □

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, the execution of a graph program may yield a proper graph, failure, or diverge/get stuck. The latter only may happen when a loop exists in the program. In some cases, we may want to not considering 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 16 (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 $\mathcal{R} = \{r_1, \dots, r_n\}$, 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. □

3 Monadic Second-Order Formulas for Graph Programs

Here, we define monadic second-order formulas that can be used to express GP 2 graphs. By considering the properties of GP 2 graphs and conditions of rule schemata, we define monadic second-order formulas as an abstract syntax as can be seen in Figure 6. In the syntax, Character is the set of all printable characters except ‘"’ (i.e. ASCII characters 32, 33, and 35-126), and Digit is the digit set $\{0, \ldots, 9\}$.

Formula ::= true | false | Elem | Cond | Equal
 | Formula (‘$\wedge$’ | ‘$\vee$’) Formula | ‘$\neg$’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 (‘$\in$’ | ‘$\notin$’) SetNodeVar | EdgeVar (‘$\in$’ | ‘$\notin$’) SetEdgeVar
Cond ::= (int | char | string | atom) ‘(’Var‘)’
 | Lst (‘=’ | ‘$\neq$’) 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 | l_V ‘(’Node‘)’ | 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 | m_E‘(′EdgeVar‘)′
VMark ::= none | red | blue | green | grey | any | m_V‘(′Node‘)
Equal ::= Node (’=’ | ‘$\neq$’) Node | EdgeVar (’=’ | ‘$\neq$’) EdgeVar
 | Lst (’=’ | ‘$\neq$’) Lst | VMark (’=’ | ‘$\neq$’) VMark
 | EMark (’=’ | ‘$\neq$’) EMark

Fig. 6: Abstract syntax of monadic second-order formulas

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 $\exists_\mathsf{v}\mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{n}(\mathsf{c})$ for $\exists_\mathsf{v}\mathsf{x}_1(\exists_\mathsf{v}\mathsf{x}_2(\ldots\exists_\mathsf{v}\mathsf{x}_\mathsf{n}(\mathsf{c})\ldots))$ (also for other quantifiers). Also, we define ’term’ as the set of variables, constants, and functions in MSO formulas.

If we compare MSO formulas and FO formulas defined in [24], other than additional variables for node set variables (SetNodeVar) and edge set variables (SetEdgeVar), now we have additional function card, predicate path, and Boolean operators $\in, \notin, \subset, \subseteq$.

Also, we define 'term' as the set of variables, constants, and functions in MSO formulas.

Definition 17 (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* □

Remark 2. 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 [4], 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 $\mathsf{card}(\mathsf{X}) = \mathsf{k} * \mathsf{n} + \mathsf{m}$ for some fresh variable k or $(\mathsf{card}(\mathsf{X}) - \mathsf{m})/\mathsf{n} \neq (\mathsf{card}(\mathsf{X}) - \mathsf{m} - 1)/\mathsf{n}$. As an example, we can express $p(0, 2, X)$ by $\mathsf{card}(\mathsf{X}) = 2 * \mathsf{k}$ or $\mathsf{card}(\mathsf{X})/\mathsf{n} \neq (\mathsf{card}(\mathsf{X}) - 1)/\mathsf{n}$.

Example 2 (Monadic second-order formulas).

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

Note that the first example above is a first-order formula, while the second example can be expressed in first-order formula, or monadic second-order formula with ulitising the function card. The third example, can not be expressed in first-order formula since we can not expresses the evenness of nodes in first-order logic [4].

3.1 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 18. Informally, an assignment is a function that maps free variables to their domain.

Table 1: Type of a variable and its domain in a graph G

type	Node	Edge	SetNode	SetEdge	List	Atom	Int	String	Char
domain	V_G	E_G	2^{V_G}	2^{E_G}	$(\mathbb{Z} \cup (\mathrm{Char})^*)^*$	$\mathbb{Z} \cup \mathrm{Char}^*$	$\mathbb{Z}$	Char^*	Char

Definition 18 (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 1. A *formula assignment* of c on a host graph G is a tuple $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ of functions $\alpha_G = \langle \alpha_V : A \to V_G, \alpha_E : B \to E_G, \alpha_{2V} : D \to 2^{V_G}, \alpha_{2E} : E \to 2^{E_G} \rangle \rangle$, 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 FO 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 $s_G(\mathsf{x}^\alpha)$, $t_G(\mathsf{x}^\alpha)$, $\ell^E_G(\mathsf{x}^\alpha), m^E_G(\mathsf{x}^\alpha), \ell^V_G(\mathsf{x}^\alpha), m^V_G(\mathsf{x}^\alpha)$, 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 $\mathsf{x_1}$ and $\mathsf{x_2}$;
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 $\mathsf{x_1}$ and $\mathsf{x_2}$ □

Satisfaction of a formula c on 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 19.

Definition 19 (Satisfaction). Given a graph G and a monadic second-order formula c. G satisfies c, written $G \vDash c$, if there exists an assignment α such that c^α is true in G (denotes by $G \models^\alpha c$). 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 $c^\alpha = \mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}, \mathsf{atom(x)}$, or $\mathsf{root(x)}$, c^α is true ifff $x^\alpha \in \mathbb{Z}, x^\alpha \in \text{Char}, x^\alpha \in \text{Char}^*, x^\alpha \in \mathbb{Z} \cup \text{Char}^*$, or $p_G(x^\alpha) = 1$ respectively.
3. If c^α is in the form $\mathsf{edge(x_1, x_2)}$ 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 $\mathsf{path(x_1, x_2)}$ 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. 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 $\texttt{blue} \ominus_\mathbb{B} t_2 \ \vee \ \texttt{red} \ominus_\mathbb{B} t_2 \ \vee \ \texttt{green} \ominus_\mathbb{B} t_2 \ \vee \ \texttt{grey} \ominus_\mathbb{B} t_2 \ \vee \ \texttt{dashed} \ominus_\mathbb{B} t_2$ 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 $\exists_\mathsf{v}\mathsf{x(b)}$ 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 $\exists_\mathsf{e}\mathsf{x(b)}$ 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 $\exists_\mathsf{l}\mathsf{l(b)}$ 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 $\exists_\mathsf{V}\mathsf{X(b)}$ 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 $\exists_\mathsf{E}\mathsf{X(b)}$ 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 1.

Lemma 1 (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 19, 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. Hence, $\mathsf{path}(\mathsf{x},\mathsf{y})$ 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 $2\leq i\leq n$ 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 $1\leq i\leq n-1$ 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 $1\leq i<j\leq n$ 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.

3.2 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 $\mathsf{edge}(\mathsf{x},\mathsf{y})$, we can express it with $\exists_\mathsf{e}\mathsf{z}(\mathsf{s}(\mathsf{z}) = \mathsf{x} \wedge \mathsf{t}(\mathsf{z}) = \mathsf{y})$. The optional arguments label and mark of the predicate edge, e.g. the predicate $\mathsf{edge}(\mathsf{x},\mathsf{y},5,\mathsf{none})$, can be expressed as:
$\exists_\mathsf{e}\mathsf{z}(\mathsf{s}(\mathsf{z}) = \mathsf{x} \wedge \mathsf{t}(\mathsf{z}) = \mathsf{y} \wedge \mathsf{l}_\mathsf{E}(\mathsf{z}) = 5 \wedge \mathsf{m}_\mathsf{E}\mathsf{z} = \mathsf{none})$.

Then the predicate $\mathsf{path}(\mathsf{x},\mathsf{y}$ can also be express by MSO formula as in Lemma 1. The optional argument edge-set variable Y can be added by conjunct the formulas inside the edge-set quantifier with $\neg(\mathsf{Y} \not\subseteq)\mathsf{X}$.

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

Note that we have proven some properties for first-order formulas in [24]. Hence, to not make a redundant prove, we consider first-order formula as a base case in induction on monadic second-order formulas.

Definition 20 (Structural induction on terms).
Given a property *Prop*. 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 21 (Structural induction on monadic second-order formulas).
Given a property *Prop*. 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 $c_1 \wedge c_2$, $c_1 \vee c_2$, $\neg c_1$, $\exists_\mathsf{v}\mathsf{x}(c_1)$, $\exists_\mathsf{e}\mathsf{x}(c_1)$, $\exists_\mathsf{l}\mathsf{X}(c_1)$, $\exists_\mathsf{V}\mathsf{X}(c_1)$, and $\exists_\mathsf{e}\mathsf{x}(c_1)$. □

Later in following section, 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, $\mathsf{x} = \mathsf{s}(\mathsf{y})$ or $\mathsf{x} = \mathsf{t}(\mathsf{y})$ 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.

3.3 Monadic second-order formulas in rule schema application

To construct a strongest liberal postcondition, we need to express properties that are changes due to a rule schema application. The applications depend on the graphs of the rule schemata, so that we need to express properties of graphs with considering the graphs in the rule schemata. For this, we introduce graph replacement and conditions over a graph in [24] so that we can express properties of graphs based on morphisms we have on the graphs in the rule schemata.

Definition 22 (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. □

Definition 23 (Replacement graph). Given 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. □

We also introduce a generalised rule schema in [24] to be able to have a right-application condition in a rule schema, also to have rule schema without restriction on graph labels and marks. The generalised rule schema make it possible to have an inverse of a rule schema in generalised version. This inverse is later used in the soundness of our construction, and can be used to construct a weakest liberal precondition over a postcondition. Here, we also use the definition of specification of a graph and variablisation of a condition, which were also introduced in [24].

Definition 24 (Specifying a totally labelled graph). *Given 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 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 Root$_L(v)$ for $v \in V_L$ be a function such that Root$_L(v)$ =* $\mathsf{root(v)}$ *if $p_L(v) = 1$, and Root$_L(v) = \neg$*$\mathsf{root(v)}$ *otherwise.* A specification of L, *denoted by Spec(L), is the condition over L:*

$$\bigwedge_{\mathsf{i=1}}^{\mathsf{k}} \mathit{Type}(x_i) \;\wedge\; \bigwedge_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{l_V(v_i)} = \ell_L^V(v_i) \;\wedge\; \mathsf{m_V(v_i)} = m_L^V(v_i) \;\wedge\; \mathit{Root}_L(v_i)$$

$$\wedge \bigwedge_{\mathsf{i=1}}^{\mathsf{m}} \mathsf{s(e_i)} = s_L(e_i) \;\wedge\; \mathsf{t(e_i)} = t_L(e_i) \;\wedge\; \mathsf{l_E(e_i)} = \ell_L^E(e_i) \;\wedge\; \mathsf{m_E(e_i)} = m_L^E(e_i) \square$$

Definition 25 (Variablisation of a condition over a graph). Given 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 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 $c^{[a \mapsto b][d \mapsto e]} = (c^{[a \mapsto b]})^{[d \mapsto e]}$. □

4 Constructing a Strongest Liberal Postcondition

In this section, we present a construction that can be used to obtain a strongest liberal postcondition from a given precondition and a rule schema. Here, we limit the precondition to closed monadic second-order formulas. In [24], we have shown the construction to obtain a strongest liberal postcondition over a first-order formula.

Definition 26 (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. □

As mentioned in [24], our definition of a strongest liberal postcondition is different with the definitions in [8, 6, 5] 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 $G \Rightarrow_r H$. But we have proven that both definitions are actually equivalent.

Lemma 2. Given a rule schema r, 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 $G \Rightarrow_r H$.

As in [24], here we also construct a strongest liberal postcondition by obtaining a left-application condition, which then be used to obtain a right-application condition, so that finally we can use to obtain a strongest liberal postcondition.

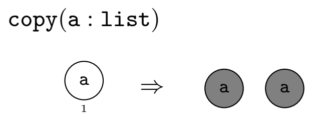
As a running example, let us consider the rule schema copy of Fig. 7 and a closed monadic second-order formula $e \equiv \exists_{\mathsf{V}}\mathsf{X}(\neg\exists_{\mathsf{v}}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \Leftrightarrow \mathsf{x} \in \mathsf{X}) \wedge \exists_{\mathsf{l}}\mathsf{n}(\mathsf{card}(\mathsf{X}) = 2 * \mathsf{n}))$.

Before we start to define the construction formally, let us observe the construction by considering the rule copy of Fig. 7. Let us also consider the formulas c_1 and c_2 bellow:

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

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.



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

Fig. 7: GP 2 conditional rule schema `copy`

4.1 From precondition to left-application condition

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. However, first-order formula is sufficient to express the dangling condition so that we do not need to extend the condition Dang we have in [24].

Definition 27 (Condition Dang). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ where $\{v_1, \ldots, v_n\}$ 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 $V_L - V_K = \emptyset$ then Dang$(r) = \mathsf{true}$
2. if $V_L - V_K \neq \emptyset$ then

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

□

Definition 28 (Transformation Split). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$. where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. 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 Split(c, r) over L inductively as follows:

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

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

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

$$\text{Split}(c, r) = c$$

- Inductive case.
Let c_1 and c_2 be conditions over L.
1) $\text{Split}(c_1 \vee c_2, r) = \text{Split}(c_1, r) \vee \text{Split}(c_2, r)$,
2) $\text{Split}(c_1 \wedge c_2, r) = \text{Split}(c_1, r) \wedge \text{Split}(c_2, r)$,
3) $\text{Split}(\neg c_1, r) = \neg\text{Split}(c_1, r)$,
4) $\text{Split}(\exists_{\mathsf{v}}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i=1}}^{\mathsf{n}}\text{Split}(c_1^{[x\mapsto v_i]}, r)) \vee \exists_{\mathsf{V}}\mathsf{x}(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{x} {\neq} \mathsf{v_i} \wedge \text{Split}(c_1, r)$,
5) $\text{Split}(\exists_{\mathsf{e}}\mathsf{x}(c_1), r) = (\bigvee_{\mathsf{i=1}}^{\mathsf{m}}\text{Split}(c_1^{[x\mapsto e_i]}, r)) \vee \exists_{\mathsf{e}}\mathsf{x}(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}} \mathsf{x} {\neq} \mathsf{e_i} \wedge \text{inc}(c_1, r, x))$,
where

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

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

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 3. From the definition of Split,
$\text{Split}(e, r) = \exists_{\mathsf{V}}\mathsf{X}((1 \in \mathsf{X} \Rightarrow \mathsf{d} \wedge \mathsf{card(X)} = 2^*\mathsf{n}) \wedge (1 \notin \mathsf{X} \Rightarrow \mathsf{d} \wedge \mathsf{card(X)} = 2^*\mathsf{n}))$
where

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

Lemma 3. Given 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,

$$G \models c \text{ if and only if } \rho_g(G) \models \text{Split}(c, r).$$

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).
1) If c is a first order formula, see [24] for the proof.

$$\Leftrightarrow \rho_g(G) \models \text{Split}(c, r)$$

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

$$\begin{aligned} G \models c \quad &\Leftrightarrow \rho_g(G) \models c \\ &\Leftrightarrow \rho_g(G) \models \text{Split}(c, r) \end{aligned}$$

(Inductive case).
Assuming that for some conditions c_1 and c_2 over L, the lemma holds.
1) $G \models c_1 \vee c_2 \quad \Leftrightarrow G \models c_1 \vee G \models c_2$

$$
\begin{array}{ll}
 & \Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \rho_g(G) \models \mathrm{Split}(c_2, r) \\
 & \Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r) \\
2)\ G \models c_1 \wedge c_2 & \Leftrightarrow G \models^{\alpha} c_1 \wedge G \models^{\alpha} c_2 \text{ for some assignment } \alpha \\
 & \Leftrightarrow \rho_g(G) \models^{\beta} \mathrm{Split}(c_1, r) \vee \rho_g(G) \models^{\beta} \mathrm{Split}(c_2, r) \\
 & \quad \text{where } \beta(x) = \alpha(x) \text{ if } x \notin V_L;\ \beta(x) = g^{-1}(\alpha(x)) \text{ otherwise} \\
 & \Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r) \\
3)\ G \models \neg\, c_1 & \Leftrightarrow \neg(G \vDash^{\alpha}\ c_1) \text{ for some assignment } \alpha \\
 & \Leftrightarrow \neg(\rho_g(G) \models^{\beta} \mathrm{Split}(c_1, r)) \\
 & \quad \text{where } \beta(x) = \alpha(x) \text{ if } x \notin V_L;\ \beta(x) = g^{-1}(\alpha(x)) \text{ otherwise} \\
 & \Leftrightarrow \rho_g(G) \models \neg\mathrm{Split}(c_1, r) \\
4)\ G \models \exists_{\mathsf{v}}\mathsf{x}(c_1) & \Leftrightarrow G \models \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 \rho_g(G) \models \bigvee_{\mathsf{i}=1}^{\mathsf{n}} \mathrm{Split}(\mathsf{c}_1^{[\mathsf{x}\mapsto\mathsf{v_i}]}, \mathsf{r}) \vee \exists_{\mathsf{V}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v_i} \wedge \mathrm{Split}(\mathsf{c_1}, \mathsf{r})) \\
5)\ G \models \exists_{\mathsf{e}}\mathsf{x}(c_1) & \Leftrightarrow G \models \bigvee_{\mathsf{i}=1}^{\mathsf{m}} c_1{}^{[\mathsf{x}\mapsto\mathsf{e_i}]} \vee \exists_{\mathsf{V}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1}) \\
 & \Leftrightarrow \rho_g(G) \models \bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}(\mathsf{c}_1^{[\mathsf{x}\mapsto\mathsf{e_i}]}, \mathsf{r}) \vee \exists_{\mathsf{e}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{Split}(\mathsf{c_1}, \mathsf{r})) \\
 & \Leftrightarrow \rho_g(G) \models \bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}(\mathsf{c}_1^{[\mathsf{x}\mapsto\mathsf{e_i}]}, \mathsf{r}) \vee \exists_{\mathsf{e}}\mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{inc}(\mathsf{c_1}, \mathsf{r}, \mathsf{x})) \\
6)\ G \models \exists_{\mathsf{l}}\mathsf{x}(c_1) & \Leftrightarrow G \models c_1 \\
 & \Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \\
 & \Leftrightarrow \rho_g(G) \models \exists_{\mathsf{l}}\mathsf{x}(\mathrm{Split}(c_1, r)) \\
7)\ G \models \exists_{\mathsf{V}}\mathsf{X}(c_1) & \Leftrightarrow G \models \exists_{\mathsf{V}}\mathsf{X}(\bigvee_{\mathsf{i}=0}^{2^{\mathsf{n}}} \mathsf{V_i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j}\in\mathsf{V}-\mathsf{V_i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathsf{c_1}) \\
 & \Leftrightarrow G \models \bigvee_{\mathsf{i}=0}^{2^{\mathsf{n}}} \mathsf{V_i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j}\in\mathsf{V}-\mathsf{V_i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathsf{c_1} \\
 & \Leftrightarrow \rho_g(G) \models \bigvee_{\mathsf{i}=0}^{2^{\mathsf{n}}} \mathsf{V_i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j}\in\mathsf{V}-\mathsf{V_i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathrm{Split}(\mathsf{c_1}, \mathsf{r}) \\
 & \Leftrightarrow \rho_g(G) \models \exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{\mathsf{i}=0}^{2^{\mathsf{n}}}(\mathsf{V_i} \subseteq \mathsf{X} \wedge \bigwedge_{\mathsf{j}\in\mathsf{V}-\mathsf{V_i}} \mathsf{j} \notin \mathsf{X} \Rightarrow \mathrm{Split}(\mathsf{c_1}, \mathsf{r}))) \\
 & \Leftrightarrow \rho_g(G) \models \exists_{\mathsf{V}}\mathsf{X}(\bigwedge_{\mathsf{i}=0}^{2^{\mathsf{n}}}(\mathsf{d_i} \Rightarrow \mathrm{Split}(\mathsf{c_1}, \mathsf{r}))) \\
8)\ \text{analogous to point 7} &
\end{array}
$$

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 one we defined in [24].

Definition 29 (Transformation Val). Given 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 $\mathrm{Val}(c, r)$, 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 $f \in F$,

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

(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. 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.
4. 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 4. Let d be the conditions Split(e, r) from the previous example. Then, Val(d, r) =

$$\exists_\mathsf{V}\mathsf{X}(\mathsf{card(X)} \wedge (1 \in \mathsf{X} \Rightarrow \mathsf{false})$$
$$\wedge\ (1 \notin \mathsf{X} \Rightarrow \neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 1 \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V(x)} = \mathsf{grey}) \vee (\mathsf{m_V(x)} \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))))$$

In the example above, we change every $\mathsf{m_V}(1) \neq \mathsf{grey}$ to true and $\mathsf{m_V}(1) = \mathsf{grey}$ to false because $m_L^V = \mathit{none}$. 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 4. Given 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,

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

Proof. From the case of FO logic (see [24]), we understand that the first step of Val 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 one for FO formulas [24].

Definition 30 (Transformation Lift). Given 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 $\mathrm{Lift}(c, w)$, is the condition over L:

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

$\square$

Example 5. Note that $\Gamma = \mathsf{true}$ and $\mathrm{Dang}(r) = \mathsf{indeg(1)} = \mathsf{0} \wedge \mathsf{outdeg(1)} = \mathsf{0}$ such that $\mathrm{Val}(\Gamma \wedge \mathrm{Dang}(r), r) = \mathsf{true}$. Hence, $\mathrm{Lift}(e, r^\vee) = \mathrm{Val}(\mathrm{Split}(e, r), r)$

Proposition 1 (Left-application condition for MSO formulas). Given 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 $g : L^\alpha \rightarrow G$. For some host graph H,

$$G \vDash c \text{ and } G \Rightarrow_{w,g,g^*} H \text{ implies } \rho_g(G) \vDash (\mathrm{Lift}(c, w))^\alpha$$

Proof. From Lemma 3, we know that $G \vDash c$ implies $\rho_g(G) \vDash \mathrm{Split}(c, r)$. Then $G \Rightarrow_{w,g,g^*} H$ implies $\rho_g(G) \vDash ac_L$ and the existence of natural double-pushout with match $g : L^\alpha \rightarrow G$. The latter implies the satisfaction of the dangling condition. The satisfaction of the dangling condition implies $\rho_g(G) \vDash \mathrm{Dang}(r)$, such that $\rho_g(G) \vDash \mathrm{Split}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r)$, and $\rho_g(G) \vDash \mathrm{Split}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r), r)^\alpha$ from Lemma 4.

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

Definition 31 (Lifted form). Given a rule graph L where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. Let $\{V_1, \ldots, V_{2^n}\}$ 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 $\{E_1, \ldots, E_{2^m}\}$ 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, $\mathsf{x} = \mathsf{s}(\mathsf{y})$ or $\mathsf{x} = \mathsf{t}(\mathsf{y})$ 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. $\exists_\mathsf{v}\mathsf{x}\left(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v}_\mathsf{i} \wedge \mathsf{c}_1\right)$ for some condition c_1 over L in lifted form
9. $\exists_\mathsf{e}\mathsf{x}\left(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e}_\mathsf{i} \wedge \mathsf{c}_1\right)$ for some condition c_1 over L in lifted form
10. $\exists_\mathsf{V}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^\mathsf{n}} \mathsf{d}_\mathsf{i} \Rightarrow \mathsf{c}_\mathsf{i})$ where each c_i is a condition over L in lifted form
11. $\exists_\mathsf{E}\mathsf{X}(\bigwedge_{\mathsf{i}=1}^{2^\mathsf{m}} \mathsf{a}_\mathsf{i} \Rightarrow \mathsf{c}_\mathsf{i})$ 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 5. Given a precondition c and a rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$. Then, $\text{Lift}(c, r^\vee)$ is a condition over L in lifted form.

Proof. Note that $\text{Lift}(c, r^\vee)$ is formed from conjunctions of $\text{Val}(\text{Dang}(r))$, $\text{Val}(\text{Split}(\Gamma, r))$, and $\text{Val}(\text{Split}(c, r))$. From the construction of $\text{Val}(\text{Dang}(r))$, 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 $\mathsf{edge}(\mathsf{u}, \mathsf{v\#m})$ for some nodes u, v in L and some edge mark m. We need to change this to $\exists_\mathsf{e}\mathsf{x}(\mathsf{s}(\mathsf{x}) = \mathsf{u} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v} \wedge \mathsf{m}_\mathsf{E}(\mathsf{x}) = \mathsf{m})$ so that $\text{Val}(\text{Split}(\Gamma, r))$ will be in lifted form; that is form number 9 in Definition 31. Then, for $\text{Val}(\text{Split}(c, r))$, 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 31. Moreover, by Val, 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.

4.2 From left to right-application condition

Similar to what we have in [24], 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 32 (Adjusment in MSO logic). Given 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 $\{V_1, \ldots, V_{2^n}\}$ 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 $\{U_1, \ldots, U_{2^k}\}$ 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 $\{E_1, \ldots, E_{2^m}\}$ 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,
   $\mathrm{Adj}(c, r) = c$
2. If c is predicate $\mathsf{int(x)}, \mathsf{char(x)}, \mathsf{string(x)}$, or $\mathsf{atom(x)}$ for some list variable x,
   $\mathrm{Adj}(c, r) = c$
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,
   $\mathrm{Adj}(c, r) = c$
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,
   $\mathrm{Adj}(c, r) = c$
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,
   $$\mathrm{Adj}(c, r) = \begin{cases} \mathsf{false} & \text{, if } \ominus \in \{=\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ \mathsf{true} & \text{, if } \ominus \in \{\neq\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ c' & \text{, otherwise} \end{cases}$$
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,
   $\mathrm{Adj}(c, r) = c$
7. If $c = \exists_\mathsf{l}\mathsf{x}(\mathsf{c_1}$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_\mathsf{l} x(\mathrm{Adj}(c_1, r))$
8. If $c = \exists_\mathsf{v}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1}\right)$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_\mathsf{v} x(\bigwedge_{i=1}^{p}, x \neq w_i \wedge \mathrm{Adj}(c_1, r))$
9. If $c = \exists_\mathsf{e}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1}\right)$ for some condition c_1 over L in lifted form,
   $\mathrm{Adj}(c, r) = \exists_\mathsf{e} x(\bigwedge_{i=1}^{q}, x \neq z_i \wedge \mathrm{Adj}(c_1, r))$
10. If $c = \exists_\mathsf{V}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^n}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form or contains $\mathsf{card(X)}$
    $\mathrm{Adj}(c, r) = \exists_\mathsf{V} X(\bigwedge_{v \in V_R - V_K} v \notin X \bigwedge_{i=1}^{2^k}(b_i \Rightarrow \bigvee_{j \in W_i} c'_j))$
    where $c'_j = \mathrm{Adj}(c_j, r)^{[\mathsf{card(X)} \mapsto \mathsf{card(X)} + |(\mathsf{V_L} - \mathsf{V_K}) \cap \mathsf{V_j}|]}$ 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 $c = \exists_\mathsf{E}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^m}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form, construction of $\mathrm{Adj}(c, r)$ 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,
    $\mathrm{Adj}(c, r) = \mathrm{Adj}(c_1, r) \vee \mathrm{Adj}(c_2, r)$

13. If $c = c_1 \wedge c_2$ for some conditions c_1, c_2 over L in lifted form,
 $\text{Adj}(c, r) = \text{Adj}(c_1, r) \wedge \text{Adj}(c_2, r)$
14. If $c = \neg c_1$ for some condition c_1 over L in lifted form,
 $\text{Adj}(c, r) = \neg\text{Adj}(c_1, r)$

Example 6.
Let $f = \text{Lift}(e, r^\vee)$.

$$\begin{aligned}\text{Adj}(f, r) = \exists_\mathsf{V}\mathsf{X}(\neg\exists_\mathsf{v}\mathsf{x}(&\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \\ &\wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \\ \wedge\ &\mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})\end{aligned}$$

In the above example, we change every occurrence of $\mathsf{x} \neq 1$ 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. Given 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 $\text{Lift}(d, r^\vee)$. Then,

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

Proof. From Lemma 5, we know that $c = \text{Lift}(d, r^\vee)$ 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 $\rho_{g^*}(H) \vDash (\text{Adj}(c, r))^\alpha$ holds. Base case.

1. if $c = \mathsf{true}$, then $\text{Adj}(c, r) = \mathsf{true}$ such that $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash \text{Adj}(c, r))^\alpha$ holds.
2. if c is predicate $\mathsf{int}(\mathsf{x}), \mathsf{char}(\mathsf{x}), \mathsf{string}(\mathsf{x})$, or $\mathsf{atom}(\mathsf{x})$ 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 $\rho_{g^*}(H) \vDash c = \text{Adj}(c, r)$.
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 $\rho_{g^*}(H) \vDash c = \text{Adj}(c, r)$.
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 $\mathsf{s}(\mathsf{x})$ or $\mathsf{t}(\mathsf{x})$ 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 31, 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 $\rho_{g^*}(H) \vDash c = \text{Adj}(c, r)$.

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. See [24] for the proof.
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 31, v is not in L so that v must be in $\rho_g(G) - L$. Hence, v is in $\rho_{g^*}(H)$ so that $\rho_{g^*}(H) \vDash x \in X = \mathrm{Adj}(c, r)$.

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 $\rho_{g^*}(H) \vDash \mathrm{Adj}(c_i, r)^\alpha$. Then,

1. if $c = \exists_\mathsf{l}\mathsf{x}(\mathsf{c_1}$ 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, $\mathrm{Adj}(c_1, r)^\alpha$ is true in $\rho_{g^*}(H)$ when we substitute i for x. Hence, $\rho_{g^*}(H) \vDash \mathrm{Adj}(c, r)$.
2. if $c = \exists_\mathsf{v}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1}\right)$ 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 $\mathrm{Adj}(c_1, r)$ 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, $\rho_{g^*}(H) \vDash \mathrm{Adj}(c_1, r))$.
3. if $c = \exists_\mathsf{e}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1}\right)$ 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 $\mathrm{Adj}(c_1, r)$ 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, $\rho_{g^*}(H) \vDash \mathrm{Adj}(c_1, r))$.
4. if $c = \exists_\mathsf{V}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^n}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$, $\rho_g(G) \vDash c^\alpha$ 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 $\rho_{g^*}(H) \vDash \mathrm{Adj}(c_i)$. 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, $\rho_{g^*}(H) \vDash \bigvee_{j \in W_i} \mathrm{Adj}(c_j, r)$.
   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 $\mathsf{card(A)} + |(\mathsf{V_L} - \mathsf{V_K}) \cap \mathsf{V_j}|$ in $\rho_{g^*}(H)$.
5. If $c = \exists_\mathsf{E}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^m}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ 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 $\rho_{g^*}(H) \vDash \text{Adj}(c_1, r)^\alpha$, and $\rho_g(G) \vDash c_2^\alpha$ implies $\rho_{g^*}(H) \vDash \text{Adj}(c_2, r)^\alpha$. Hence, $\rho_{g^*}(H) \vDash \text{Adj}(c, r)$.
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 $\rho_{g^*}(H) \vDash {}^\beta\text{Adj}(c_1, r)^\alpha$, and $\rho_g(G) \vDash {}^\beta c_2^\alpha$ implies $\rho_{g^*}(H) \vDash {}^\beta\text{Adj}(c_2, r)^\alpha$. This means, $\rho_{g^*}(H) \vDash \text{Adj}(c_1, r)^\alpha \wedge \text{Adj}(c_2, r)^\alpha$. Hence, $\rho_{g^*}(H) \vDash \text{Adj}(c, r)$.
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 $\neg\rho_{g^*}(H) \vDash {}^\beta\text{Adj}(c_1, r)^\alpha$, such that $\rho_{g^*}(H) \vDash \text{Adj}(c, r)^\alpha$.

Lemma 7. Given 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 $\text{Lift}(d, r^\vee)$. Then,

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

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

1. if $c = \mathsf{true}$ or $c = \mathsf{false}$, then $\text{Adj}(c, r) = c$ and $\text{Adj}(\text{Adj}(c, r), r^{-1}) = c$. 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 $\text{Adj}(c, r) = c$ and $\text{Adj}(\text{Adj}(c, r), r^{-1}) = c$. 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 $\text{Adj}(c, r) = c$ and $\text{Adj}(\text{Adj}(c, r), r^{-1}) = c$. 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 $\text{Adj}(c, r) = c$ and $\text{Adj}(\text{Adj}(c, r), r^{-1}) = c$. 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 $\text{Adj}(c, r) = c$ and $\text{Adj}(\text{Adj}(c, r), r^{-1}) = c$. 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 $\rho_g(G) \vDash \text{Adj}(\text{Adj}(c_i, r), r^{-1})^\alpha$. Then,

1. if $c = \exists_\mathsf{l}\mathsf{x}(\mathsf{c_1}$ 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, $\text{Adj}(\text{Adj}(c_1, r), r^{-1})^\alpha$ is true in $\rho_{g^*}(H)$ when we substitute i for x. Hence, *prop* holds.

2. if $c = \exists_\mathsf{v}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathsf{c_1}\right)$,
   $\mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1}) = \exists_\mathsf{v}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{n}}, \mathsf{x} \neq \mathsf{v_i} \wedge \mathrm{Adj}(\mathrm{Adj}(\mathsf{c_1}, \mathsf{r}), \mathsf{r}^{-1})\right)$. From the assumption, $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(c_1, r), r^{-1}))$ implies $\rho_g(G) \vDash c_1$ so that *prop* holds.
3. if $c = \exists_\mathsf{e}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathsf{c_1}\right)$,
   $\mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1}) = \exists_\mathsf{e}\mathsf{x}\left(\bigwedge_{\mathsf{i=1}}^{\mathsf{m}}, \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{Adj}(\mathrm{Adj}(\mathsf{c_1}, \mathsf{r}), \mathsf{r}^{-1})\right)$. From the assumption, $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(c_1, r), r^{-1}))$ implies $\rho_g(G) \vDash c_1$ so that *prop* holds.
4. if $c = \exists_\mathsf{V}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^n}} \mathsf{d_i} \Rightarrow \mathsf{c_i})$,

5. If $c = \exists_\mathsf{E}\mathsf{X}(\bigwedge_{\mathsf{i=1}}^{\mathsf{2^m}} \mathsf{a_i} \Rightarrow \mathsf{c_i})$ where each c_i is a condition over L in lifted form, the proof is analogous to point 10
6. If $c = c_1 \vee c_2$,
   $\mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1}) = \mathrm{Adj}(\mathrm{Adj}(c_1, r), r^{-1}) \vee \mathrm{Adj}(\mathrm{Adj}(c_2, r), r^{-1})$. From assumption, $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1})$ iff $\rho_g(G) \vDash c_1 \vee c_2 = c$.
7. If $c = c_1 \wedge c_2$,
   $\mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1}) = \mathrm{Adj}(\mathrm{Adj}(c_1, r), r^{-1}) \wedge \mathrm{Adj}(\mathrm{Adj}(c_2, r), r^{-1})$. From assumption, $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1})$ iff $\rho_g(G) \vDash c_1 \wedge c_2 = c$.
8. If $c = \neg c_1$,
   $\mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1}) = \neg\mathrm{Adj}(\mathrm{Adj}(c_1, r), r^{-1})$. From assumption,
   $\rho_g(G) \vDash \mathrm{Adj}(\mathrm{Adj}(c, r), r^{-1})$ iff $\rho_g(G) \vDash \neg c_1 = c$.

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

Definition 33 (Shifting). Given 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. c and w, denoted by $\mathrm{Shift}(c, w)$, is defined as:

$$\mathrm{Shift}(c, w) = \mathrm{Adj}(\mathrm{Lift}(c, w), r) \wedge ac_R \wedge \mathrm{Spec}(R) \wedge \mathrm{Dang}(r^{-1})$$

□

Example 7.

$$\begin{aligned}
\text{a) } \mathrm{Shift}(c_1, r^\vee) = {} & \neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{m_V(x)} \neq \mathsf{none}) \\
& \wedge \mathsf{m_V(2) = grey} \wedge \mathsf{m_V(3) = grey} \wedge \mathsf{l_V(2) = a} \wedge \mathsf{l_V(3) = a} \\
& \wedge \mathsf{indeg(2) = 0} \wedge \mathsf{indeg(3) = 0} \wedge \mathsf{outdeg(2) = 0} \wedge \mathsf{outdeg(3) = 0} \\
\text{b) } \mathrm{Shift}(c_2, r^\vee) = {} & \exists_\mathsf{V}\mathsf{X}(\neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \wedge \mathsf{x} \in \mathsf{X} \wedge \mathsf{m_V(x)} \neq \mathsf{none}) \wedge \mathsf{card(X)} \geq 1) \\
& \wedge \mathsf{m_V(2) = grey} \wedge \mathsf{m_V(3) = grey} \wedge \mathsf{l_V(2) = a} \wedge \mathsf{l_V(3) = a} \\
& \wedge \mathsf{indeg(2) = 0} \wedge \mathsf{indeg(3) = 0} \wedge \mathsf{outdeg(2) = 0} \wedge \mathsf{outdeg(3) = 0} \\
\text{c) } \mathrm{Shift}(c_3, r^\vee) = {} & \exists_\mathsf{V}\mathsf{X}(\neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq 2 \wedge \mathsf{x} \neq 3 \\
& \qquad\qquad \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V(x) = grey}) \vee (\mathsf{m_V(x)} \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X}))) \\
& \qquad \wedge \mathsf{card(X) = 2^*n}) \\
& \wedge \mathsf{m_V(2) = grey} \wedge \mathsf{m_V(3) = grey} \wedge \mathsf{l_V(2) = a} \wedge \mathsf{l_V(3) = a} \\
& \wedge \mathsf{indeg(2) = 0} \wedge \mathsf{indeg(3) = 0} \wedge \mathsf{outdeg(2) = 0} \wedge \mathsf{outdeg(3) = 0}
\end{aligned}$$

Lemma 8. Given 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/edge i where $m_L(i) = m_R(i) = \mathtt{any}$,

$$\rho_{g^*}(H) \vDash (\mathrm{Adj}(\mathrm{Lift}(d,w)), r)^\beta \text{ if and only if } \rho_{g^*}(H) \vDash (\mathrm{Shift}(d,w))^\beta$$

Proof. It is obvious that $\mathrm{Adj}(\mathrm{Lift}(d,w)), r)^\beta$ is implied by $\mathrm{Shift}(d,w)^\beta$, so now we show that $\mathrm{Adj}(\mathrm{Lift}(d,w)), r)^\beta$ implies $\mathrm{Shift}(d,w)^\beta$. That is, $ac_R^\beta \wedge \mathrm{Spec}(R)^\beta \wedge \mathrm{Dang}(r^{-1})^\beta$ is satisfied by $\rho_{g^*}(H)$.

4.3 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 define the formula Post.

Definition 34 (Formula Post). Given a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule $r = \langle L \leftarrow K \rightarrow R\rangle$ and a precondition c. A postcondition w.r.t. c and w, denoted by $\mathrm{Post}(c,w)$, is the monadic second-order formula:

$$\exists_\mathsf{v} x_1, \cdots, x_n(\exists_\mathsf{e} y_1, \cdots, y_m(\exists_\mathsf{l} z_1, \cdots, z_k(\mathrm{Var}(\mathrm{Shift}(c,w))))).$$

where $\{x_1, \cdots, x_n\}$, $\{y_1, \cdots, y_m\}$, and $\{z_1, \cdots, z_k\}$ denote the set of free node, edge, and label (resp.) variables in $\mathrm{Var}(\mathrm{Shift}(c,w))$. We then denote by $\mathrm{Slp}(c,r)$ the formula $\mathrm{Post}(c, r^\vee)$, and $\mathrm{Slp}(c, r^{-1})$ for the formula $\mathrm{Post}(c, (r^\vee)^{-1})$. □

Example 8.

$$\begin{aligned}\mathrm{Slp}(e, r^\vee) = \exists_\mathsf{v}&\mathsf{y}, \mathsf{z}(\exists_\mathsf{l}\mathsf{a}(\\ &\exists_\mathsf{V}\mathsf{X}(\neg\exists_\mathsf{v}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge \mathsf{x} \neq \mathsf{z}\\ &\qquad\qquad \wedge ((\mathsf{x} \notin \mathsf{X} \wedge \mathsf{m_V}(\mathsf{x}) = \mathsf{grey}) \vee (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{grey} \wedge \mathsf{x} \in \mathsf{X})))\\ &\qquad \wedge\ \mathsf{card}(\mathsf{X}) = 2^*\mathsf{n})\\ &\wedge \mathsf{m_V}(\mathsf{y}) = \mathsf{grey} \wedge \mathsf{m_V}(\mathsf{z}) = \mathsf{grey} \wedge \mathsf{l_V}(\mathsf{y}) = \mathsf{a} \wedge \mathsf{l_V}(\mathsf{z}) = \mathsf{a}\\ &\wedge \mathsf{indeg}(\mathsf{y}) = 0 \wedge \mathsf{indeg}(\mathsf{z}) = 0 \wedge \mathsf{outdeg}(\mathsf{y}) = 0 \wedge \mathsf{outdeg}(\mathsf{z}) = 0))\end{aligned}$$

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.

Proposition 2 (Post). Given 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,

$$\rho_{g^*}(H) \vDash (\mathrm{Shift}(c,w))^\beta \text{ if and only if } H \vDash \mathrm{Post}(c,w)^\beta$$

Proof. Note that in the construction of Lift, Shift, and Post, we do not change any set variable to a set constant. Hence, there is no new set constant in Shift. Hence, $\rho_{g^*}(H) \vDash$ Shift$(c, w))^\beta$ iff $H \vDash$Var(Shift$(c, w))^\beta$. 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 Var(Shift$(c, w)))^\beta$ so that there is no additional node (or edge) quantifier for Var(Shift$(c, w)))^\beta$. If there exists a node (or edge) in H, then adding an existential quantifier will not change its satisfaction on H. Hence, $H \vDash$Var(Shift$(c, w)))^\beta$ iff $H \vDash$ Post$(c, w)^\beta$.

Finally, we show that Post$(c, r^\vee)$ is a strongest liberal postcondition w.r.t. c and r. That is, by showing that for all host graph G, $G \vDash c$ and $G \Rightarrow_r H$ implies $H \vDash$ Post$(c, r^\vee)$, and showing that for all host graph H, $H \vDash$ Post$(c, r^\vee)$ implies the existence of host graph G such that $G \vDash c$ and $G \Rightarrow_r H$.

Theorem 1 (Strongest liberal postconditions). Given a precondition c and a conditional rule schema $r = \langle\langle L \leftarrow K \rightarrow R\rangle, \Gamma\rangle$. Then, Slp$(c, r)$ is a strongest liberal postcondition w.r.t. c and r.

Proof. From the definition of generalised rule schema (see [24]), $G \Rightarrow_r H$ iff $G \Rightarrow_{w,g,g*} H$ for some injective morphisms $g : L^\alpha \rightarrow G$ and $g^* : R^\beta \rightarrow H$ 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/edge i where $m_L(i) = m_R(i) =$ `any`. From Proposition 1, Lemma 6, Lemma 8, and Proposition 2, $G \vDash c$ and $G \Rightarrow_{r^\vee,g,g^*} H$ implies $\rho_g(G) \vDash$ Lift$(c, r^\vee))^\alpha$ implies $\rho_{g^*}(H) \vDash$ Shift$(c, r^\vee)^\beta$ implies $H \vDash$ Post$(c, r^\vee)$. Hence, Post$(c, r^\vee)$ is a liberal postcondition w.r.t. c and r.

To show that Post$(c, r^\vee)$ is a strongest liberal postcondition, based on Lemma 2, we need to show that for every graph H satisfying *postM*$(c, r^\vee)$, there exists a host graph G satisfying c such that $G \Rightarrow_r H$.

Recall the construction of Shift$(c, r^\vee)$. A graph satisfying Shift$(c, r^\vee)$ must satisfying Spec(R) such that $H \vDash$ (Post$(c, r^\vee)$) implies $H \vDash$ (Post$(c, r^\vee))^\beta$ for some label assignment β_R, which implies $H \vDash$Var(Spec$(R))^\beta \equiv$Var(Spec(R^β)). This implies the existence of an injective morphism $g^* : R^\beta \rightarrow H$. From Proposition 2, $H \vDash$ Post$(c, r^\vee)^\beta$ implies $\rho_{g^*}(H) \vDash$ Shift$(c, r^\vee)^\beta$. From the construction of Shift$(c, r^\vee)$, Dang(r^{-1}) asserts that the dangling condition is satisfied by g^*. Hence, there exists a natural double-pushout bellow where every morphism is inclusion:

$$\begin{tikzcd}
R^\beta \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & L^\alpha \arrow[d] \\
\rho_{g^*}(H) & D \arrow[l] \arrow[r] & A
\end{tikzcd}$$

(1) (2)

Since $\rho_{g^*}(H) \vDash$ Adj(Lift$(c, r^\vee), r)^\beta$, from Lemma 6 this implies A satisfies Adj(Adj(Lift$(c, r^\vee), r), r^{-1})^\alpha$. From Lemma 7, this implies $A \vDash c^\alpha$. Since direct derivations are invertible, $A \Rightarrow_{r^\vee,g,g^*} H$. Hence, $A \Rightarrow_r H$.

4.4 Complexity of a strongest liberal postcondition

Weakest liberal preconditions produced in [19, 15] can produce unwieldy expressions. 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.

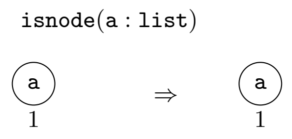


Fig. 8: Rule schema `isnode`

Example 9. Let us consider the rule `isnode` of Fig. 8. Let $\mathrm{P}(n)$ for $n = 1, 2, 3, \ldots$ be formula $\exists_v x_1, \ldots, x_n(l_V(x_1) < 1 \wedge \ldots \wedge l_V(x_n) < n)$. Table 5 shows us the obtained $\mathrm{Slp}(\mathrm{P}(n)$,`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 5: Strongest liberal postcondition over $\mathrm{P}(n)$ and `isnode`

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

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 bind by nested quantifiers, and where Val does not simplify the condition obtained from Split.

Proposition 3. *In the worst-case, the construction of a strongest liberal postcondition can result in an exponential blow-up of the given precondition size.*

Now, let us consider a 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 28). 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 Val. 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 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.

5 Proof Calculi

In this section, we define two proof calculi in the sense of total correctness, that are SEM and SYN, where SEM considers semantic assertions as pre- and postconditions while SYN consider closed monadic second-order formulas as pre- and postconditions.

5.1 Semantic proof calculus

Now let us consider a total correctness, where a Hoare triple $\{c\}\ P\ \{d\}$ for assertions c, d and a graph program P is true iff $\{c\}\ P\ \{d\}$ is partially correct and for every host graph $G \vDash c$, the execution of P on G cannot diverge or get stuck.

In the previous section, we have discussed a strongest liberal postcondition. Now, let us also consider a weakest liberal precondition over a postcondition.

Definition 35 (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,

$$G \vDash c \text{ and } H \in [\![P]\!]G \text{ implies } H \vDash d.$$

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. □

In [8], it is shown that if we have a construction for a strongest liberal postcondition, we can also obtain a weakest liberal precondition from a given rule schema and postcondition.

Lemma 9. Given a conditional rule schema r and a semantic assertion d. Then for all host graphs G,

$$G \vDash \mathrm{WLP}(r, d) \text{ if and only if } G \vDash \neg\mathrm{SLP}(\neg d, r^{-1}).$$

Proof.
$G \vDash \mathrm{WLP}(r, d)$ iff $\forall H.G \Rightarrow_r H$ implies $H \vDash d)$
iff $\neg(\exists H.G \Rightarrow_r H \wedge H \vDash \neg d)$
iff $\neg(\exists H, g, g^*.G \Rightarrow_{(r^\vee),g,g^*} H \wedge H \vDash \neg d)$ □
iff $\neg(\exists H, g, g^*.H \Rightarrow_{(r^\vee)^{-1},g^*,g} G \wedge H \vDash \neg d)$
iff $G \vDash \neg\mathrm{SLP}(\neg d, (r^\vee)^{-1})$

Definition 36 (Partial correctness [21]). A graph program P is *partially correct* with respect to a precondition c and a postcondition d, denoted by $\vDash_{\mathrm{par}}$ $\{c\}\ P\ \{d\}$ if for every host graph G and every graph H in $[\![P]\!]G$, $G \models c$ implies $H \models d$. □

Definition 37 (Total correctness [19]). Given assertions c, d and graph program P. P is *totally correct* with respect to a precondition c and postcondition d, denoted by $\vDash_{\text{tot}}$ $\{c\}$ P $\{d\}$, $\{c\}$ P $\{d\}$ is partially correct and if for every graph $G \in \mathcal{G}(\mathcal{L})$ such that $G \vDash c$, there is no infinite sequence $\langle P, G \rangle \to \langle P_1, G_1 \rangle \to \langle P_2, G_2 \rangle \to \ldots$ (divergence), and there is no terminal configuration $\langle Q, H \rangle$ such that $\langle P, G \rangle \to^* \langle Q, H \rangle$ (getting stuck).

A program can get stuck if it contains a command `if/try` C `then` P `else` Q such that C can diverge from a graph G, or it contains a loop $B!$ whose body B can diverge from a graph G. Hence, getting stuck is always a signal of divergence. A graph program may diverge when it has a loop whose body does not reduce the number of elements with certain properties that is related to the execution of the body. To express the reduction, we introduce a termination function $\#$. This function was introduced in [19], but we make some modifications to consider the command `break`.

Definition 38 (Termination function; #-decreasing). A *termination function* is a mapping $\# : \mathcal{G}(\mathcal{L}) \to \mathbb{N}$ from (semantic) graphs to natural numbers. Given an assertion c and a graph program P. We say P is *#-decreasing* (under c) if for all graphs $G, H \in \mathcal{G}(\mathcal{L})$ such that $G \vDash c$,

$$\langle P, G \rangle \to^* H \text{ implies } \#G > \#H \text{ or } \langle P, G \rangle \to^* \langle \texttt{break}, H \rangle.$$

Here, we consider the 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`.

To define a semantic proof calculus, we need assertions that can express preconditions of failing or successful executions. For this, we also use the assertion SUCCESS and FAIL as defined in [24]. We also use the predicate Break to consider programs with a loop whose body containing the command `break`.

Definition 39 (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 40 (Assertion FAIL). Given a graph program P. FAIL(P) is the predicate on host graphs where for all host graph G,

$$G \vDash \text{FAIL}(P) \text{ if and only if fail} \in [\![P]\!]G.$$

Definition 41 (Predicate Break). Given a graph program P and assertions c and d. Break(c, P, d) is the predicate defined by:

$$\text{Break}(c, P, d) \text{ holds iff for all derivations } \langle P, G \rangle \to^* \langle \texttt{break}, H \rangle, G \vDash c \text{ implies } H \vDash d.$$

□

$$[\text{ruleapp}]_{\text{slp}}\ \frac{}{\{c\}\ r\ \{\text{SLP}(c,r)\}}$$

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\neg\text{SLP}(\neg d, r^{-1})\}\ r\ \{d\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P;Q\ \{d\}}$$

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

$$[\text{if}]\ \frac{\{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\} \quad \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge \text{SUCCESS}(C)\}\ C;P\ \{d\} \quad \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{alap}]\ \frac{\{c\}\ P\ \{c\} \qquad P \text{ is \#-decreasing under } c \qquad \text{Break}(c,P,d)}{\{c\}\ P!\ \{(c \wedge \text{FAIL}(P)) \vee d\}}$$

Fig. 9: Calculus SEM of semantic total correctness proof rules

Definition 42 (Semantic proof calculus). The semantic partial correctness proof rules for core commands, denoted by SEM, is defined in Fig. 9, 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.

5.2 Syntactic proof calculus

In [24], we have shown that if we can construct a strongest liberal postcondition over a rule schema, then we also can construct a weakest liberal precondition over a rule schema, a strongest liberal postcondition over a loop-free program, precondition for failure execution of an iteration command, and precondition for successful execution of a loop-free program.

Definition 43 (App(r) [24]). Given a conditional rule schema $r : \langle L \leftarrow K \rightarrow R,\ \Gamma\rangle$. The formula App($r$) is defined as

$$\text{App}(r) = \text{Var}(\text{Spec}(L) \wedge \text{Dang}(r) \wedge \Gamma).$$

□

Definition 44 (Slp, Success, Fail, Pre of a loop-free program [24]). Given a condition c and a loop-free program S. The monadic second-order formulas Slp(c, S), Pre(c, S), Success(S), and Fail(S) are defined inductively:

1. If S is a set of rule schemata $\mathcal{R} = \{r_1, \ldots, r_n\}$,

   (a) $\text{Slp}(c, S) = \begin{cases} \text{Post}(c, r_1^\vee) \vee \ldots \vee \text{Post}(c, r_n^\vee) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$

   (b) $\text{Pre}(S, c) = \begin{cases} \text{Post}(c, (r_1^\vee)^{-1}) \vee \ldots \vee \text{Post}(c, (r_n^\vee)^{-1}) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$

   (c) $\text{Success}(S) = \begin{cases} \text{App}(r_1) \vee \ldots \vee \text{App}(r_n) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$

   (d) $\text{Fail}(S) = \begin{cases} \neg(\text{App}(r_1) \vee \ldots \vee \text{App}(r_n)) & \text{if } n > 0, \\ \mathsf{false} & \text{otherwise} \end{cases}$

2. For loop-free programs C, P, and Q,
   (i) If $S = P$ `or` Q,
      (a) Slp(c, S)=Slp(c, P)$\vee$Slp(c, Q)
      (b) Pre(S, c)=Pre(P, c)$\vee$Pre(Q, c)
      (c) Success(S) =Success(P)$\vee$Success(Q)
      (d) Fail(S) =Fail(P)$\vee$Success(Q)
   (ii) If $S = P; Q$,
      (a) Slp(c, S)=Slp(Slp(c, P), Q)
      (b) Pre(S, c)=Pre(P,Pre(Q, c))
      (c) Success(S) =Pre(P, Success(Q))
      (d) Fail(S) =Fail(P)$\vee$Pre(P,Fail(Q))
   (iii) If $S =$ `if` C `then` P `else` Q,
      (a) Slp(c, S)=Slp($c \wedge$ Success(C), P) $\vee$ Slp($c \wedge$ Fail(C), Q)
      (b) Pre(S, c)=(Success(C)$\wedge$Pre(P, c)) $\vee$ (Fail(C)$\wedge$Pre(Q, c))
      (c) Success(S) = (Success(C)$\wedge$Success(P) $\vee$ (Fail(C)$\wedge$Success(Q))
      (d) Fail(S)=(Success(C)$\wedge$Fail(P)) $\vee$ (Fail(C)$\wedge$Fail(Q))
   (iv) If $S =$ `try` C `then` P `else` Q,
      (a) Slp(c, S)=Slp($c \wedge$ Success(C), $C; P$) $\vee$ Slp($c \wedge$ Fail(C), Q)
      (b) Pre(S, c) =Pre(C, Pre(P, c)) $\vee$ (Fail(C)$\wedge$Pre(Q, c))
      (c) Success(S) =Pre(C,Success(P)) $\vee$ (Fail(C)$\wedge$Success(Q))
      (d) Fail(S) =Pre(Fail(P), C)) $\vee$ (Fail(C)$\wedge$Fail(Q))

$\square$

Theorem 2 (Slp, Pre, Success, and Fail [24]). For all condition c and loop-free program S, the following holds:

(a) Slp(c, S) is a strongest liberal postcondition w.r.t. c and S
(b) For all host graph G, $G \vDash \text{Pre}(S, c)$ if and only if there exists host graph H such that $H \in [\![S]\!]G$ and $H \vDash c$
(c) $G \vDash \text{Success}(S)$ if and only if $G \vDash \text{SUCCESS}(S)$
(d) $G \vDash \text{Fail}(S)$ if and only if $G \vDash \text{FAIL}(S)$

The proof of the above theorem can be seen in [24]. Although we only consider first-order formulas in [24], here we have shown that $G \vDash \text{SLP}(c, r)$ iff $G \vDash \text{Slp}(c, r)$ (Theorem 1). Hence. the we can follow the proof we have in [24] to prove the above theorem.

Definition 45 (Fail of iteration commands [24]). Let $\mathrm{Fail}_{\mathrm{lf}}(C)$ denotes the formula $\mathrm{Fail}(C)$ for a loop-free program C as defined in Definition 44. For any iteration command S,

$$\mathrm{Fail}(S) = \begin{cases} \mathsf{false} & \text{if } S \text{ is a non-failing command} \\ \mathrm{Fail}_{\mathrm{lf}}(S) & \text{if } S \text{ is a loop-free program} \\ \mathrm{Fail}(C) & \text{if } S = C; P \text{ for a loop-free program } C\text{, a non-failing program } P \end{cases}$$

□

Theorem 3. Given an iteration command S. Then [24],

$$G \vDash \mathrm{Fail}(S) \text{ if and only if } G \vDash \mathrm{FAIL}(S).$$

The proof for Theorem 3 can be seen in [24].

For syntactic proof calculus, we use the same proof rules as in the semantic proof calculus. However, since there are limitation on programs where we can construct MSO formulas Fail and Success, we need to limit the programs that can use the proof calculus.

Definition 46 (Syntactic total correctness proof rules). The syntactic total correctness proof rules, denoted by SYN, is defined in Fig. 10, 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`. □

$$[\text{ruleapp}]_{\text{slp}}\ \frac{}{\{c\}\ r\ \{\mathrm{Slp}(c,r)\}}$$

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\neg\mathrm{Slp}(\neg d, r^{-1})\}\ r\ \{d\}}$$

$$[\text{ruleset}]\ \frac{\{c\}\ r\ \{d\} \text{ for each } r \in \mathcal{R}}{\{c\}\ \mathcal{R}\ \{d\}}$$

$$[\text{comp}]\ \frac{\{c\}\ P\ \{e\} \quad \{e\}\ P\ \{d\}}{\{c\}\ P; Q\ \{d\}}$$

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

$$[\text{if}]\ \frac{\{c \wedge \mathrm{Success}(C)\}\ P\ \{d\} \quad \{c \wedge \mathrm{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{try}]\ \frac{\{c \wedge \mathrm{Success}(C)\}\ C; P\ \{d\} \quad \{c \wedge \mathrm{Fail}(C)\}\ Q\ \{d\}}{\{c\}\ \texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

$$[\text{alap}]\ \frac{\{c\}\ S\ \{c\} \qquad P \text{ is } \#\text{-decreasing under } c \qquad \mathrm{Break}(c, , d)}{\{c\}\ S!\ \{(c \wedge \mathrm{Fail}(S)) \vee d\}}$$

Fig. 10: Calculus SYN of syntactic total correctness proof rules

5.3 Verification of graph programs

By using the proof calculi we defined above, we can verify graph programs by using proof tree [20].

Definition 47 (Provability; proof tree[19]). A triple $\{c\}\ P\ \{d\}$ is provable in I, denoted by $\vdash_{\mathrm{I}} \{c\}\ P\ \{d\}$, if one can construct a *proof tree* from the axioms and inference rules of I with that triple as the root. If $\{c\}\ P\ \{d\}$ is an instance of an axiom X then

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

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

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

is a proof tree, and $\vdash_{\mathrm{I}} \{c\}\ P\ \{d\}$. □

Definition 48 (Structural induction on proof trees). Given 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. Assuming that *Prop* holds for each premise T of inference rules in the proof rules, show that *Prop* holds for the conclusion of each inference rules in the proof rules. □

6 Soundness and completeness of proof calculi

In this section, we show that our proof calculi are sound, in the sense that if some triple can be proven in a calculus, then the triple must be totally correct. In addition, we also discuss about the relative completeness of the proof calculi.

6.1 Soundness

Theorem 4 (Soundness for SEM). Given a graph program P and assertions c, d,

$$\vdash_{\mathrm{SEM}} \{d\}\ P\ \{d\} \text{ implies } \vDash_{\mathrm{tot}} \{c\}\ P\ \{d\}.$$

Proof. Here, we proof the implication by induction on proof trees.
Base case.

a) $[\text{ruleapp}]_{\text{slp}}$. Suppose that $\vdash_{\mathrm{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs H, $H \vDash d$ iff $H \vDash \mathrm{SLP}(c, r)$. Suppose that $G \vDash c$. From Definition 26, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash_{\mathrm{par}} \{c\}\ r\ \{d\}$. Note that r only have one transition, which either transform the initial graph or fail, r can not diverge. Hence, $\vDash_{\mathrm{tot}} \{c\}\ r\ \{d\}$.

b) $[\text{ruleapp}]_{\text{wlp}}$. Suppose that $\vdash_{\text{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs G, $G \vDash c$ iff $G \vDash \neg\text{SLP}(\neg d, r^{-1})$. Suppose that $G \vDash c$. From Lemma 9, $G \vDash \text{WLP}(r,d)$. From Definition 35, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash_{\text{par}} \{c\}\ r\ \{d\}$. Note that r only have one transition, which either transform the initial graph or fail, r can not diverge. Hence, $\vDash_{\text{tot}} \{c\}\ r\ \{d\}$.

Inductive case.
Assume that the implication holds for each premise of inference rules in Definition 42 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 $\vdash_{\text{SEM}} \{c\}\ \mathcal{R}\ \{d\}$ and $G \vDash c$. Since we can have a proof tree where $\{c\}\ \mathcal{R}\ \{d\}$ is the root, then $\vdash_{\text{SEM}} \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From the base case, this means that $\vDash_{\text{tot}} \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From the semantics of graph programs, $H \in [\![\mathcal{R}]\!]G$ 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. Also, the execution of a rule set call only have one transition, which either transform the initial graph or fail so that it can not diverge, so that $\vDash_{\text{tot}} \{c\}\ \mathcal{R}\ \{d\}$.
b) [comp]. Suppose that $\vdash_{\text{SEM}} \{c\}\ P; Q\ \{d\}$ and $G \vDash c$. From the induction hypothesis, $\vdash_{\text{SEM}} \{c\}\ P; Q\ \{d\}$, implies $\vDash_{\text{tot}} \{c\}\ P\ \{e\}$ and $\vdash_{\text{tot}} \{e\}\ Q\ \{d\}$ which means that the execution of P and Q always terminate w.r.t their pre- and postconditions. From $\vDash_{\text{tot}} \{c\}\ P\ \{e\}$, we know that for a graph $G \vDash c$, we have either $\langle P, G\rangle \rightarrow^*$ fail or $\langle P, G\rangle \rightarrow^* G'$ for some graph $G \vDash e$. For the first case, from the semantic of graph programs, $\langle P; Q, G\rangle \rightarrow^*$ fail. For the second case, note that because of $\vdash_{\text{tot}} \{e\}\ Q\ \{d\}$ we know that for $\langle Q, G'\rangle \rightarrow^*$ fail or $\langle Q, G'\rangle \rightarrow^* H$ for some graph $H \vDash d$. The former gives us $\langle P; Q, G\rangle \rightarrow^*$ fail, while the latter gives us $\langle P; Q, G\rangle \rightarrow^* H$ for a graph $H \vDash e$ (which yields $\vDash_{\text{par}} \{c\}\ P; Q\ \{d\}$). Hence, $\vDash_{\text{tot}} \{c\}\ P; Q\ \{d\}$.
c) [cons]. Suppose that $\vdash_{\text{SEM}} \{c\}\ P\ \{d\}$ and $G \vDash c$. From the inference rule and induction hypothesis, we know that $\vdash_{\text{SEM}} \{c'\}\ P\ \{d'\}$, c implies c' (so that $G \vDash c'$), and d' implies d. From induction hypothesis, \$ $\vdash_{\text{SEM}} \{c'\}\ P\ \{d'\}$ implies $\vDash_{\text{tot}} \{c'\}\ P\ \{d'\}$. Note that $G \vDash c$ implies $G \vDash c'$, so that $\vDash_{\text{tot}}$ $\{c'\}\ P\ \{d'\}$ means that $\langle P, G\rangle \rightarrow^*$ fail or $\langle P, G\rangle \rightarrow^* H$ for some $H \vDash d'$. Since d' implies d, $H \vDash d$ as well (so that $\vDash_{\text{par}} \{c\}\ P\ \{d\}$). Hence, $\vDash_{\text{tot}} \{c\}\ P\ \{d\}$.
d) [if]. Suppose that $\vdash_{\text{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$. From $\vdash_{\text{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and induction hypothesis, we get that $\vdash_{\text{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ and $\vdash_{\text{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$ so that $\vdash_{\text{tot}} \{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ and $\vdash_{\text{tot}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From $\vdash_{\text{tot}}$ $\{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ we know that in the case of $G \vDash \text{SUCCESS}(C)$, $\langle P, G\rangle \rightarrow^*$ fail or $\langle P, G\rangle \rightarrow^* H$ for some $H \vDash d$. From the semantics of graph programs, the former means $\langle \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \rightarrow^*$ fail, while the latter means $\langle \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \rightarrow^* H$ for some $H \vDash d$. Then from $\vdash_{\text{tot}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$ we know that in the case $G \vDash \text{FAIL}(C)$, $\langle Q, G\rangle \rightarrow^*$ fail or $\langle Q, G\rangle \rightarrow^* H$ for some $H \vDash d$. From the semantics of graph programs, the former means $\langle \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \rightarrow^*$ fail, while the

latter means $\langle \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \to^* H$ for some $H \vDash d$ (so that $\vDash_{\text{par}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$). Hence, $\vDash_{\text{tot}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.

e) [try]. Similar to argument for [if]:
Suppose that $\vdash_{\text{SEM}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$. From the proof rule and induction hypothesis, $\vdash_{\text{SEM}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ means that $\vdash_{\text{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ C; P\ \{d\}$ and $\vdash_{\text{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$ so that $\vdash_{\text{tot}} \{c \wedge \text{SUCCESS}(C)\}\ C; P\ \{d\}$ and $\vdash_{\text{tot}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From $\vdash_{\text{tot}}$ $\{c \wedge \text{SUCCESS}(C)\}\ C; P\ \{d\}$ we know that in the case of $G \vDash \text{SUCCESS}(C)$, $\langle C; P, G\rangle \to^*$ fail or $\langle C; P, G\rangle \to^* H$ for some $H \vDash d$. From the semantics of graph programs, the former means $\langle \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \to^*$ fail, while the latter means $\langle \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \to^* H$ for some $H \vDash d$. Then from $\vdash_{\text{tot}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$ we know that in the case $G \vDash \text{FAIL}(C)$, $\langle Q, G\rangle \to^*$ fail or $\langle Q, G\rangle \to^* H$ for some $H \vDash d$. From the semantics of graph programs, the former means $\langle \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \to^*$ fail, while the latter means $\langle \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}, G\rangle \to^* H$ for some $H \vDash d$ (so that $\vDash_{\text{par}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$). Hence, $\vDash_{\text{tot}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.

f) [alap]. Suppose that $\vdash_{\text{SEM}} \{c\}\ P!\ \{d\}$ where for all graphs H, $H \vDash d$ iff $H \vDash (c \wedge \text{FAIL}(P)) \vee e$ for some assertion e such that $\text{Break}(c, P, e)$ holds. Suppose that $G \vDash c$. From the proof rule and induction hypothesis, $\vdash_{\text{SEM}} \{c\}\ P!\ \{d\}$ means that $\vdash_{\text{SEM}} \{c\}\ P\ \{c\}$, P is $\#$-decreasing, and $\text{Break}(c, P, e)$ holds, also $\vDash_{\text{tot}} \{c\}\ P\ \{c\}$.
Recall that loop $P!$ can not fail. Now let us consider the case where the execution of $P!$ on G result in a graph H. From the semantics of graph programs, the termination occur if $\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle$, $\langle P, G\rangle \to^+ \texttt{fail}$, or $\langle P!, G\rangle \to \langle P!, G_1\rangle \to \langle P!, G_2\rangle \to \ldots \to \langle P!, G_n\rangle \to H$ for some $n \geq 1$. For the first case, $H \vDash e$ (such that $H \vDash (c \wedge \text{FAIL}(P)) \vee e$) because we know that $\text{Break}(c, P, e)$ holds and $G \vDash c$. For the second case, $H \vDash c$ because $G = H$ and $H \vDash \text{FAIL}(P)$ because fail$\in [\![P]\!]G$ (and $G = H$) so that $H \vDash (c \wedge \text{FAIL}(P)) \vee e$. For the third case, we know from the inference rule [Loop$_1$] that $\langle P, G\rangle \to^+ G_1$ and $\langle P, G_i\rangle \to^+ G_{i+1}$ for all $i = 1, \ldots, n-1$ because $\langle P!, G\rangle \to \langle P!, G_1\rangle \to \langle P!, G_2\rangle \to \ldots \to \langle P!, G_n\rangle$. Because $\vDash_{\text{tot}}$ $\{c\}\ P\ \{c\}$ holds and $G \vDash c$, $G_i \vDash c$ for all $i = 1, \ldots, n$. Because $G_n \vDash c$ and $\langle P, G_n\rangle \to H$, similar with the argument for the first and second case above, $H \vDash (c \wedge \text{FAIL}(P)) \vee e$. Since for all cases the resulting graph will always satisfy d, $\vDash_{\text{par}} \{c\}\ P!\ \{d\}$. Next, we need to show that P will not diverge on G by contradiction.
The non-termination then may appear if there is an infinite sequence $\langle P!, G\rangle \to \langle P!, G_1\rangle \to \ldots$. Recall that from the proof rule we get that P is $\#$-decreasing over c, for some termination function $\#$. By Definition 38, $\langle P, G_1\rangle \to^* G_2$ implies $\#G > \#H$ or $\langle P, G_1\rangle \to^* \langle \texttt{break}, G_2\rangle$. Assume that $P!$ may diverge, so that there is infinitely many executions of P. For each execution, it either yields $\langle \texttt{break}, H\rangle$ (hence contradiction since it is terminating), or results in a graph for which $\#$ returns a smaller number (than $\#$ value of the initial graph). Since there are finitely many smaller natural numbers than any natural number n, we have a contradiction. Hence, $P!$ can not be diverge from G satisfying c. Hence, $\vDash_{\text{tot}} \{c\}\ P!\ \{d\}$.

Theorem 5 (Soundness of SYN). Given graph program P and monadic second-order formulas c, d. Then,

$$\vdash_{\mathrm{SYN}} \{c\}\ P\ \{d\} \text{ implies } \vDash_{\mathrm{tot}} \{c\}\ P\ \{d\}.$$

Proof. The soundness of $[\text{ruleapp}]_{\text{slp}}$ follows from Theorem 1 and Theorem 4, while the soundness of $[\text{ruleapp}]_{\text{wlp}}$ follows from Theorem 1 and Lemma 9. The soundness of [ruleset], [comp], [cons], [if], and [try] follows from Theorem 4 and Theorem 2 about defining SUCCESS and FAIL in monadic second-order formulas. Finally, the soundness of the inference rule [alap] follows from Theorem 4 and Theorem 3.

6.2 Relative completeness

Before showing that SEM is relative complete, we first show that total correctness implies #-decreasing. Also, we show that $\vdash_{\mathrm{SEM}} \{\mathrm{WLP}(P, d)\}\ P\ \{d\}$. Note that $\mathrm{WLP}(P, d)$ must exist because we always have at least one liberal precondition for any graph program, that is: `true`.

Poskitt in his thesis [19] shows that total correctness implies #-decreasing by showing that total correctness implies finiteness. We follow the lemmas and the proofs from the thesis, but with considering the command `break`, which was not considered in Poskit's thesis.

Lemma 10 (total correctness and finiteness). Given a graph program P and assertions c, d. For all host graphs G satisfying c,

$$\vDash_{\mathrm{tot}} \{c\}\ P\ \{c\} \text{ implies } [\![P]\!]G \text{ is finite up to isomorphism.}$$

Proof. Assume that $\vDash_{\mathrm{tot}} \{c\}\ P\ \{c\}$. That is, P does not diverge or get stuck on any graph $G \vDash c$. Now we show by induction that $[\![P]\!]G$ is finite up to isomorphism by induction on graph programs.

Base case.
If $P = \mathcal{R}$ for a rule set $\mathcal{R}$. From the semantics of graph programs, $H \in [\![P]\!]G$ iff $G \Rightarrow_{\mathcal{R}} H$ for some host graph H. We have finitely many rule schema r in $\mathcal{R}$, with finitely many possible matches g for each r because host graphs have finitely many nodes and edges. Also, each application $G \Rightarrow_{r,g} H$ is unique up to isomorphism. Hence, $[\![P]\!]G$ is finite.

Inductive case.
Suppose that for graph programs C, Q, and S, the lemma holds.

1. If $P = Q; S$,
   From the semantics of graph programs, $H \in [\![P]\!]G$ iff $H \in [\![S]\!]G'$ for some $G' \in [\![Q]\!]G$. From induction hypothesis, we know that there are finitely many $G' \in [\![Q]\!]G$ and for each G', there are finitely many $H \in [\![S]\!]G'$. Hence, $[\![P]\!]G$ must also be finite up to isomorphism.

2. If $P =$ `if` C `then` Q `else` S,
   From assumption, we know that C must terminate on G. From the semantics of graph programs, $H \in [\![P]\!]G$ iff $G \vDash$ SUCCESS$C \Rightarrow H \in [\![Q]\!]G$ and $G \vDash$ FAIL$C \Rightarrow H \in [\![S]\!]G$. From the induction hypothesis, both $[\![Q]\!]G$ and $[\![S]\!]G$ are finite up to isomorphism. Hence, $[\![P]\!]G$ must also be finite up to isomorphism.
3. If $P =$ `try` C `then` Q `else` S,
   From assumption, we know that C must terminate on G. From the semantics of graph programs, $H \in [\![P]\!]G$ iff $G \vDash$ SUCCESS$C \Rightarrow H \in [\![C; Q]\!]G$ and $G \vDash$ FAIL$C \Rightarrow H \in [\![S]\!]G$. From the induction hypothesis, $[\![S]\!]G$ is finite up to isomorphism. From point 1, we also know that $[\![C; Q]\!]G$ is also finite up to isomorphism. Hence, $[\![P]\!]G$ must also be finite up to isomorphism.
4. If $P = Q!$,
   From assumption, $Q!$ must terminate on G so that there is no infinite sequence of iteration of Q. Also by assumption, we know that if we have $\langle P!, G_0\rangle \to \langle P!, G_1\rangle \to \ldots \to \langle P!, G_n\rangle$ for some $n \geq 1$ and $G_0 = G$, then $G_i \vDash c$ for all $i = 0, 1, \ldots, n$ so that by induction hypothesis, $[\![Q]\!]G_i$ is finite up to isomorphism. Hence, for every iteration of Q, there are finitely many result graph, and there are finitely many sequence of iteration of Q so that $[\![P]\!]G$ is finite (note that in the case where P terminates with the `break` command, by assumption the finiteness still hold for Q).

Lemma 11. Given a graph program P and assertions c, d. Then,

$$\vDash_{\text{tot}} \{c\}\ P\ \{c\} \text{ and } \vDash_{\text{tot}} \{c\}\ P!\ \{d\} \text{ implies } P \text{ is \#-decreasing under } c$$

for some termination function #.

Proof. From the definition of total correctness, $\vDash_{\text{tot}} \{c\}\ P!\ \{d\}$ implies that for every graph G satisfying c, the execution of $P!$ on G will not diverge or get stuck. This means, there must be a finite sequence of derivations:

$$\langle P!, G_0\rangle \to \langle P!, G_1\rangle \to \ldots \to \langle P!, G_n\rangle \to H$$

for some $H \vDash d$, where $G_0 = G$ and G_i denotes a graph resulting from the i-th iteration of P. Next we define the termination function # by considering the length of finite sequence of derivations.

Here, we use notion that is used in [19]: for $M, N \in \mathcal{G}(\mathcal{L})$, let $M \rightsquigarrow_P N$ denotes a transition $\langle P!, M\rangle \to \langle P!, N\rangle$ that is obtained from one iteration of P ($\langle P, M\rangle \to N$). We consider $\rightsquigarrow_P$ is closed under isomorphism, in the sense that if for graphs M, M', N, N' where $M \cong M'$ and $N \cong N'$, then $M \rightsquigarrow_P N$ implies $M' \rightsquigarrow_P N'$. We then write a relation on isomorphism classes of graphs $[M] \rightsquigarrow_P [N]$ if $M \rightsquigarrow_P N$. If $\vDash_{\text{tot}} \{c\}\ P\ \{c\}$, we know from Lemma 10 that the set $\{[N] \mid [M] \rightsquigarrow_P [N]\}$ is finite.

$\vDash_{\text{tot}} \{c\}\ P!\ \{d\}$ implies that there is no infinite sequence of $\rightsquigarrow_P$-steps starting from [G]. From König's lemma [12] we get that no infinite sequence of $\rightsquigarrow_P$-steps implies the length of $\rightsquigarrow_P$ transitions starting from $[G]$ is bounded because in the tree of all derivations starting from $[G]$, all nodes have finite degree (see proof of Lemma 10 point 4 of inductive case). Hence, the length of $\rightsquigarrow_P$ derivations starting from G is also bounded.

We then define the termination function $\#$ so that for any graph $G \in \mathcal{G}(\mathcal{L})$, $\#M$ is the length of a longest $\rightsquigarrow_P$ sequence starting from M if $M \vDash c$, and $\#M = 0$ otherwise. With this definition, if $M \vDash c$ and $M \rightsquigarrow_P N$, then $\#M > \#N$. Hence, P is $\#$-decreasing under c.

Lemma 12. Given a graph program S and a precondition c. Then,

$$\vDash_{\text{tot}} \{c\}S\{\text{SLP}(c,S)\} \text{ implies } \vdash_{\text{SEM}} \{c\}S\{\text{SLP}(c,S)\}$$

Proof. Let $\vDash_{\text{tot}} \{c\}S\{\text{SLP}(c,S)\}$ is true. By induction on graph programs, we prove that $\vdash_{\text{SEM}} \{c\}S\{\text{SLP}(c,S)\}$.

Base case. If S is a (conditional) rule schema r,
$\vdash_{\text{SEM}} \{c\}S\{\text{SLP}(c,S)\}$ directly follows from the axiom [ruleapp]$_{\text{slp}}$.

Inductive case.
Assume that for graph programs C, P, and Q, $\vdash_{\text{SEM}} \{c\}C\{\text{SLP}(c,S)\}$, $\vdash_{\text{SEM}} \{c\}P\{\text{SLP}(c,S)\}$, and $\vdash_{\text{SEM}} \{c\}Q\{\text{SLP}(c,S)\}$.

(a) If $S = \mathcal{R}$.
If $\mathcal{R} = \{\}$, then there is no premise to prove so that we can deduce $\vdash_{\text{SEM}} \{c\}\mathcal{R}\{\text{SLP}(c,\mathcal{R})\}$ automatically. If $\mathcal{R} = \{r_1,\ldots,r_n\}$ for $n > 0$, $\vdash_{\text{SEM}} \{\{c\}r_1\{\text{SLP}(c,r_1)\},\ldots,\vdash_{\text{SEM}} \{c\}r_n\{\text{SLP}(c,r_n)\}$ from [ruleapp]$_{\text{slp}}$. Let e be the assertion $\text{SLP}(c,r_1) \vee \ldots \vee \text{SLP}(d,r_n)$, so that by [cons], $\vdash_{\text{SEM}} \{c\}\ r_1\ \{e\},\ldots,\vdash_{\text{SEM}} \{c\}\ r_n\ \{e\}$. By [ruleset] we then get that $\vdash_{\text{SEM}} \{c\}\ \mathcal{R}\ \{e\}$. Then by [cons], $\vdash_{\text{SEM}} \{c\}\mathcal{R}\{\text{SLP}(c,\mathcal{R})\}$.

(b) If $S = P; Q$,
From the induction assumption, $\vdash_{\text{SEM}} \{c\}P\{\text{SLP}(c,P)\}$ and $\vdash_{\text{SEM}} \{\text{SLP}(c,P)\}Q\{\text{SLP}(\text{SLP}(c,P),Q)\}$. Then by the inference rule [comp], we get that $\vdash_{\text{SEM}} \{c\}\ P;Q\ \{\text{SLP}(\text{SLP}(c,P),Q)\}$. Finally by [cons], we have $\vdash_{\text{SEM}} \{c\}\ P;Q\ \{\text{SLP}(c,P;Q)\}$.

(c) If $S = \texttt{if}\,C\,\texttt{then}\,P\,\texttt{else}\,Q$,
Both $\vdash_{\text{SEM}} \{c\}\,P\,\{\text{SLP}(c,P)\}$ and $\vdash_{\text{SEM}} \{c\}\,Q\,\{\text{SLP}(c,Q)\}$ follow from the induction assumption. By [cons], we have $\vdash_{\text{SEM}} \{SUCCESS(C) \wedge c\}\,P\,\{e\}$ and $\vdash_{\text{SEM}} \{FAIL(C) \wedge c\}\,Q\,\{e\}$ for $e = \text{SLP}(c,P) \vee \text{SLP}(c,Q)$. Then, by [if], we get that $\vdash_{\text{SEM}} \{c\}\,S\,\{e\}$, which can be deduced to $\vdash_{\text{SEM}} \{c\}\,S\,\{\text{SLP}(c,S)\}$ by [cons].

(d) If $S = \texttt{try}\,C\,\texttt{then}\,P\,\texttt{else}\,Q$,
From the induction assumption, we know that $\vdash_{\text{SEM}} \{c\}\,C\,\{\text{SLP}(c,C)\}$, $\vdash_{\text{SEM}} \{\text{SLP}(c,C)\}\,P\,\{\text{SLP}(\text{SLP}(c,C),P)\}$, and $\vdash_{\text{SEM}} \{c\}\,Q\,\{\text{SLP}(c,Q)\}$. The first two can result on $\vdash_{\text{SEM}} \{c\}\,C;P\,\{\text{SLP}(c,C;P)\}$ by [cons] (see proof point b). By [cons], we have $\vdash_{\text{SEM}} \{SUCCESS(C) \wedge c\}\,C;P\,\{e\}$ and $\vdash_{\text{SEM}} \{FAIL(C) \wedge c\}\,Q\,\{e\}$ for $e = \text{SLP}(c,C;P) \vee \text{SLP}(c,Q)$. Then, by [try], we get that $\vdash_{\text{SEM}} \{c\}\,S\,\{e\}$, which can be deduced to $\vdash_{\text{SEM}} \{c\}\,S\,\{\text{SLP}(c,S)\}$ by [cons].

(e) If $S = P!$,
From the assumption, $\vdash_{\text{SEM}} \{\text{SLP}(c, P!)\}\, P\, \{\text{SLP}(\text{SLP}(c, P!), P)\}$. By [cons], we have $\vdash_{\text{SEM}} \{\text{SLP}(c, P!)\}\, P\, \{\text{SLP}(c, P!; P)\}$ (see proof b). Note that $P!; P$ is not a graph program if P contains the command `break`. Because of the assumption, we know that $P!$ does not diverge on graph satisfying c, such that the only possibilities of $P!; P$ is that for some $G \vDash c$, we have $H \in [\![P!; P]\!]G$ for some $G \vDash c$ iff $H \in [\![P]\!]G'$ for some $G' \in [\![P!]\!]G$. From the semantics of GP 2, we know that $G' \vDash \text{FAIL}(P)$ so that there exists a path to fail if P is executed on G'. In the case where we out from $P!$ due to the non-determinism, executing P again should be result in a graph satisfying $\text{SLP}(c, P!)$ since we can see $P!; P$ as "we try once again if we fail due to the non-determinism". Hence, $\vdash_{\text{SEM}} \{\text{SLP}(c, P!)\}\, P\, \{\text{SLP}(c, P!)\}$ by [cons]. Note that from Theorem 4, this implies $\vDash_{\text{tot}} \{\text{SLP}(c, P!)\}\, P\, \{\text{SLP}(c, P!)\}$ 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 $\langle P, G_n\rangle \to^* \langle \texttt{break}, H\rangle$, $G \vDash \text{SLP}(c, P!)$ implies $G_n \vDash \text{SLP}(c, P!)$ and $H \vDash \text{SLP}(c, P!)$. Hence, $\text{Break}(\text{SLP}(c, P!), P, \text{SLP}(c, P!))$ holds. From the premise of the lemma, we also know that $\vDash_{\text{tot}} \{c\}\ P!\ \{\text{SLP}(c, P!)\}$. From Lemma 11, we get that P is #-decreasing. Then by the rule [alap], we get that $\vdash_{\text{SEM}} \{\text{SLP}(c, P!)\}\, P!\, \{(\text{SLP}(c, P!) \wedge \text{FAIL}(P)) \vee \text{SLP}(c, P!)\}$ such that by [cons], $\vdash_{\text{SEM}} \{c\}\, P!\, \{\text{SLP}(c, P!)\}$.

Theorem 6 (Relative completeness of SEM). Given a graph program P and assertions c, d. Then,

$$\vDash_{\text{tot}} \{c\}\ P\ \{d\} \text{ implies } \vdash_{\text{SEM}} \{c\}\ P\ \{d\}.$$

Proof. From the definition of SLP, we know that $\vDash_{\text{par}} \{\text{c}\}\ P\ \{\text{SLP}(c, P)\}$ is true for any graph program P and assertion d. Also, if $\vDash_{\text{tot}} \{c\}\ P\ \{d\}$ then $\text{SLP}(c, P)$ implies d such that for every graph $G \vDash c$ such that every execution of P on G must be terminate. Hence, $\vDash_{\text{tot}} \{c\}\ P\ \{\text{SLP}(c, P)\}$. If $\vDash_{\text{tot}} \{c\}\ P\ \{\text{SLP}(c, P)\}$ then $\vdash_{\text{SEM}} \{c\}\ P\ \{\text{SLP}(c, P)\}$, so that by [cons] we get that $\vdash_{\text{SEM}} \{c\}\ P\ \{d\}$.

7 Case Studies

In this section, we present two graph programs: `transitive-closure` [16] and `is-connected` [7]. For each graph programs, pre- and postconditions are provided, and we verify the graph programs with respect to the given specifications.

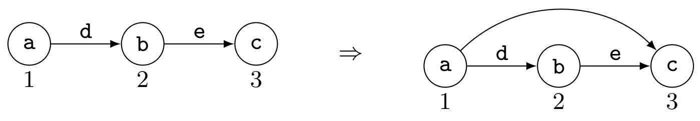
7.1 Transitive closure

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 Fig. 11. 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.

$\texttt{Main} = \texttt{link!}$

$\texttt{link(a, b, c, d, e : list)}$



$\texttt{where not edge(1, 3)}$

Fig. 11: 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.*

Proof tree of `transitive-closure`

The proof of the total correctness of `transitive-closure` is given by the proof tree of Fig. 12, while the assertions that are used in the proof tree are defined in Table 7. Note that we do not give #-decreasing function in the proof tree due to the space limit.

Now let us consider loops we have in the program `transitive-closure`: `link!`. Let $\#(G)$ be the sum of $i(v)$ for all $v \in V_G$, where $i(v)$ denotes the number of nodes x such that there is no edge from v to x. Since `link` always add an edge from a node u to another node w where initially there were no edge from u to w, so clearly that `link` is #-decreasing.

Proof of implications

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.

$$
[\text{cons}]\,\dfrac{[\text{alap}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{\text{pre}\}\ \texttt{link}\ \{\text{Slp}(\text{pre},\texttt{link})\}}}{\{\text{pre}\}\ \texttt{link}\ \{\text{pre}\}}}{\{\text{pre}\}\ \texttt{link!}\ \{\text{pre}\wedge\text{Fail}(\texttt{link})\}}}{\{\text{pre}\}\ \texttt{link!}\ \{\text{post}\}}
$$

Fig. 12: Proof tree for partial correctness of ttttransitive-closure

Table 7: Conditions inside proof tree of `transitive-closure`

MSO formulas
pre $\equiv \forall_{\mathsf{v}}\mathsf{x}(\mathsf{m_V(x) = none} \wedge \neg\mathsf{root(x)}) \wedge \forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_E(x) = none})$
post $\equiv$ pre $\wedge\forall_{\mathsf{v}}\mathsf{x,y}(\mathsf{x \neq y} \wedge \mathsf{path(x,y)} \Rightarrow \mathsf{edge(x,y)})$
Fail(`link`) $\equiv \neg\exists_{\mathsf{v}}\mathsf{x,y,z}(\mathsf{x \neq y} \wedge \mathsf{x \neq z} \wedge \mathsf{z \neq y} \wedge \mathsf{edge(x,y,none)} \wedge \mathsf{edge(y,z,none)} \wedge \neg\mathsf{edge(x,z)})$
Slp(pre,`link`)$\equiv$ $\exists_{\mathsf{v}}\mathsf{u,v,w}(\mathsf{edge(u,v,none)} \wedge \mathsf{edge(v,w,none)} \wedge \mathsf{edge(u,w,none)}$ $\wedge\ \mathsf{m_V(u) = none} \wedge \mathsf{m_V(v) = none} \wedge \mathsf{m_V(w) = none} \wedge \neg\mathsf{root(u)} \wedge \neg\mathsf{root(v)} \wedge \neg\mathsf{root(w)}$ $\wedge\ \forall_{\mathsf{v}}\mathsf{x}(\mathsf{x = u} \vee \mathsf{x = v} \vee \mathsf{x = w} \vee (\mathsf{m_V(x) = none} \wedge \neg\mathsf{root(x)}))$ $\wedge\ \exists_{\mathsf{e}}\mathsf{a,b,c}(\mathsf{s(a) = u} \wedge \mathsf{t(a) = v} \wedge \mathsf{s(b) = v} \wedge \mathsf{t(b) = w} \wedge \mathsf{s(c) = u} \wedge \mathsf{t(c) = w}$ $\wedge\ \forall_{\mathsf{e}}\mathsf{y}(\mathsf{y = a} \vee \mathsf{y = b} \vee \mathsf{y = c} \vee (\mathsf{s(y) \neq u} \wedge \mathsf{t(y) \neq w})))$ $\wedge\forall_{\mathsf{e}}\mathsf{x}(\mathsf{m_E(x) = none})$

1. *Proof of* Slp(pre,`link`) *implies* pre
   Let us consider the subformula of Slp(pre,`link`):
   $\mathsf{m_V(u) = none} \wedge \mathsf{m_V(v) = none} \wedge \mathsf{m_V(w) = none} \wedge \neg\mathsf{root(u)} \wedge \neg\mathsf{root(v)} \wedge \neg\mathsf{root(w)}$
   $\wedge\forall_{\mathsf{v}}\mathsf{x}(\mathsf{x = u} \vee \mathsf{x = v} \vee \mathsf{x = w} \vee (\mathsf{m_V(x) = none} \wedge \neg\mathsf{root(x)}))$.
   From $\mathsf{x = u}$ and $\mathsf{m_V(u) = none} \wedge \neg\mathsf{root(u)}$, $\mathsf{x = v}$ and $\mathsf{m_V(v) = none} \wedge \neg\mathsf{root(v)}$, also $\mathsf{x = w}$ and $\mathsf{m_V(w) = none} \wedge \neg\mathsf{root(w)}$, we can say that the subformula above implies
   $\forall_{\mathsf{v}}\mathsf{x}((\mathsf{m_V(u) = none} \wedge \neg\mathsf{root(u)})$
   $\quad\vee (\mathsf{m_V(v) = none} \wedge \neg\mathsf{root(v)})$
   $\quad\vee (\mathsf{m_V(w) = none} \wedge \neg\mathsf{root(w)})$
   $\quad\vee (\mathsf{m_V(x) = none} \wedge \neg\mathsf{root(x)}))$,
   which clearly implies $\forall_{\mathsf{v}}\mathsf{x}(\mathsf{m_V(x) = none} \wedge \neg\mathsf{root(x)})$. Hence, Slp(pre,`link`) implies pre.
2. *Proof of* pre$\wedge$Fail(`link`) *implies* c
   Note that pre$\wedge$Fail(`link`) implies
   pre$\wedge\forall_{\mathsf{v}}\mathsf{x,y,z}(\mathsf{x \neq y} \wedge \mathsf{x \neq z} \wedge \mathsf{y \neq z} \wedge \mathsf{edge(x,y)} \wedge \mathsf{edge(y,z)} \Rightarrow \mathsf{edge(x,z)})$.
   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 $\forall_{\mathsf{v}}\mathsf{x,y}(\mathsf{x \neq y} \wedge \mathsf{path(x,y)} \Rightarrow \mathsf{edge(x,y)})$.
   From Lemma 1, we know that the predicate $\mathsf{x \neq y} \wedge \mathsf{path(x,y)}$ can be expressed by:
   $\exists_{\mathsf{E}}\mathsf{X}(\exists_{\mathsf{e}}\mathsf{u,v}(\mathsf{u \in X} \wedge \mathsf{v \in X} \wedge \mathsf{s(u) = x} \wedge \mathsf{t(v) = y})$
   $\qquad\wedge\ \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u \in X} \Rightarrow (\mathsf{t(u) = y} \vee \exists_{\mathsf{e}}\mathsf{v}(\mathsf{v \in X} \wedge \mathsf{s(v) = t(u)}))),$
   which is equivalent to $\exists_{\mathsf{l}}\mathsf{n}(\mathsf{P(x,y,n)})$, where $\mathsf{P(x,y,n)}$ is the formula:

$\exists_{\mathsf{V}}\mathsf{X}(\mathsf{card}(\mathsf{X}) = \mathsf{n} \wedge \exists_{\mathsf{e}}\mathsf{u},\mathsf{v}(\mathsf{u} \in \mathsf{X} \wedge \mathsf{v} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{u}) = \mathsf{x} \wedge \mathsf{t}(\mathsf{v}) = \mathsf{y})$
$\quad \wedge \forall_{\mathsf{e}}\mathsf{u}(\mathsf{u} \in \mathsf{X} \Rightarrow (\mathsf{t}(\mathsf{u}) = \mathsf{y} \vee \exists_{\mathsf{e}}\mathsf{v}(\mathsf{v} \in \mathsf{X} \wedge \mathsf{s}(\mathsf{v}) = \mathsf{t}(\mathsf{u})))$.
From the meaning of implication, we know that $\forall_{\mathsf{v}}\mathsf{x},\mathsf{y}(\exists_{\mathsf{l}}\mathsf{n}(\mathsf{P}(\mathsf{x},\mathsf{y},\mathsf{n})) \Rightarrow \mathsf{edge}(\mathsf{x},\mathsf{y}))$ 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.

7.2 Connectedness

Graph program `connectedness`

Here we consider the graph program `is-connected` as seen in Fig. 13. 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.

For the specification, here we consider the case where the input graph is connected. For the case with disconnected graph, please see [25].

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

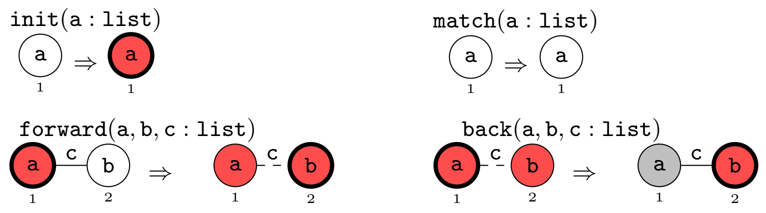


Fig. 13: Graph program `is-connected`

Precondition: *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 undirect path from x to y)*
Postcondition: *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.*

Proof tree of `is-connected`

The total correctness proof for this case study is presented in the proof tree of Fig. 14, where assertions used in the proof tree are defined in Table 8. Note that we do not write the #-decreasing requirement in the premise of proof rule [alap] we use due to the space limit.

Now let us consider loops we have in the program `is-connected`. There are two loops: `forward!` and `DFS!`. For the former, we can consider #-function that count the number of unmarked nodes. By the application of the rule schema `forward`, the number of unmarked nodes obviously decreasing. Hence `forward` is #-decreasing. For `DFS!`, we can consider a #-function that count unmarked nodes and red nodes. From the initial graph, the application of `forward!` will not change the value of #, while `try` either will decrease the value of # by 1 or make us reach `break`. Hence, `DFS` is #-decreasing as well.

From the proof tree we know the triple $\{pre\}$ `init` $\{c\}$ and $\{c\}$ `DFS!` $\{post\}$ are totally correct so that by the proof rule [comp] we can conclude that $\{pre\}$ `init`; `DFS!` $\{post\}$ is totally correct as well. Implication $post \Rightarrow \neg\text{Fail}(\texttt{match})$ must be true because the postcondition assert that there is no unmarked node. Hence, we can conclude that the execution of the program on a graph satisfying Precondition cannot fail and must resulting a graph satisfying Postcondition.

$$\text{[try]}\ \dfrac{\text{Subtree A} \qquad \text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ \mathit{pre} \wedge \mathrm{Fail}(\mathtt{init})\}\ \mathtt{skip}\ \{\mathit{pre} \wedge \mathrm{Fail}(\mathtt{init})\}}}{\{\ \mathit{pre} \wedge \mathrm{Fail}(\mathtt{init})\}\ \mathtt{skip}\ \{\mathit{post}\}}}{\{\ \mathit{pre}\ \}\ \mathtt{try\ init\ then}\ (\mathtt{DFS!}; \mathtt{Check})\ \{\ \mathit{post}\ \}}$$

where subtree A is:

$$\text{[cons]}\ \dfrac{\text{comp}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ \mathit{pre}\ \}\ \mathtt{init}\ \{\ \mathrm{Slp}(\mathit{pre}, \mathtt{init})\ \}}}{\{\ \mathit{pre}\ \}\ \mathtt{init}\ \{\ c\ \}} \qquad \text{[cons]}\ \dfrac{\text{[alap]}\ \dfrac{\text{[comp]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[alap]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ c\ \}\ \mathtt{forward}\ \{\ \mathrm{Slp}(c, \mathtt{forward})\ \}}}{\{\ c\ \}\ \mathtt{forward}\ \{\ c\ \}}}{\{\ c\ \}\ \mathtt{forward!}\ \{\ c \wedge \mathrm{Fail}(\mathtt{forward})\ \}}}{\{\ c\ \}\ \mathtt{forward!}\ \{\ d\ \}} \qquad \text{Subtree A1}}{\{\ c\ \}\ \mathtt{DFS}\ \{\ c\ \}} \qquad \mathrm{Break}(c, \mathtt{DFS}, \mathit{post})}{\{\ c\ \}\ \mathtt{DFS!}\ \{\ (c \wedge \mathrm{Fail}(\mathtt{DFS})) \vee \mathit{post}\ \}}}{\{\ c\ \}\ \mathtt{DFS!}\ \{\ \mathit{post}\ \}} \qquad \text{Subtree A2}}{\{\ \mathit{pre}\ \}\ \mathtt{init}; \mathtt{DFS!}; \mathtt{Check}\ \{\ \mathit{post}\ \}}}{\{\ \mathit{pre} \wedge \mathrm{Success}(\mathtt{init})\ \}\ \mathtt{init}; \mathtt{DFS!}; \mathtt{Check}\ \{\ \mathit{post}\ \}}$$

for Subtree A1:

$$\text{[try]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[ruleapp]}_{slp}\ \dfrac{}{\{\ d \wedge \mathrm{Success}(\mathtt{back})\ \}\ \mathtt{back}\ \{\ \mathrm{Slp}(d \wedge \mathrm{Success}(\mathtt{back}), \mathtt{back})\ \}}}{\{\ d \wedge \mathrm{Success}(\mathtt{back})\ \}\ \mathtt{back}\ \{\ c\ \}} \qquad \text{[cons]}\ \dfrac{\text{[break]}\ \dfrac{}{\{\ d \wedge \mathrm{Fail}(\mathtt{back})\ \}\ \mathtt{break}\ \{\ d \wedge \mathrm{Fail}(\mathtt{back})\ \}}}{\{\ d \wedge \mathrm{Fail}(\mathtt{back})\ \}\ \mathtt{break}\ \{\ c\ \}}}{\{\ d\ \}\ \mathtt{try\ back\ else\ break}\ \{\ c\ \}}$$

and Subtree A2:

$$\text{[if]}\ \dfrac{\text{[cons]}\ \dfrac{\text{[fail]}\ \dfrac{}{\{\ \mathsf{false}\ \}\ \mathtt{fail}\ \{\ \mathsf{false}\ \}}}{\{\ \mathit{post} \wedge \mathrm{Success}(\mathtt{match})\ \}\ \mathtt{fail}\ \{\ \mathrm{post}\ \}} \qquad \text{[cons]}\ \dfrac{\text{[skip]}\ \dfrac{}{\{\ \mathit{post} \wedge \mathrm{Fail}(\mathtt{match})\ \}\ \mathtt{skip}\ \{\ \mathit{post} \wedge \mathrm{Fail}(\mathtt{match})\ \}}}{\{\ \mathit{post} \wedge \mathrm{Fail}(\mathtt{match})\ \}\ \mathtt{skip}\ \{\ \mathit{post}\ \}}}{\{\ \mathit{post}\ \}\ \mathtt{if\ match\ then\ fail}\ \{\ \mathit{post}\ \}}$$

Fig. 14: Proof tree for partial correctness of `is-connected`

Table 8: Conditions inside proof tree of `is-connected`

MSO formulas
$pre \equiv \forall_v x(m_V(x) = \mathsf{none} \land \neg \mathsf{root}(x)) \land \forall_E x(m_E(x) = \mathsf{none})$ $\land \forall_V X(\forall_v x(x \in X) \lor \forall_v x(x \notin X) \lor \exists_v x, y(x \in X \land y \notin X \land \mathsf{adj}(x, y)))$
$post \equiv \forall_v x(\mathsf{false})$ $\lor (\exists_v x(m_V(x) = \mathsf{red} \land \mathsf{root}(x) \land \forall_v y(y = x \lor (m_V(y) = \mathsf{grey} \land \neg \mathsf{root}(y)))) \land \forall_e x(m_E(x) = \mathsf{none})$ $\land \forall_V X(\forall_v x(x \in X) \lor \forall_v x(x \notin X) \lor \exists_v x, y(x \in X \land y \in X \land \mathsf{adj}(x, y)))$
$c \equiv$ $\forall_e x(m_E(x) = \mathsf{none} \lor m_E(x) = \mathsf{dashed})$ $\land \exists_v x(\mathsf{root}(x) \land m_V(x) = \mathsf{red}$ $\land \forall_v y(x = y \lor ((m_V(y) = \mathsf{none} \lor m_V(y) = \mathsf{red} \lor m_V(y) = \mathsf{grey}) \land \neg \mathsf{root}(y)))$ $\land \forall_V X(\forall_v x(x \in X) \lor \forall_v x(x \notin X) \lor \exists_v x, y(x \in X \land y \notin X \land \mathsf{adj}(x, y)))$ $\land \forall_e x(m_E(x) = \mathsf{dashed} \Rightarrow m_V(s(x)) = \mathsf{red} \land m_V(t(x)) = \mathsf{red} \land (\neg \mathsf{root}(s(x)) \lor \neg \mathsf{root}(t(x))))$ $\land \forall_v z(m_V(z) = \mathsf{red} \land \neg \mathsf{root}(z) \Rightarrow \mathrm{Success}(\mathrm{back}))$ $\land \forall_v x(m_V(x) = \mathsf{grey} \Rightarrow \neg \exists_v y(\mathsf{adj}(x, y) \land \neg \mathsf{root}(y) \land (m_V(y) = \mathsf{none} \lor (m_V(y) = \mathsf{red} \land \neg \mathsf{root}(y)))))$
$d \equiv c \land \mathrm{Fail}(\texttt{forward})$
$\mathrm{Success}(\texttt{back}) \equiv$ $\exists_v x, y(m_V(x) = \mathsf{red} \land \mathsf{root}(x) \land m_V(y) = \mathsf{red} \land \neg \mathsf{root}(y) \land (\mathsf{edge}(x, y, \mathsf{dashed}) \lor \mathsf{edge}(y, x, \mathsf{dashed})))$
$\mathrm{Fail}(\texttt{back}) \equiv \neg \mathrm{Success}(\texttt{back})$
$\mathrm{Fail}(\texttt{init}) \equiv \neg \exists_v (m_V(x) = \mathsf{none} \land \neg \mathsf{root}(x))$
$\mathrm{Fail}(\texttt{match}) \equiv \mathrm{Fail}(\texttt{init})$
$\mathrm{Success}(\texttt{match}) \equiv \neg \mathrm{Fail}(\texttt{init})$
$\mathrm{Fail}(\texttt{DFS}) \equiv \mathsf{false}$
$\mathrm{Fail}(\texttt{forward}) \equiv$ $\neg \exists_v x, y(m_V(x) = \mathsf{red} \land \mathsf{root}(x) \land m_V(y) = \mathsf{none} \land \neg \mathsf{root}(y) \land (\mathsf{edge}(x, y, \mathsf{none}) \lor \mathsf{edge}(y, x, \mathsf{none})))$
$\mathrm{Slp}(pre, \texttt{init}) \equiv$ $\exists_v y(m_V(y) = \mathsf{red} \land \mathsf{root}(y) \land \forall_v x(x = y \lor (m_V(x) = \mathsf{none} \land \neg \mathsf{root}(x))))$ $\land \forall_e x(m_E(x) = \mathsf{none})$ $\land \exists_v z(\forall_v x(x = z \lor (\mathsf{upath}(x, z) \land \forall_v y(y = z \lor x = y \lor \mathsf{upath}(x, y)))))$
$\mathrm{Slp}(c, \texttt{forward}) \equiv$ $\exists_v a, b(\exists_e c(a \neq b \land m_V(a) = \mathsf{grey} \land m_V(b) = \mathsf{red} \land \neg \mathsf{root}(a) \land \mathsf{root}(b)$ $\land (s(c) = a \land (t(c) = b) \lor (s(c) = b \land t(c) = a)) \land m_V(c) = \mathsf{none}$ $\land f))$
$f \equiv$ $\forall_e x(x \neq c \Rightarrow m_E(x) = \mathsf{none} \lor m_E(x) = \mathsf{dashed})$ $\land \forall_v x(x \neq a \land x \neq b \Rightarrow \neg \mathsf{root}(x) \land (m_V(x) = \mathsf{red} \lor m_V(x) = \mathsf{grey} \lor m_V(x) = \mathsf{none}))$ $\land \neg \exists_V X(((a \in X \land b \in X) \lor (a \notin X \land b \notin X))$ $\land (a \in X \land b \in X \Rightarrow \exists_v x(x \notin X) \land \neg \exists_v x(x \notin X \land (\mathsf{adj}(a, x) \lor \mathsf{adj}(b, x)))$ $\land \neg \exists_v x, y(x \neq a \land x \neq b \land y \neq a \land y \neq b \land x \notin X \land (\mathsf{adj}(x, y))))$ $\land (a \notin X \land b \notin X \Rightarrow \exists_v x(x \in X) \land \neg \exists_v x(x \in X \land (\mathsf{adj}(a, x) \lor \mathsf{adj}(b, x)))$ $\land \neg \exists_v x, y(x \neq a \land x \neq b \land y \neq a \land y \neq b \land x \notin X \land (\mathsf{adj}(x, y)))))$ $\land \forall_e x(x \neq c \land m_E(x) = \mathsf{dashed} \Rightarrow m_V(s(x)) = \mathsf{red} \land m_V(t(x)) = \mathsf{red} \land (\neg \mathsf{root}(s(x)) \lor \neg \mathsf{root}(t(x))))$ $\land \forall_v z(z \neq a \land z \neq b \land m_V(z) = \mathsf{red} \land \neg \mathsf{root}(z) \Rightarrow \mathrm{Success}(\mathrm{back}))$ $\land \forall_v x(x \neq a \land x \neq b \land m_V(x) = \mathsf{grey}$ $\Rightarrow \neg \exists_v y(\mathsf{adj}(x, y) \land \neg \mathsf{root}(y) \land (m_V(y) = \mathsf{none} \lor (m_V(y) = \mathsf{red} \land \neg \mathsf{root}(y)))))$
$\mathrm{Slp}(d \land \mathrm{Success}(back), \texttt{back}) \equiv$ $\exists_v a, b(\exists_e c(a \neq b \land m_V(a) = \mathsf{grey} \land m_V(b) = \mathsf{red} \land \neg \mathsf{root}(a) \land \mathsf{root}(b)$ $\land (s(c) = a \land (t(c) = b) \lor (s(c) = b \land t(c) = a)) \land m_V(c) = \mathsf{none}$ $\land f$ $\land \neg \exists_v y(y \neq a \land y \neq b \land m_V(y) = \mathsf{none} \land \neg \mathsf{root}(y) \land \mathsf{adj}(b, y, \mathsf{none}))$ $\land \neg \exists_v x, y(x \neq a \land x \neq b \land y \neq a \land y \neq b$ $\land m_V(x) = \mathsf{red} \land \mathsf{root}(x) \land m_V(y) = \mathsf{none} \land \neg \mathsf{root}(y) \land \mathsf{adj}(b, y, \mathsf{none}))))$

Proof of implications

1. *Proof of* $pre \wedge \mathrm{Fail}(\texttt{init})$ *implies post*
   From the meaning of conjunction, pre implies $\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))$, while $\mathrm{Fail}(\texttt{init})$ implies $\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) \neq \mathsf{none} \vee \mathsf{root}(\mathsf{x}))$. From the meaning of universal quantifiers, we know that
   $\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))\wedge\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) \neq \mathsf{none} \vee \mathsf{root}(\mathsf{x}))$ implies $\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}) \wedge (\mathsf{m_V}(\mathsf{x})$
   which implies $\forall_\mathsf{v}\mathsf{x}(\mathsf{false})$. Hence, it implies *post* from the meaning of disjunction.
2. *Proof of* $\mathrm{Slp}(pre, \texttt{init})$ *implies* c
   From the meaning of disjunction, the formula
   $\exists_\mathsf{v}\mathsf{y}(\mathsf{m_V}(\mathsf{y}) = \mathsf{red} \wedge \mathsf{root}(\mathsf{y}) \wedge \forall_\mathsf{v}\mathsf{x}(\mathsf{x} = \mathsf{y} \vee (\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))))$ implies
   $\exists_\mathsf{v}\mathsf{x}(\mathsf{root}(\mathsf{x}) \wedge \mathsf{m_V}(\mathsf{x}) = \mathsf{red} \wedge \forall_\mathsf{v}\mathsf{y}(\mathsf{x} = \mathsf{y} \vee \neg\mathsf{root}(\mathsf{y})))$. The formula also implies
   $\forall_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \vee \mathsf{m_V}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m_V}(\mathsf{x}) = \mathsf{grey})$ because from the formula we know that the node represented by y is red and others are unmarked. The formula
   $\forall_\mathsf{e}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{none})$ clearly implies $\forall_\mathsf{e}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{none}) \vee \mathsf{m_E}(\mathsf{x}) = \mathsf{dashed}$, while the formula $\forall_\mathsf{v}\mathsf{x}(\mathsf{x} = 1 \vee (\mathsf{upath}(\mathsf{x}, 1) \wedge \forall_\mathsf{v}\mathsf{y}(\mathsf{y} = 1 \vee \mathsf{x} = \mathsf{y} \vee \mathsf{upath}(\mathsf{x}, \mathsf{y}))))$ implies
   $\forall_\mathsf{v}\mathsf{x}, \mathsf{y}(\mathsf{x} = \mathsf{y} \vee \mathsf{upath}(\mathsf{x}, \mathsf{y}))$.
   Also, if $\mathrm{Slp}(pre, \texttt{init})$ is true then the implications with
   $\forall_\mathsf{e}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{dashed}$ or $(\exists_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{red} \wedge \neg\mathsf{root}(\mathsf{x}))$ as premise are always true because $\mathrm{Slp}(pre, \texttt{init})$ implies $\forall_\mathsf{e}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{none})$ which means that all edges are unmarked, and
   $\exists_\mathsf{v}\mathsf{y}(\mathsf{m_V}(\mathsf{y}) = \mathsf{red} \wedge \mathsf{root}(\mathsf{y}) \wedge \forall_\mathsf{v}\mathsf{x}(\mathsf{x} = \mathsf{y} \vee (\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x}))))$ which means there is exactly one red rooted node while other nodes are unmarked and unrooted.
   Finally, $\mathrm{Slp}(pre, \texttt{init})$ 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
   $\exists_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{unmarked}) \Rightarrow \exists_\mathsf{v}\mathsf{x}, \mathsf{y}(\mathsf{m_V}(\mathsf{x}) = \mathsf{none} \wedge \mathsf{m_V}(\mathsf{y}) = \mathsf{red} \wedge \mathsf{adj}(\mathsf{x}, \mathsf{y})$ must hold.
3. *Proof of* $\mathrm{Break}(c, \texttt{DFS}, post)$
   From the semantics of GP 2 commands, when we have the derivation $[\![DFS, G]\!] \to^*$ $[\![\texttt{break}, H]\!]$ iff we have the derivation $[\![\texttt{forward!}, G]\!] \to^H$ for a graph H such that $\mathrm{Fail}(\texttt{back})$ holds on H. From the proof tree of Fig. 14, we know that $\{c\}$ `forward!` $\{c\wedge\mathrm{Fail}(\texttt{forward})\}$ is correct so that H must imply $c\wedge\mathrm{Fail}(\texttt{forward})$ as well. Hence, H must satisfy $c\wedge\mathrm{Fail}(\texttt{forward})\wedge\mathrm{Fail}(\texttt{back})$ if the input graph G satisfies c.
   Let us consider the formula $\exists_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{red} \wedge \neg\mathsf{root}(\mathsf{x})) \Rightarrow\mathrm{Success}(\texttt{back})$ of c. If
   $c\wedge\mathrm{Fail}(\texttt{forward})\wedge\mathrm{Fail}(\texttt{back})$ holds, then $\mathrm{Success}(\texttt{back})$ must not hold so that from the meaning of implication, we know that $\exists_\mathsf{v}\mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{red} \wedge \neg\mathsf{root}(\mathsf{x}))$ does not hold either. By the meaning of implication and from the formula
   $\forall_\mathsf{e}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{dashed} \Rightarrow (\mathsf{m_V}(\mathsf{s}(\mathsf{x})) = \mathsf{red} \wedge \neg\mathsf{root}) \vee (\mathsf{m_V}(\mathsf{t}(\mathsf{x})) = \mathsf{red} \wedge \neg\mathsf{root}))$ 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
$\forall_v z(\mathsf{m_V}(z) = \mathsf{red} \wedge \neg\mathsf{root}(z) \Rightarrow$ Success(`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, $\mathrm{Slp}(c, \texttt{forward})$ implies
$\forall_{\mathsf{v}} \mathsf{x}(\mathsf{m_V}(\mathsf{x}) = \mathsf{grey} \Rightarrow \neg\exists_{\mathsf{v}} \mathsf{y}(\mathsf{adj}(\mathsf{x}, \mathsf{y}) \wedge \neg\mathsf{root}(\mathsf{y}) \wedge (\mathsf{m_V}(\mathsf{y}) = \mathsf{none} \vee (\mathsf{m_V}(\mathsf{y}) = \mathsf{red} \wedge \neg\mathsf{root}(\mathsf{y})))))$

5. *Proof of* $\mathrm{Slp}(d \wedge \textbf{Success}(\texttt{back}), \texttt{back})$ *implies* c
From the meaning of conjunction, we know that $\mathrm{Slp}(d \wedge \textbf{Success}(\texttt{back}), \texttt{back})$ 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 $\mathrm{Slp}(d \wedge \textbf{Success}(\texttt{back}), \texttt{back})$ implies c.
6. *Proof of post* $\wedge$ Success($\texttt{match}$) *implies* false
post clearly implies that all nodes are either red or grey, so there must not exist an unmarked node.

8 Related Work

Poskitt and Plump [21] have defined a calculus to verify graph programs by using a so-called E-conditions [19] and M-conditions [22] as assertions. E-conditions are only able to express FO properties of GP 2 graph, while M-conditions can express properties of MSO properties of non-attributed graph (not all GP 2 graphs). However, there are only limited graph programs that can be verified by the calculus (e.g. programs with no nested loop).

E-condition is an extension of nested graph conditions [8]. Pennemann [15] shows how to obtain a weakest liberal precondition (wlp) w.r.t a graph condition and a program and introduced a theorem prover to prove implication between a precondition and the obtained wlp. However, graph conditions also only able to express FO properties of a non-attributed graph. Habel and Radke [10] then introduced HR* conditions, which extend the graph conditions by introducing graph variables that represent graphs generated by hyperedge-replacement systems. Radke [23] showed that HR* conditions is somewhere between node-counting MSO graph formulas and SO graph formulas and showed how to construct a wlp w.r.t the conditions. However, theorem prover for this condition is not available yet, and we believe that having a wlp alone is not enough for program verifications.

9 Conclusion and Future Work

In this paper, we have defined MSO formulas that can express local properties of GP 2 graphs, even properties that can not be expressed in counting MSO graph formulas [4]. By using the MSO formulas as assertions, we show that we can construct a strongest liberal postcondition (Slp) over a rule schema. Moreover, we also can use the construction to obtain Slp over a loop-free program, precondition Success(P) (or Fail(P)) that asserts the existence of a proper graph (or path to failure) in the execution of loop-free program P (or iteration command S). With this result, we can define a proof calculus to verify total correctness of graph programs with nested loops in certain forms.

As usual for Hoare calculi, our calculus does not cover implications between assertions. Currently, we have started to experiment of the use of SMT solver Z3 [2] to prove the implication.

References

1. C. Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015.
2. N. Bjørner, L. de Moura, L. Nachmanson, and C. M. Wintersteiger. Programming Z3. In *SETSS 2018*, volume 11430 of *LNCS*, pages 148–201. Springer, 2018.
3. G. Campbell. Efficient graph rewriting. *CoRR*, abs/1906.05170, 2019.
4. B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach.* Cambridge University Press, New York, NY, USA, 1st edition, 2012.
5. P. Cousot. Chapter 15 - methods and logics for proving programs. In J. V. Leeuwen, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 841 – 993. Elsevier, Amsterdam, 1990.
6. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer, 1990.
7. B. C. Graham Campbell and D. Plump. Fast rule-based graph programs. Technical report, University of York, 2020.
8. A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19:245–296, 2009.
9. A. Habel and D. 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.
10. A. Habel and H. Radke. Expressiveness of graph conditions with variables. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 30, 2010.
11. I. Hristakiev and D. 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.
12. D. König. Sur les correspondances multivoques des ensembles. *Fundamenta Mathematicae*, 38:114–134, 1926.
13. T. Nipkow. *Hoare Logics in Isabelle/HOL*, pages 341–367. Springer Netherlands, Dordrecht, 2002.
14. C. Paulin-Mohring. Introduction to the Coq proof-assistant for practical software verification. In *Tools for Practical Software Verification, LASER, International Summer School, Revised Tutorial Lectures*, volume 7682 of *LNCS*, pages 45–95. Springer, 2011.
15. K.-H. Pennemann. *Development of Correct Graph Transformation Systems.* PhD thesis, Department of Computing Science, University of Oldenburg, 2009.
16. D. Plump. The graph programming language GP. In *Proc. International Conference on Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer, 2009.
17. D. 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.

18. D. Plump and C. Bak. Rooted graph programs. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 54, 2012.
19. C. M. Poskitt. *Verification of Graph Programs.* PhD thesis, The University of York, 2013.
20. C. M. Poskitt and D. 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.
21. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
22. C. M. Poskitt and D. 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.
23. H. Radke. *A Theory of HR* Graph Conditions and their Application to Meta-Modeling.* PhD thesis, University of Oldenburg, Germany, 2016.
24. G. Wulandari and D. Plump. Verifying graph programs with first-order logic (extended version). *ArXiv e-prints*, arXiv:2010.14549 [cs.LO], 2020.
25. G. S. Wulandari. Verification of graph programs with monadic second-order logic. Submitted PhD thesis, 2021.