Verifying Graph Programs with First-Order Logic

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. We consider Hoare-style verification for the graph programming language GP 2. In previous work, graph properties were specified by so-called E-conditions which extend nested graph conditions. However, this type of assertions is not easy to comprehend by programmers that are used to formal specifications in standard first-order logic. In this paper, we present an approach to verify GP 2 programs with a standard first-order logic. We show how to construct a strongest liberal postcondition with respect to a rule schema and a precondition. We then extend this construction to obtain strongest liberal postconditions for arbitrary loop-free programs. Compared with previous work, this allows to reason about a vastly generalised class of graph programs. In particular, many programs with nested loops can be verified with the new calculus.

1 Introduction

Reasoning about graph programs in the language GP 2 by using Hoare-style proof systems has been done by Poskitt and Plump [18, 17]. For assertions, they introduced E-constraints which extend nested graph conditions[14] with support for expressions. However, E-constraints may be not easy to understand by general reader, especially for someone not familiar with nested conditions. For example, if we want to express "every node is labelled by an atom" by E-constraints, it is expressed by $\forall(\,_1\text{a}, \exists(\,_1\text{a} \mid \mathtt{atom(a)})) \vee \forall(\,_1\text{a}, \exists(\,_1\text{a} \mid \mathtt{atom(a)})) \vee \forall(\,_1\text{a}, \exists(\,_1\text{a} \mid \mathtt{atom(a)})) \vee \forall(\,_1\text{a}, \exists(\,_1\text{a} \mid \mathtt{atom(a)})) \vee \forall(\,_1\text{a}, \exists(\,_1\text{a} \mid \mathtt{atom(a)}))$. The existence of two quantifiers to express a universal properties may seems unnatural because in standard logic we usually only need to have one quantifier. In addition, the universal properties must be stated for each mark resulting a long condition. In first-order formula we introduce in this paper, the above condition is simply written as $\forall_{\mathrm{V}}\mathrm{x}(\mathrm{atom}(\mathrm{l_V}(\mathrm{x})))$.

In this paper, we present the verification of a graph program by using another type of assertions, that is standard logic. By using standard logic, we aim to have a more practical approach of graph program verification in the sense that in the future we can use existing tools such as the proof assistant Isabelle [13] to minimise human error in verification. Moreover, standard logic should be easier to understand by general readers. There is also a lot of literature about standard

* Supported by Indonesia Endowment Fund for Education (LPDP)

logic, especially in monadic second-order logic. By expressing graph properties in standard logic, there are possibilities to extend our approach based on literature to overcome some limitation we may have.

We introduce first-order formulas to be used as assertions for reasoning about graph programs. For a given precondition and a given rule schema, we then show how to construct a strongest liberal postcondition. From this construction, we are also able to construct a strongest liberal postcondition over a loop-free program and a precondition. Also, we are able to express a condition of host graphs which implies the existence of failure or successful execution of loop-free graph programs. With this, we can have proof rules that can handle a bigger class of graph programs than proof rules that are used in [17].

There are a lot of properties we introduce in this paper. Considering the page limit, we cannot put the proofs of the properties in this paper. This paper focuses on the introduction of the properties, but the complete proof supporting them can be found at [**?**].

The remainder of this paper is structured as follows: Section 3 describes first-order formulas for graph program. In Section 4, we show how we construct a strongest liberal postcondition for first-order formulas. In Section 6, we describe proof rules we use for reasoning about graph programs with first-order formulas. In Section 7, we show an example of graph program verification with our approach. In Section 8, we compare our approach with other approaches in the literature. Finally, we conclude and give some topics for future work in Section 9. A theory about graph programming language GP 2 can be found in the appendix.

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 [15]. In this section, we briefly introduce graph transformation systems in GP 2. For more detail documentation of GP 2, we refer readers to [2].

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. $\square$

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 ($p(v) = 0$), bold circle for a rooted node ($p(v) = 1$). A standard circle also represents a node with undefined rootedness ($p(v) = \perp$). We use the same representation because nodes with undefined rootedness only exist in the interface of GP 2 rules, and the interface contain only these kind of nodes.

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 $\{\texttt{none}, \texttt{red}, \texttt{blue}, \texttt{green}, \texttt{grey}\}$. A set of edge marks, denoted by $\mathbb{M}_E$, is the set $\{\texttt{none}, \texttt{red}, \texttt{blue}, \texttt{green}, \texttt{dashed}\}$. 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:

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

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

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:

$$\begin{array}{lll} & & \texttt{int} \\ \texttt{list} \supseteq \texttt{atom} & \begin{array}{c} \supseteq \\ \supseteq \end{array} & \\ & & \texttt{string} \supseteq \texttt{char} \end{array}$$

where the domain for $\texttt{list}, \texttt{atom}, \texttt{int}, \texttt{string}$, and $\texttt{char}$ 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:

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 $\langle \ell, m \rangle \in \mathbb{E} \times (\mathbb{M}_V \cup \{\texttt{any}\})$ and edge label $\langle \ell, m \rangle \in \mathbb{E} \times (\mathbb{M}_E \cup \{\texttt{any}\})$ $\square$

Definition 5 (GP 2 host graphs and rule graphs). A *host graph* G is a graph over $\mathcal{L}$ with labelling functions $l_G = \langle \ell_G^V, m_G^V \rangle$ and $m_G = \langle \ell_G^E, m_G^E \rangle$ where $\ell_G^V : V_G \to \mathbb{L}$, $m_G^V : V_G \to \mathbb{M}_V$, $\ell_G^E : E_G \to \mathbb{L}$, and $m_G^E : E_G \to \mathbb{M}_E$.

A *rule graph* H is a graph over $\mathcal{S}$ with labelling functions $l_H = \langle \ell_H^V, m_H^V \rangle$ and $m_H = \langle \ell_H^E, m_H^E \rangle$ where $\ell_H^V : V_H \to \mathbb{E}$, $m_H^V : V_H \to \mathbb{M}_V \cup \{\texttt{any}\}$, $\ell_H^E : E_H \to \mathbb{E}$, and $m_H^E : E_H \to \mathbb{M}_E \cup \{\texttt{any}\}$. $\square$

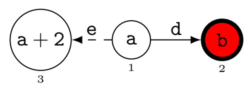
If we consider the grammars of Definition 1 and Definition 4, it is obvious that $\mathbb{L}$ is part of expressions that can be derived in the latter grammar. Hence, $\mathcal{L} \subset \mathcal{S}$, which means we can consider host graphs as special cases of rule graphs. From here, we may refer 'rule 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. In GP 2, we only need to define graphs in $\mathcal{G}(\mathcal{L})$ (including $\mathcal{G}(\mathcal{RS})$), such that labels are always defined and nodes without additional information'(B)' represent unrooted nodes.

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, a) (2[R], b#red) (3, a + 2) | (e1, 1, 2, d) (e2, 1, 3, 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 preserves 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. 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* $r = \langle L \leftarrow K \rightarrow R\rangle$ comprises totally labelled rule graphs L and R, a graph K containing only unlabelled nodes with undefined rootedness, and inclusions $K \rightarrow L$ and $K \rightarrow R$. 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. □

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.

Definition 10 (Label assignment). Given a rule graph L and let X be the set of all variables in L. For $x \in X$, let $\mathrm{dom}(x)$ denotes the domain of x associated with the type of x. A *label assignment* of L is $\alpha_L = \langle \alpha_{\mathbb{L}}, \mu_{\mathbb{M}}\rangle$ where $\alpha_{\mathbb{L}} : X \rightarrow \mathbb{L}$ is a function assigning a list to each $x \in X$ such that for each $x \in X$,

$\alpha_{\mathbb{L}}(x) \in \mathrm{dom}(x)$, and $\mu_{\mathbb{M}} = \langle \mu_V : V_L \to \mathbb{M}_V \backslash \{\texttt{none}\}, \mu_E : E_L \to \mathbb{M}_E \backslash \{\texttt{none}\} \rangle$ is a partial function assigning mark `grey`, `red`, `blue`, or `green` to each node whose mark is `any`, and mark `dashed`, `red`, `blue`, or `green` to each edge whose mark is `any`. $\square$

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

The truth value of $\Gamma^{g,\alpha}$ is required for the application of a conditional rule schema. In addition, the application also depends on the dangling condition, which is a condition that asserts the production of a graph after node removal.

Definition 11 (Dangling condition; match). Let $r = L \leftarrow K \to R$ be a rule schema with host graphs L, R, G be a host graph, and $g : L \to G$ 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.

Definition 12 (Direct derivation; comatch). A *direct derivation* from a host graph G to a host graph H via a rule $r = \langle L \leftarrow K \to R \rangle$ 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. $\square$

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

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, 10]. In [2], 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.

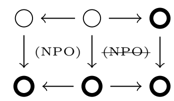


Fig. 2: Non-natural double-pushout

The natural double-pushout construction such that we have natural double-pushout is described in [2, 3], that are:

1. To obtain D, remove all nodes and edges in $g(LK)$ 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 RK 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 $r = \langle L \leftarrow K \to R,\ \Gamma \rangle$, and host graphs G, H. G directly derives r, denoted by $G \Rightarrow_{r,g} H$ (or $G \Rightarrow_r H$), if there exists a premorphism $g : L \to G$ and a label assignment α_L such that:
(i) $g : L^\alpha \to G$ is an injective morphism,
(ii) $\Gamma^{g,\alpha}$ is true,
(iii) $G \Rightarrow_{r^{g,\alpha},g} H$. □

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

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

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

where Condition is the set of GP 2 rule conditions as defined in Definition 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.

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

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

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

Configurations in GP 2 represents a program state of program execution in any stage. Configurations are given by $(\mathtt{ComSeq}\times\mathcal{G}(\mathcal{L})) \cup \mathcal{G}(\mathcal{L}) \cup (\mathtt{fail}))$, 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.

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

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

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

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

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

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

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

Fig. 4: Inference rules for core commands [16]

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

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

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

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

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

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

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

Fig. 5: Inference rules for derived commands [16]

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

Definition 14 (Semantic function [16]). The semantic function $[\![\_]\!]$: ComSeq $\to (\mathcal{G}(\mathbb{L}) \to 2^{\mathcal{G}(\mathbb{L}) \cup \{fail, \bot\}})$ is defined by

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

$\square$

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 $\mathcal{R} = \{r_1, \cdots, r_n\}$, 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. $\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
3. $\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, 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 $\texttt{if}\, (\texttt{Delete}!;\, \{\texttt{nothing}, \texttt{add}\};\, \texttt{zero})\, \texttt{then}\, P\, \texttt{else}\, Q$ 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 $\mathcal{R} = \{r_1, \cdots, r_n\}$, 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. $\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, and
4. $\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$. □

3 First-Order Formulas for Graph Programs

In this section, we define first-order formulas which are able to express properties of GP 2 graphs. Also, we define structural induction on the first-order formulas and replacement graphs which later can be used to show satisfaction of a first-order formula in a morphism.

3.1 Syntax

Our first-order (FO) formulas have logical connectives, variables, constants, also auxiliary, predicate, and function symbols.

Definition 17 (Alphabet of a first-order language). The alphabet of a first-order language consists of the following sets of symbols:

1. Logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not), true, false, equality symbols $=, \neq, >, \geq, <, \leq$, and quantifiers $\exists_\mathrm{V}, \exists_\mathrm{E}, \exists_\mathrm{L}$ for nodes, edges, and labels respectively.
2. Variables: a countably infinite set of lowercase letters.
3. Predicate symbols: $\mathrm{int}, \mathrm{char}, \mathrm{string}, \mathrm{atom}, \mathrm{edge}, \mathrm{root}$.
4. Function symbols: s (source), t (target), $\mathrm{l_V}$ (node label), $\mathrm{l_E}$ (edge label), $\mathrm{m_V}$ (node mark), $\mathrm{m_E}$ (edge mark), indeg, outdeg, length, integer operators $+, -, *, /$, label operator :, and string operator . (concatenation).
5. Constants: all elements in $\mathbb{L}$, $\mathrm{empty}, \mathrm{none}, \mathrm{red}, \mathrm{green}, \mathrm{blue}, \mathrm{green}, \mathrm{dashed}, \mathrm{grey}$, and any.

$\square$

Here we differentiate variables in seven kinds, which are first-order variables (single variables) for nodes, edges, and labels where labels are typed as in GP 2. Table 1 shows the seven kinds of variables and their domains in a graph G.

Table 1: Kinds of variables and their domain on a graph G

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

The syntax of FO formulas is given by the grammar of Figure 6. In the syntax, NodeVar and EdgeVar represent disjoint sets of first-order node and edge variables, respectively. We use ListVar, AtomVar, IntVar, StringVar, and CharVar for sets of first-order label variables of type $\mathrm{list}, \mathrm{atom}, \mathrm{int}, \mathrm{string}$, and char respectively. The nonterminals Character and Digit in the syntax represent the fixed character set of GP 2, and the digit set $\{0, \cdots, 9\}$ respectively.

The quantifiers $\exists_\mathrm{V}, \exists_\mathrm{E}$, and $\exists_\mathrm{L}$ in the grammar are reserved for variables of nodes, edges, and labels respectively. The function symbols $\mathrm{indeg}, \mathrm{outdeg}$ and

Formula ::= true | false | Cond | Equal
  | Formula (‘∧’ | ‘∨’) Formula | ‘¬’Formula | ‘(’Formula‘)’
  | ‘$\exists_\mathtt{V}$’ (NodeVar) ‘(’Formula‘)’
  | ‘$\exists_\mathtt{E}$’ (EdgeVar) ‘(’Formula‘)’
  | ‘$\exists_\mathtt{L}$’ (ListVar) ‘(’Formula‘)’
Number ::= Digit {Digit}
Cond ::= (int | char | string | atom) ‘(’Var‘)’
  | Lst (‘=’ | ‘≠’) Lst | Int (‘>’ | ‘>=’ | ‘<’ | ‘<=’) Int
  | edge ‘(’ Node ‘,’ Node [‘,’ Lst] [‘,’ EMark] ‘)’ | root ‘(’ Node ‘)’
Var ::= ListVar | AtomVar | IntVar | StringVar | CharVar
Lst ::= empty | Atm | Lst ‘:’ Lst | ListVar | $\mathrm{l_V}$ ‘(’Node‘)’ | $\mathrm{l_E}$ ‘(’EdgeVar‘)’
Atm ::= Int | String | AtomVar
Int ::= [‘-’] Number | ‘(’Int‘)’ | IntVar | Int (‘+’ | ‘-’ | ‘*’ | ‘/’) Int
  | (indeg | outdeg) ‘(’Node‘)’ | length ‘(’AtomVar | StringVar | ListVar‘)’
String ::= ‘ “ ’ Character ‘ ” ’ | CharVar | StringVar | String ‘.’ String
Node ::= NodeVar | (s | t) ‘(’ EdgeVar‘)’
EMark ::= none | red | green | blue | dashed | any
VMark ::= none | red | blue | green | grey | any
Equal ::= Node (’=’ | ‘≠’) Node | EdgeVar (’=’ | ‘≠’) EdgeVar
  | Lst (’=’ | ‘≠’) Lst | $\mathrm{m_V}$‘(’Node‘)’ (’=’ | ‘≠’) VMark
  | $\mathrm{m_E}$‘(’EdgeVar‘)’ (’=’ | ‘≠’) EMark

Fig. 6: Syntax of first-order formulas

length work similar with functions with the same names in GP 2 rule schema conditions. In addition, we also have unary functions $\mathrm{s}, \mathrm{t}, \mathrm{l_V}, \mathrm{l_E}, \mathrm{m_V}$, and m_E. These functions return the mapping result of the argument based on their functions as defined in Definition 2. For example, the function s takes an edge variable in the argument and returns the node that is the source of the edge in a host graph. The predicate edge expresses the existence of an edge between two nodes. The predicates int, char, string, atom are typing predicates to specify the type of the variable in their argument. When a variable is not an argument of any typing predicate, then the variable is a list variable. We have the predicate root to express rootedness of a node. For brevity, we sometimes write $\forall_V x(c)$ for $\neg\exists_\mathrm{V} x(\neg c)$ and $\exists_\mathrm{V} x_1, \cdots, x_n(c)$ for $\exists_\mathrm{V} x_1(\exists_\mathrm{V} x_2(...\exists_\mathrm{V} x_n(c)\cdots))$ (also for edge and label quantifiers). Also, we define ’term’ as the set of variables, constants, and functions in first-order formulas.

3.2 Structural induction on first-order formulas

To prove properties related to our first-order formulas, we classify first-order formulas into eight cases, based on their forms. To prove that some properties hold for these cases, we define structural induction on first-order formulas. Three cases are defined as base cases since they are formed from terms while the others are defined as inductive cases since they can be formed from other FO formulas. As mentioned before, terms can exist as a variable, a constant, or functions (including operators). For terms, we also define a structural induction on terms with variables and constants are its base cases.

Definition 18 (Structural induction on terms). Given a property *Prop*. Proving that *Prop* holds for all terms by *structural induction on terms* is done by:

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

For simplicity, we do not consider a FO formula in the form (c) as it is equivalent to FO formula c. We also do not include the predicate edge because we can express it as $\exists_\mathrm{E}\mathrm{z(s(z) = x \wedge t(z) = y)}$. The optional arguments list and mark of the predicate edge can be conjunct inside the quantifier, e.g. the predicate $\mathrm{edge(x, y, 5, none)}$ can be expressed as $\exists_\mathrm{E}\mathrm{z(s(z) = x \wedge t(z) = y \wedge l_E(z) = 5 \wedge m_E z = none)}$.

Definition 19 (Structural induction on first-order formulas). Given a property *Prop*. Proving that *Prop* holds for all FO formulas by *structural induction on FO formulas* is done by:

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

3.3 Satisfaction of a first-order formula

The satisfaction of a FO formula c in a host graph G relies on assignments. A formula assignment of c on G is defined in Definition 20. Informally, a formula assignment is a function that maps free variables to constants in their own domain. When we have an assignment for a FO formula on a graph, we can check the satisfaction of the FO formula. The satisfaction of a FO formula on a graph is then defined in Definition 21.

Definition 20 (Formula assignment). Let c be a FO formula, X and Y be the set of free node and edge variables in c respectively, and Z be the set of free list variables in c. For a free variable x, $\text{dom}(x)$ 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 $\alpha_G = \langle \alpha_V : X \to V_G, \alpha_E : Y \to E_G) \rangle$ and $\alpha_{\mathbb{L}} = Z \to \mathbb{L}$ such that for each free variable x, $\alpha(x) \in \text{dom}(x)$. We then denote by c^α the FO formula c after replacement of each term y to y^α where y^α is defined inductively:

1. If y is a free variable, $\text{y}^\alpha = \alpha(x)$;
2. If y is a constant, $\text{y}^\alpha = y$;
3. If $y = \text{length(x)}$ for some variable x, y^α returns 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 y if x is a list variable;
4. If y is $\text{s(x)}, \text{t(x)}, \text{l}_\text{E}(\text{x}), \text{m}_\text{E}(\text{x}), \text{l}_\text{V}(\text{x}), \text{m}_\text{V}(\text{x}), \text{indeg(x)}$, or outdeg(x), y^α is $s_G(x^\alpha)$, $t_G(x^\alpha)$, $\ell^E_G(x^\alpha), m^E_G(x^\alpha), \ell^V_G(x^\alpha), m^V_G(x^\alpha)$, indegree of x^α in G , or outdegree of x^α in G, respectively;
5. If $y = x_1 \oplus x_2$ for $\oplus \in \{+, -, *, /\}$ and integers x_1^α, x_2^α, $y^\alpha = x_1 \oplus_{\mathbb{Z}} x_2$;
6. If $y = x_1.x_2$ for some terms x_1^α, x_2^α, y^α is string concatenation x_1 and x_2;
7. If $y = x_1 : x_2$ for some list x_1^α, x_2^α, y^α is list concatenation x_1 and x_2;

□

Definition 21 (Satisfaction). Given a graph G and a first-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 $G \models^\alpha c$), that is, for each Boolean sub-expression b^α of c^α, the value of b^α in $\mathbb{B}$ is inductively defined:

1. If $b^\alpha = \text{true}$ (or $b = \text{false}$), then b^α is true (or false);
2. If $b^\alpha = \text{int(x)}, \text{char(x)}, \text{string(x)}, \text{atom(x)}$, or root(x), b^α is true if only if $x^\alpha \in \mathbb{Z}, x^\alpha \in \text{Char}, x^\alpha \in \text{Char}^*, x^\alpha \in \mathbb{Z} \cup \text{Char}^*$, or $p_G(x^\alpha) = 1$ respectively.
3. If b^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 $\otimes$. Then if b^α has the form $t_1 \ominus t_2$ where $\ominus \in \{=, \neq\}$ and $t_1, t_2 \in V_G \cup E_G \cup \mathbb{L} \cup \mathbb{M}\{\text{any}\}$, b^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 $t_1 = \text{any}$, b^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 $t_2 = \texttt{any}$).
4. If b^α has the form $b_1 \oslash b_2$ where $\oslash \in \{\vee, \wedge\}$ and b_1, b_2 are Boolean expressions, b^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 $\oslash$.
5. If the form of b^α is $\neg b_1$ where b_1 is a Boolean expression, b^α is true if and only if b_1 is false.
6. If b^α has the form $\exists_\text{V} e_1(e_2)$ where e_1 is a first-order node variable and e_2 is a Boolean expression, b^α is true if and only if there exists $v \in V_G$ such that when we add $e_1 \mapsto v$ to assignment α_G, e_2 is true.

7. If b^α has the form $\exists_\mathrm{E} e_1(e_2)$ where e_1 is a first-order edge variable and e_2 is a Boolean expression, b^α is true if and only if there exists $e \in E_G$ such that when we add $e_1 \mapsto e$ to assignment α_G, e_2 is true.
8. If b^α is in the form $\exists_\mathrm{L} e_1(e_2)$ where e_1 is a first-order list variable and e_2 is a Boolean expression, b^α is true if and only if there exists $l \in \mathbb{L}$ such that when we add $e_1 \mapsto l$ to assignment α, e_2 is true.

□

3.4 First-order formulas in rule schema application

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

Definition 22 (Conditions over a graph). Given a graph G. A *condition over* G is obtained from a first-order formula by substituting node (or edge) identifiers in G for free node (or edge) variables in the first-order formula. □

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

Example 2. Let G and H be graphs where $V_G = \{1, 2\}$ and $V_H = \{1\}$.
1. $c_1 = \exists_\mathrm{E}\mathrm{x}(\mathrm{s}(\mathrm{x}) = 1)$ is a condition over G, also over H
2. $c_2 = \forall_\mathrm{V}\mathrm{x}(\mathrm{edge}(\mathrm{x}, 1) \wedge \mathrm{indeg}(\mathrm{x}) = 2)$ is a condition over G, but not over H

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

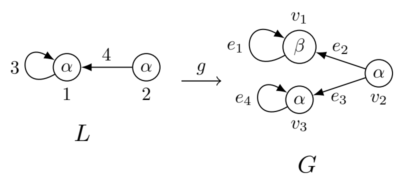
With a condition over a graph, we can express properties of left and right-hand graphs with explicitly mentioning node/edge identifiers in the graphs. In graph program verification, we need to express the properties of the initial and output graph with respect to a given rule schema. In [17, 8], they express them by showing the satisfaction of a condition on a morphism. Here, we define a replacement graph H of a host graph G with respect to an injective morphism g, where H is isomorphic to G and there exists an inclusion from the domain of g to H.

Definition 23 (Replacement graph). Given an injective (pre)morphism $g : L \to G$ where $V_G \cap V_L = \{v_1, \cdots, v_n\}$ and $E_G \cap E_L = \{e_1, \cdots, e_m\}$. Let also

$U = \{u_1, \cdots, u_n\}$ be a set of identifiers not in V_L and V_G, and $W = \{w_1, \cdots, w_n\}$ be a set of identifiers not in E_L and E_G. *Graph replacement* $\rho_g(G)$ is is obtained from G by renaming every item $g(i)$ to i for $i \in V_G$ and $i \in E_G$, every v_i to u_i for $i = 1, \cdots, n$, and every e_i to w_i for $i = 1, \cdots, m$, such that $V_{\rho_g(G)} = V_G \cup V_L \cup U$ and $E_{\rho_g(G)} = E_G \cup E_L \cup W$. $\square$

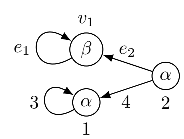
From the definition above, it is obvious that a host graph and its replacement graph are isomorphic. For a host graph G, a host graph L, and a morphism $g : L \to G$, it is also obvious that there exists an inclusion $f : L \to \rho_g(G)$, because g preserves identifiers, sources, targets, and labels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

Example 3. Given g, a morphism from L to G as follows:



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

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



Now we have defined a condition over a graph to express properties of a host graph w.r.t the left-hand graph or right-hand graph. Let $\langle L \leftarrow K \to R\rangle$ be a rule schema, and ac_L, ac_R denote a condition over a rule graph L and R respectively. To associate ac_L and ac_R with the rule schema, we define a *generalised rule schema*. Unlike a rule schema, a generalised rule schema consists of an unrestricted rule schema that allows both left and right application condition.

Definition 24 (Generalised rule schema). Given an unrestricted rule schema $r = \langle L \leftarrow K \to R\rangle$. *A generalised rule* is a tuple $w = \langle r, ac_L, ac_R\rangle$ where ac_L is a condition over L and ac_R is a condition over R. We say ac_L the left application condition and ac_R the right application condition. The inverse of w, written w^{-1}, is then defined as the tuple $\langle r^{-1}, ac_R, ac_L\rangle$ where $r^{-1} = \langle R \leftarrow K \to L\rangle$. $\square$

The application of a generalised rule schema is similar to the application of a rule schema. But here, we also check the satisfaction of both ac_L and ac_R in the replacement of input graph G and final graph H by match and comatch respectively.

Definition 25 (Application of generalised rule schema). Given a generalised rule schema $w = \langle r, ac_L, ac_R\rangle$ with an unrestricted rule schema $r = \langle L \leftarrow$

$K \to R\rangle$. There exists a direct derivation from G to H by w, written $G \Rightarrow_{w,g,g^*} H$ (or $G \Rightarrow_w H$) iff there exists premorphisms $g : L \to G$ and $g^* : R \to H$ and label assignments α_L and β_R where $\beta_R(i) = \alpha_L(i)$ for every variable i in L such that i is in R, also for every node/edge i where $m_L(i) = m_R(i) = \mathtt{any}$, such that:
(i) $g : L^\alpha \to G$ is an injective morphism
(ii) $g^* : R^\beta \to H$ is an injective morphism
(iii) $\rho_g(G) \vDash ac_L^\alpha$,
(iv) $\rho_{g^*}(H) \vDash ac_R^\beta$,
(v) $G \Rightarrow_{r^{\alpha,g},g} H$,
where $G \Rightarrow_{r^\alpha,g} H$ denotes the existence of natural pushouts (1) and (2) as in the diagram of Fig. 7. □

$$\begin{tikzcd}
& L^\alpha \arrow[dl, bend right, "\text{incl}"'] \arrow[d, "g"'] \arrow[r, phantom, "(1)" below] & K \arrow[l] \arrow[r] \arrow[d] \arrow[r, phantom, "(2)" below] & R^\beta \arrow[d, "g^*"] \arrow[dr, bend left, "\text{incl}"] & \\
ac_L^\alpha \dashv \rho_g(G) \arrow[r, phantom, "\cong"] & G & D \arrow[l] \arrow[r] & H \arrow[r, phantom, "\cong"] & \rho_{g^*}(H) \vDash ac_R^\beta
\end{tikzcd}$$

Fig. 7: Direct derivation for generalised rule schema

Recall the application of conditional rule schema in Definition 13. The condition of the rule schema is clearly can be considered as the left-application condition of the rule schema. Since there is no right-application condition in a conditional rule schema, there is no requirement about the condition such that we can always consider true as the right-application condition of a conditional rule schema.

Definition 26 (Generalised version of a conditional rule schema). Given a conditional rule schema $\langle r, \Gamma\rangle$. The generalised version of r, denoted by $r^\vee$, is the generalised rule schema $r^\vee = \langle r, \Gamma^\vee, \text{true}\rangle$ where $\Gamma^\vee$ is obtained from Γ by replacing the notations $\text{not}, != , \text{and}, \text{or}, \#$ with $\neg, \neq, \wedge, \vee, `,'$ (comma symbol) respectively. □

Lemma 1. Given a conditional rule schema $\langle r, \Gamma\rangle$ with $r = \langle L \leftarrow K \to R\rangle$. Then for any host graphs G, H,

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

Proof. (Only if). Recall the restrictions about variables and any-mark of a rule schema. It is obvious that every variable in R is in L and every node/edge with mark `any` in R is marked `any` in L as well. From Definition 13, we know that $G \Rightarrow_r H$ asserts the existence of α_L and premorphism $g : L \to G$ such that: 1) $g : L^\alpha \to G$ is an injective morphism, 2) $\Gamma^{\alpha,g}$ is true in G, and 3) $G \Rightarrow_{r^{\alpha,g},g} H$. From 3) and the variable restrictions mentioned above, it is obvious that there exists morphism $g^* : R^\alpha \to H$, and $\rho_{g^*}(H) \vDash ac_R$ because all graphs satisfy

true. Hence, (ii), (iv), and (v) of Definition 25 are satisfied. Point 1) then asserts (i) of Definition 25. The fact that $\Gamma^{\alpha,g}$ is true in G from point 2) is then asserts $\rho_g(G) \vDash \Gamma^\vee$ because it is obvious that the change of symbols does not change the semantics of the condition. Moreover, $\rho_g(G)$ is a replacement graph w.r.t. g such that evaluating $\Gamma^{\alpha,g}$ in G is the same as evaluating Γ^α in $\rho_g(G)$.
(If). Similarly, from Definition 25, we know that $G \Rightarrow_{r^\vee} H$ asserts the existence of label assignment α_L and premorphism $g : L \to G$ such that: 1) $g : L^\alpha \to G$ is an injective morphism, 2)$\rho_g(G) \vDash \Gamma^\vee$, and 3)$G \Rightarrow_{r^{\alpha,g},g} H$. These obviously assert $G \Rightarrow_r H$ from Definition 13 and the argument about $\Gamma^\vee$ above.

Remark 2. For morphism $g : L^\alpha \to G$, the semantics of Γ in G with respect to g and $\Gamma^\vee$ in $\rho_g(G)$ is identical. From here, Γ also refers to $\Gamma^\vee$ when it obviously refers to a condition over L.

Lemma 2. Given a generalised rule schema $w = \langle r, ac_L, ac_R\rangle$ with an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ and label assignment α_L. Then for host graphs G and H with premorphisms $g : L \to G$ and $g^* : R \to H$,

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

Proof.
(Only if.) From Definition 25 we know that when $G \Rightarrow_{w,g,g^*} H$, it means that there exists label assignment α_L and β_R where $\alpha_L(i) = \alpha_R(i)$ for every variable i in L such that i is in R, and for every node/edge i where $m_L(i) = m_R(i) = \texttt{any}$, such that $g : L^\alpha \to G$ and $g^* : R^\beta \to H$ are injective morphisms where
(i) $\rho_g(G) \vDash ac_L^\alpha$
(ii) $\rho_{g^*}(H) \vDash ac_R^\beta$
(iii) $G \Rightarrow_{r^\alpha,g} H$.
These are obviously defines direct derivation $H \Rightarrow_{(r^{-1})^{g^*,\alpha},g^*} G$ such that $H \Rightarrow_{w^{-1},g^*,g} G$.
(If). We can apply the above proof analogously. □

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

Definition 27 (Specifying a totally labelled graph). Given a totally labelled graph L with $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let $X = \{x_1, \cdots, x_k\}$ be the set of all list variables in L, and Type(x) for $x \in X$ is int(x), char(x), string(x), 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$_L(v)$ = root(v) if $p_L(v) = 1$, and Root$_L(v) = \neg$root(v) otherwise. *A specification of* L, denoted by Spec(L), is the condition over L:

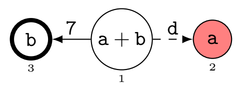
$$\bigwedge_{\mathrm{i=1}}^{\mathrm{k}} \mathrm{Type}(x_i) \ \wedge \ \bigwedge_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{l_V(v_i)} = \ell_L(v_i) \ \wedge \ \mathrm{m_V(v_i)} = m_L(v_i) \ \wedge \ \mathrm{Root}_L(v_i)$$

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

□

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

Example 4 (Specification of L). Let us consider the graph L below:



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

$$\mathrm{int(a)} \wedge \mathrm{int(b)} \wedge \mathrm{l_V(1) = a + b} \wedge \mathrm{l_V(2) = a} \wedge \mathrm{l_V(3) = b}$$
$$\wedge \mathrm{m_V(1) = none} \wedge \mathrm{m_V(2) = red} \wedge \mathrm{m_V(3) = none} \wedge \neg\mathrm{root(1)} \wedge \neg\mathrm{root(2)} \wedge \mathrm{root(3)}$$
$$\wedge \mathrm{s(e1) = 1} \wedge \mathrm{t(e1) = 2} \wedge \mathrm{s(e2) = 1} \wedge \mathrm{t(e2) = 3} \wedge \mathrm{l_E(e1) = d} \wedge \mathrm{l_E(e2) = 7}$$
$$\wedge \mathrm{m_E(e1) = dashed} \wedge \mathrm{m_E(e2) = none}$$

When a graph G satisfying Spec(L), it means G has a subgraph H with identical node and edge identifiers and with the same structure (sources, targets, and rootedness) as L. The labels of H and L should also be the same if both are host graphs, but not necessarily if at least one of them is a rule graph. However, if G is a host graph satisfying Spec(L), then there must exist label assignment α_L such that Spec(L^α) is satisfied by G, yields to the existence of inclusion $L^\alpha \to G$.

Proposition 1 (Spec(L) and inclusion). Given a rule graph L and a host graph G where $V_L \subseteq V_G$ and $E_L \subseteq E_G$. Then, $G \vDash$ Spec(L) if and only if there exists a label assignment α_L such that there exists inclusion $g : L^\alpha \to G$.

Proof. Let us consider the construction of Spec(L). It is clear that there is no node or edge variables in the condition. Hence, G satisfies Spec(L) if and only if there exists an assignment β for all list variables in Spec(L) and a partial function $\mu : V_L \cup E_L \to \mathbb{M}\backslash\{\texttt{none}\}$ for every item i whose mark is `any` such that substituting $\beta(x)$ for every variable x and $\mu(i)$ for every `any`-mark associated with i in Spec(L) resulting a valid statement in G. Let we denote by $V_L = \{v_1, \cdots, v_n\}$, $E_L = \{e_1, \cdots, e_m\}$, and $X = \{x_1, \cdots, x_p\}$ the set of all nodes, edges, label variables in L. From the semantics of satisfaction, it is clear that

$$\bigwedge_{i=1}^{n} \ell_G^V(v_i) = (\ell_L^V(v_i))^\beta \ \wedge \ m_G^V(v_i) = (m_L^V(v_i))^\mu \ \wedge \ \mathrm{Root}_G(v_i)$$

$$\wedge \bigwedge_{i=1}^{m} s_G(e_i) = s_L(e_i) \wedge t_G(e_i) = t_L(e_i) \wedge \ell_G^E(e_i) = (\ell_L^E(e_i))^\beta \wedge m_G^E(e_i) = (m_L^E(e_i))^\mu$$

Define $g(i) = i$ for every item $i \in V_L \cup E_L$ (such that identifiers are preserved by g), and $\alpha = \langle \beta, \mu \rangle$. It is clear that g preserves sources, targets, lists, marks, and rootedness. □

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

Definition 28 (Variablisation of a condition over a graph). Given a graph L and a condition c over L where $\{v_1, \cdots, v_n\}$ and $\{e_1, \cdots, e_m\}$ represent the set of node and edge constants in c respectively. Let $x_1, \cdots, x_n$ be node variables not in c and $y_1, \cdots, y_m$ be edge variables not in c. *Variablisation of* c, denoted by Var(c), is the FO formula

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

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

Lemma 3. Given a graph L and a condition c over L. For every host graph G and morphism $g : L \to G$,

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

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

If we apply this variablisation to Spec(L) for a rule graph L, morphism as in Proposition 1 should also exist but without necessarily preserves identifiers.

Lemma 4. Given rule graph L and host graph G. Then, $G \vDash \mathrm{Var}(\mathrm{Spec}(L))$ if and only if there exists a label assignment α such that there exists injective morphism $g : L^\alpha \to G$.

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

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

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

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

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

Define $g(i) = \beta(i)$ for every item $i \in V_L \cup E_L$, and $\alpha = \langle \gamma, \mu \rangle$. It is clear that $g : L^{\alpha} \to G$ preserves sources, targets, lists, marks, and rootedness. □

3.5 Properties of first-order formulas

Lemma 5. Given a FO formula c and two isomorphic host graphs G and H with isomorphism $f : G \to H$. Let $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ and $\beta = \langle \beta_H, \beta_{\mathbb{L}} \rangle$ be formula assignments where $\beta_H(x) = f(\alpha_G(x))$ for every node and edge variable x in c and $\beta_{\mathbb{L}}(x) = \alpha_{\mathbb{L}}(x)$ for every list variable x in c. Then,

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

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

1. If $c = \text{true}$ or $c = \text{false}$, it is obvious that $G \vDash^{\alpha} c$ iff $H \vDash^{\beta} c$
2. If c is a predicate $\text{P}(x)$ for $P \in \{\text{int}, \text{char}, \text{string}, \text{atom}\}$ and some list variable x, the satisfaction of the predicate is independent on host graphs. Also, it is obvious that $x^{\alpha} = x^{\beta}$ such that $\text{P}(x^{\alpha})$ is true in every host graphs iff $\text{P}(x^{\beta})$ is true in every host graph
3. If $c = \text{root(x)}$ for some term x representing a node, $x^{\beta} = g(x^{\alpha})$. From Definition 6, we know that $p_G(x^{\alpha}) = p_H(g(x^{\alpha}))$. Hence, $\text{root}(\text{x}^{\alpha}$ is true in G iff $\text{root}(\text{x}^{\beta})$ is true in H
4. If $c = x_1 \otimes x_2$ for $\otimes \in \{=, \neq\}$ and terms x_1, x_2 representing edges or nodes, $x_1^{\beta} = g(x_1^{\alpha})$ and $x_2^{\beta} = g(x_2^{\alpha})$. It is clear that $x_1^{\alpha} \otimes x_2^{\alpha}$ iff $g(x_1^{\alpha}) \otimes g(x_2^{\alpha})$ because g is injective.
5. If $c = x_1 \otimes x_2$ for $\otimes \in \{=, \neq, \leq, \geq\}$ and terms x_1, x_2 representing lists, $x_1^{\alpha} = x_1^{\beta}$ and $x_2^{\alpha} = x_2^{\beta}$ (note that $l_V(x^{\alpha}) = l_V(g(x^{\alpha})) = l_V(x^{\beta})$ for all node variable x in c, and analogously for $l_E(x)$). Since the truth value of $x_1^{\alpha} \otimes x_2^{\alpha}$ does not depend on host graphs, $x_1^{\alpha} \otimes x_2^{\alpha}$ is true in G iff $x_1^{\beta} \otimes x_2^{\beta}$ is true in H

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

1. If $c = \neg c_1$, $G \vDash^{\alpha} \neg c_1$ iff c_1^{α} is false in G iff c_1^{β} is false in H iff $H \vDash^{\beta} \neg c_1$
2. If $c = c_1 \vee c_2$, $G \vDash^{\alpha} c_1 \vee c_2$ iff $G \vDash^{\alpha} c_1 \vee G \vDash^{\alpha} c_2$ iff $H \vDash^{\beta} c_1 \vee H \vDash^{\beta} c_2$ iff $H \vDash^{\beta} c_1 \vee c_2$
3. If $c = c_1 \wedge c_2$, $G \vDash^{\alpha} c_1 \wedge c_2$ iff $G \vDash^{\alpha} c_1 \wedge G \vDash^{\alpha} c_2$ iff $H \vDash^{\beta} c_1 \wedge H \vDash^{\beta} c_2$ iff $H \vDash^{\beta} c_1 \wedge c_2$
4. $G \vDash^{\alpha} \exists_{\mathrm{V}} x(c_1)$ iff $(c_1^{\alpha})^{[x \mapsto v]}$ for some $v \in V_G$ is true in G iff $(c_1^{\beta})^{[x \mapsto g(v)]}$ is true in H iff $H \vDash^{\beta} \exists_{\mathrm{V}} x(c_1)$
5. $G \vDash^{\alpha} \exists_{\mathrm{E}} x(c_1)$ iff $(c_1^{\alpha})^{[x \mapsto e]}$ for some $e \in E_G$ is true in G iff $(c_1^{\beta})^{[x \mapsto g(e)]}$ is true in H iff $H \vDash^{\beta} \exists_{\mathrm{E}} x(c_1)$
6. $G \vDash^{\alpha} \exists_{\mathrm{L}} x(c_1)$ iff $(c_1^{\alpha})^{[x \mapsto i]}$ for some $i \in \mathbb{L}$ is true in G iff $(c_1^{\beta})^{[x \mapsto i]}$ is true in H iff $H \vDash^{\beta} \exists_{\mathrm{L}} x(c_1)$

□

Corollary 1. *Given two isomorphic host graphs G and H, and a FO formula c. It is true that*

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

Proof. $G \vDash c$ iff there exists an assignment $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ such that $G \vDash^{\alpha} c$. By Lemma 5, Gc iff $H \vDash^{\beta} c$ for $\beta = \langle \beta_H, \alpha_{\mathbb{L}} \rangle$ where $\beta_H(x) = g(\alpha_G(x))$ for all node and edge variables x iff $H \vDash c$. □

Lemma 6. Given a host graph G and FO formulas c_1, c_2. Then, the following holds:

1. $G \vDash c_1 \vee c_2$ if and only if $G \vDash c_1 \vee G \vDash c_2$
2. $G \vDash c_1 \wedge c_2$ if and only if $G \vDash {}^{\alpha} c_1 \wedge G \vDash {}^{\alpha} c_2$ for some assignment α
3. $G \vDash \neg c_1$ if and only if $\neg(G \vDash {}^{\alpha} c_1)$ for some assignment α
4. $V_G \neq \emptyset \wedge G \vDash \exists_{\mathrm{V}} x(c_1)$ if and only if $G \vDash c_1$
5. $E_G \neq \emptyset \wedge G \vDash \exists_{\mathrm{E}} x(c_1)$ if and only if $G \vDash c_1$
6. $G \vDash \exists_{\mathrm{L}} x(c_1)$ if and only if $G \vDash c_1$

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

Proof.
For a condition d over G, from the definition of condition over a graph we know that $d = c^{\alpha_G}$ for some FO formula c and node/edge assignment α_G. Also, there is no free node and edge variables in d so that $G \vDash {}^{\alpha_{\mathbb{L}}} d$ for some list assignment $\alpha_{\mathbb{L}}$ is equivalent to $G \vDash^{\alpha} c$ for $\alpha = \langle \alpha_G, \alpha_{\mathbb{L}} \rangle$. Hence, we can consider a condition over a graph as a FO formula with a fixed node/label assignment. Then for some FO formulas c_1, c_2,

1. (only if) $G \models c_1 \vee c_2$ implies $G \models {}^{\alpha}c_1 \vee c_2$ for some assignment α implies $G \models {}^{\alpha}c_1$ or $G \models {}^{\alpha}c_2$ implies $G \models c_1 \vee G \models c_2$
   (if) $G \models c_1 \vee G \models c_2$ implies $G \models {}^{\alpha}c_1 \vee G \models {}^{\beta}c_2$ for some assignments α, β. It implies $(G \models {}^{\alpha}c_1 \vee c_2) \vee (G \models {}^{\beta}c_1 \vee c_2)$. Hence, $G \models c_1 \vee c_2$
2. $G \models c_1 \wedge c_2$ iff $G \models {}^{\alpha}c_1 \wedge c_2$ for some assignment α iff $G \models {}^{\alpha}c_1$ and $G \models {}^{\alpha}c_2$
3. $G \models \neg c_1$ iff c_1^{α} is false in G for some assignment α such that $G \models {}^{\alpha}c_1$ is false. Hence, $\neg(G \models^{\alpha} c_1)$
4. $G \models \exists_{\mathrm{V}} x(c_1)$ iff $G \models c_1^{[x \mapsto v]}$ for some $v \in V_G$ iff $G \models^{\alpha} c_1$ for some assignment α such that $\alpha(x) = v$ iff $G \models c_1$
5. $G \models \exists_{\mathrm{E}} x(c_1)$ iff $G \models c_1^{[x \mapsto e]}$ for some $e \in E_G$ iff $G \models^{\alpha} c_1$ for some assignment α such that $\alpha(x) = e$ iff $G \models c_1$
6. $G \models \exists_{\mathrm{L}} x(c_1)$ iff $G \models c_1^{[x \mapsto k]}$ for some $k \in \mathbb{L}$ iff $G \models^{\alpha} c_1$ for some assignment α such that $\alpha(x) = k$ iff $G \models c_1$

□

Lemma 7. Given a host graph G and a condition c over G. Let $\{v_1, \cdots, v_n\} \subseteq V_G$ and $\{e_1, \cdots, e_m\} \subseteq E_G$. Then,

1. $\exists_{\mathrm{V}}\mathrm{x}(\mathrm{c}) \equiv \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_1]} \vee \cdots \vee \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_n]} \vee \exists_{\mathrm{V}}\mathrm{x}(\mathrm{x} \neq \mathrm{v}_1 \wedge \cdots \wedge \mathrm{x} \neq \mathrm{v}_n \wedge \mathrm{c})$
2. $\exists_{\mathrm{E}}\mathrm{x}(\mathrm{c}) \equiv \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{e}_1]} \vee \cdots \vee \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{e}_m]} \vee \exists_{\mathrm{V}}\mathrm{x}(\mathrm{x} \neq \mathrm{e}_1 \wedge \cdots \wedge \mathrm{x} \neq \mathrm{v}_m \wedge \mathrm{c})$
3. $$\begin{aligned}\exists_{\mathrm{E}}\mathrm{x}(\mathrm{c}) \equiv \exists_{\mathrm{E}}\mathrm{x}(\textstyle\bigvee_{i=1}^{n}(&\textstyle\bigvee_{j=1}^{n} \mathrm{s(x) = v_i \wedge t(x) = v_j} \wedge \mathrm{c}^{[\mathrm{s(x)}\mapsto\mathrm{v_i},\mathrm{t(x)}\mapsto\mathrm{v_j}]}) \\ &\vee (\mathrm{s(x) = v_i} \wedge \textstyle\bigwedge_{j=1}^{n} \mathrm{t(x) \neq v_j} \wedge \mathrm{c}^{[\mathrm{s(x)}\mapsto\mathrm{v_i}]}) \\ &\vee (\textstyle\bigwedge_{j=1}^{n} \mathrm{s(x) \neq v_j} \wedge \mathrm{t(x) = v_i} \wedge \mathrm{c}^{[\mathrm{t(x)}\mapsto\mathrm{v_i}]}) \\ \vee (&\textstyle\bigwedge_{i=1}^{n} \mathrm{s(x) \neq v_i} \wedge \textstyle\bigwedge_{i=1}^{n} \mathrm{t(x) \neq v_i} \wedge \mathrm{c}))\end{aligned}$$

Proof.

1. $$\begin{aligned}\exists_{\mathrm{V}}\mathrm{x}(\mathrm{c}) &\equiv \exists_{\mathrm{V}}\mathrm{x}(((\mathrm{x = v_1} \vee \cdots \vee \mathrm{x = v_n}) \vee \neg(\mathrm{x = v_1} \vee \cdots \vee \mathrm{x = v_n})) \wedge \mathrm{c}) \\ &\equiv \exists_{\mathrm{V}}\mathrm{x}((\mathrm{x = v_1} \wedge \mathrm{c}) \vee \cdots \vee (\mathrm{x = v_n} \wedge \mathrm{c}) \vee (\mathrm{x \neq v_1} \wedge \cdots \wedge \mathrm{x \neq v_n} \wedge \mathrm{c})) \\ &\equiv \exists_{\mathrm{V}}\mathrm{x}(\mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_1]} \vee \cdots \vee \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_n]} \vee (\mathrm{x \neq v_1} \wedge \cdots \wedge \mathrm{x \neq v_n} \wedge \mathrm{c})) \\ &\equiv \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_1]} \vee \cdots \vee \mathrm{c}^{[\mathrm{x}\mapsto\mathrm{v}_n]} \vee \exists_{\mathrm{V}}\mathrm{x}(\mathrm{x \neq v_1} \wedge \cdots \wedge \mathrm{x \neq v_n} \wedge \mathrm{c})\end{aligned}$$
2. Analogous to point 1
3. $$\begin{aligned}\exists_{\mathrm{E}}\mathrm{x}(\mathrm{c}) &\equiv \exists_{\mathrm{E}}\mathrm{x}(((\mathrm{s(x) = v_1} \vee \cdots \vee \mathrm{s(x) = v_n}) \vee \neg(\mathrm{s(x) = v_1} \vee \cdots \vee \mathrm{s(x) = v_n})) \\ &\qquad \wedge (\mathrm{t(x) = v_1} \vee \cdots \vee \mathrm{t(x) = v_n} \vee \neg(\mathrm{t(x) = v_1} \vee \cdots \vee \mathrm{t(x) = v_n})) \wedge \mathrm{c}) \\ &\equiv \exists_{\mathrm{E}}\mathrm{x}((\mathrm{s(x) = v_1} \wedge (\mathrm{t(x) = v_1} \vee \cdots \vee \mathrm{t(x) = v_n}) \wedge \mathrm{c}) \\ &\qquad \cdots \\ &\qquad (\mathrm{s(x) = v_n} \wedge (\mathrm{t(x) = v_1} \vee \cdots \vee \mathrm{t(x) = v_n}) \wedge \mathrm{c}) \\ &\qquad (\mathrm{s(x) \neq v_1} \wedge \cdots \wedge \mathrm{s(x) \neq v_n} \wedge (\mathrm{t(x) = v_1} \vee \cdots \vee \mathrm{t(x) = v_n}) \wedge \mathrm{c}) \\ &\qquad (\mathrm{s(x) = v_i} \wedge (\mathrm{t(x) \neq v_1} \wedge \cdots \wedge \mathrm{t(x) \neq v_n}) \wedge \mathrm{c}) \\ &\qquad (\mathrm{s(x) \neq v_1} \wedge \cdots \wedge \mathrm{s(x) \neq v_n} \wedge (\mathrm{t(x) \neq v_1} \wedge \cdots \wedge \mathrm{t(x) \neq v_n}) \wedge \mathrm{c}) \\ &\equiv \exists_{\mathrm{E}}\mathrm{x}(\textstyle\bigvee_{i=1}^{n}(\textstyle\bigvee_{j=1}^{n} \mathrm{s(x) = v_i \wedge t(x) = v_j} \wedge \mathrm{c}^{[\mathrm{s(x)}\mapsto\mathrm{v_i},\mathrm{t(x)}\mapsto\mathrm{v_j}]}) \\ &\qquad \vee (\textstyle\bigwedge_{j=1}^{n} \mathrm{s(x) \neq v_j} \wedge \mathrm{t(x) = v_i} \wedge \mathrm{c}^{[\mathrm{t(x)}\mapsto\mathrm{v_i}]}) \\ &\qquad \vee (\mathrm{s(x) = v_i} \wedge \textstyle\bigwedge_{j=1}^{n} \mathrm{t(x) \neq v_j} \wedge \mathrm{c}^{[\mathrm{s(x)}\mapsto\mathrm{v_i}]}) \\ &\quad \vee (\textstyle\bigwedge_{i=1}^{n} \mathrm{s(x) \neq v_i} \wedge \textstyle\bigwedge_{i=1}^{n} \mathrm{t(x) \neq v_i} \wedge \mathrm{c}))\end{aligned}$$

□

4 Constructing a Strongest Liberal Postcondition

In this section, we introduce a way to construct a strongest liberal postcondition over a graph program. Here, conditions (including pre- and postconditions) refer to closed FO formulas.

4.1 Calculating strongest liberal postconditions

A strongest liberal postcondition is one of predicate transformers [7] for forward reasoning. It expresses properties that must be satisfied by every graph result from the application of the input rule schema to a graph satisfying the input precondition.

Definition 29 (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. $\square$

Our definition of a strongest liberal postcondition is different with the definitions in [8, 7, 6] where they define $\mathrm{SLP}(c, r)$ as a condition such that for every host graph H satisfying the condition, there exists a host graph G satisfying c where $G \Rightarrow_r H$. Lemma 8 shows that their definition and ours are equivalent.

Lemma 8. 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 $G \Rightarrow_r H$.

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

not guarantee the satisfaction of any liberal postcondition a over c and r in H because a is dependent of c. Hence, we can not guarantee that H satisfying all liberal postcondition w.r.t. r and c. $\square$

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

In this paper, $\mathrm{SLP}(c, r)$ is obtained by defining transformations $\mathrm{Lift}(c, r^\vee)$, $\mathrm{Shift}(c, r^\vee)$, and $\mathrm{Post}(c, r^\vee)$. The transformation Lift transforms the given condition c into a left-application condition w.r.t. the given unrestricted rule schema r. Then, we transform the left-application condition to right-application condition by transformation Shift. Finally, the transformation Post transforms the right-application condition to a strongest liberal postcondition (see Fig. 8).

$$\begin{tikzcd}
ac_L \arrow[r, "\text{Shift}"] & ac_R \arrow[d, "\text{Post}"] \\
c, r \arrow[u, "\text{Lift}"] \arrow[r] & \mathrm{SLP}(c, r)
\end{tikzcd}$$

Fig. 8: Constructing $\mathrm{SLP}(c, r)$

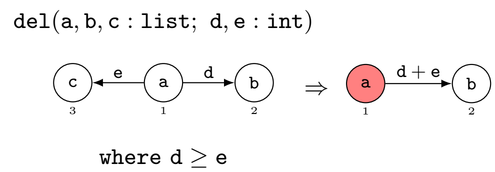
For a conditional rule schema $\langle r, \Gamma\rangle$ with rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ and a precondition c, when a graph G satisfying c and there exists a label assignment α_L such that $G \Rightarrow_{r^\alpha, g} H$ for some host graph H and injective graph morphism $g : L^\alpha \hookrightarrow G$, $ac_L^\alpha = (\mathrm{Lift}(c, r^\vee))^\alpha$ should be satisfied by G w.r.t. g. The replacement graph $\rho_g(G)$ should satisfies ac_L which means ac_L should consist of the precondition c, rule schema condition Γ, and the dangling condition.

$G \Rightarrow_{r^\alpha, g} H$ with injective graph morphism $g : L^\alpha \hookrightarrow G$ and label assignment α_L obviously assert the existence of injective morphism $g^* : R^\beta \rightarrow H$ for some label assignment β_R such that $\alpha_L(i) = \beta_R(i)$ for common element i (see Fig. 7). The graph replacement $\rho_{g^*}(H)$ then should satisfy $ac_R^\beta = (\mathrm{Shift}(c, r^\vee))^\beta$. The graph condition ac_R should describe the elements of the image of the comatch and some properties of c that are still relevant after the rule schema application.

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

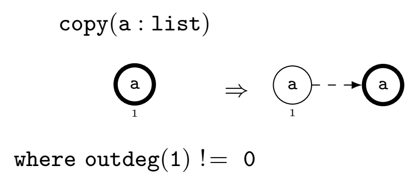
To give a better idea of the transformations we define in this chapter, we show examples after each definition. We use the conditional rule schemata $r_1 =$ `del` of

Fig. 9 and `copy` of Fig. 10 and the preconditions $q_1 = \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{m_V}(\mathrm{s}(\mathrm{x})) \neq \mathrm{none})$ and $q_2 = \exists_{\mathrm{V}}\mathrm{x}(\neg\mathrm{root}(\mathrm{x}))$ as running examples. We denote by Γ_1 and Γ_2 the GP 2 rule schema conditions $\mathrm{d} \geq \mathrm{e}$ and $\mathrm{outdeg}(1) \neq 0$ respectively. Also, we denote by r_1 and r_2 the rule schema of `del` and `copy` respectively.




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

Fig. 9: GP 2 conditional rule schema `del`




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

Fig. 10: GP 2 conditional rule schema `copy`

4.2 The dangling condition

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

Recall the dangling condition from Definition 11. $\rho_g(G)$ satisfies the dangling condition if every node $v \in L - K$ does not incident to any edge outside L. This means that the indegree and outdegree of every node $v \in L - K$ in L represent the indegree and outdegree of v in G as well.

Definition 30 (Condition Dang). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ where $\{v_1, \cdots, v_n\}$ is the set of all nodes in $L - K$. Let $indeg_L(v)$ and $outdeg_L(v)$ denotes the indegree and outdegree of v in L, respectively. The condition $\mathrm{Dang}(r)$ is defined as:

(1) if $V_L - V_K = \emptyset$ then $\text{Dang}(r) = \text{true}$
(2) if $V_L - V_K \neq \emptyset$ then

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

□

Example 5 (Condition Dang).

1. $\text{Dang}(r_1) = \text{indeg}(3) = 1 \wedge \text{outdeg}(3) = 0$
2. $\text{Dang}(r_2) = \text{true}$

Observation 1 Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. Let G be a host graph and $g : L \rightarrow G$ be a premorphism. The dangling condition is satisfied if and only if $\rho_g(G) \vDash \text{Dang}(r)$.

Proof. From the definition of the dangling condition (see Definition 11), the dangling condition is satisfied when no edge in $G - g(L)$ is incident to any node in $g(L - K)$. By the definition of replacement graph (see Definition 23), it is obvious that $G - g(L)$ is equivalent to $\rho_g(G) - L$. Then, evaluating the construct of $g(L - K)$ in G w.r.t. g is the same as evaluating the $L - K$ in $\rho_g(G)$. Hence, the dangling condition is satisfied iff no edge in $\rho_g(G) - L$ incident to any node in $L - K$, which means all nodes in $L - K$ only incident to edges in L. Hence, $\text{Dang}(r)$ is true. □

4.3 From precondition to left-application condition

Now, we start with transforming a precondition c to a left-application condition with respect to a generalised rule $w = \langle r, ac_L, ac_r \rangle$. Intuitively, the transformation is done by: 1) Find all possibilities of variables in c representing nodes/edges in an input and form a disjunction from all possibilities, denotes by $\text{Split}(c, r)$; 2) Express the dangling condition as a condition over L, denoted by $\text{Dang}(r)$; 3) Evaluate terms and Boolean expression we can evaluate in $\text{Split}(c, r)$, $\text{Dang}(r)$, and Γ, then form a conjunction from the result of evaluation, and simplify the conjunction.

A possibility of variables in c representing nodes/edges in an input graph as mentioned above refers to how variables in c can represent node or edge constants in the replacement of the input graph. A simple example would be for a precondition $c = \exists_\mathrm{V} x(c_1)$ for some FO formula c_1 with a free variable x, c holds on a host graph G if there exists a node v in G such that c_1^α where $\alpha(x) = v$ is true in G. The node v can be any node in G. In the replacement graph of G, v can be any node in the left-hand graph of the rule schema, or any node outside it. $\text{Split}(c, r)$ is obtained from the disjunction of all these possibilities.

Definition 31 (Transformation Split). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$. where $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let c be a condition over L sharing no variables with r (note that it is always possible to replace the label variables in c with new variables that are distinct from variables in r). We define the condition $\mathrm{Split}(c, r)$ over L inductively as follows:

- Base case.
  If c is true, false, a predicate $\mathrm{int(t), char(t), string(t), atom(t), root(t)}$ for some term t, or in the form $\mathrm{t_1} \ominus \mathrm{t_2}$ for $\ominus \in \{= . \neq . <, \leq, >, \geq\}$ and some terms $\mathrm{t_1, t_2}$,
  $$\mathrm{Split}(c, r) = c$$
- Inductive case.
  Let c_1 and c_2 be conditions over L.
  1) $\mathrm{Split}(c_1 \vee c_2, r) = \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r)$,
  2) $\mathrm{Split}(c_1 \wedge c_2, r) = \mathrm{Split}(c_1, r) \wedge \mathrm{Split}(c_2, r)$,
  3) $\mathrm{Split}(\neg c_1, r) = \neg\mathrm{Split}(c_1, r)$,
  4) $\mathrm{Split}(\exists_\mathrm{V}\mathrm{x}(c_1), r) = (\bigvee_{\mathrm{i=1}}^{\mathrm{n}}\mathrm{Split}(c_1^{[x\mapsto v_i]}, r)) \vee \exists_\mathrm{V}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{x{\neq}v_i} \wedge \mathrm{Split}(c_1, r)$,
  5) $\mathrm{Split}(\exists_\mathrm{E}\mathrm{x}(c_1), r) = (\bigvee_{\mathrm{i=1}}^{\mathrm{m}}\mathrm{Split}(c_1^{[x\mapsto e_i]}, r)) \vee \exists_\mathrm{E}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{x{\neq}e_i} \wedge \mathrm{inc}(c_1, r))$,
     where
     $$\begin{aligned}\mathrm{inc}(c_1, r) = \bigvee_{\mathrm{i=1}}^{\mathrm{n}}(&\bigvee_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{s(x) = v_i} \wedge \mathrm{t(x) = v_j} \wedge \mathrm{Split}(c_1^{[\mathrm{s(x)\mapsto v_i, t(x)\mapsto v_j}]}, r))\\ &\vee (\mathrm{s(x) = i} \ \wedge \ \bigwedge_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{t(x)} \neq \mathrm{v_j} \wedge \mathrm{Split}(c_1^{[\mathrm{s(x)\mapsto v_i}]}, r))\\ &\vee (\bigwedge_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{s(x)} \neq \mathrm{v_j} \ \wedge \ \mathrm{t(x) = v_i} \wedge \mathrm{Split}(c_1^{[\mathrm{t(x)\mapsto v_i}]}, r))\\ \vee (&\bigwedge_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{s(x)} \neq \mathrm{v_i} \ \wedge \ \bigwedge_{\mathrm{j=1}}^{\mathrm{n}} \mathrm{t(x)} \neq \mathrm{v_j} \wedge \mathrm{Split}(c_1, r))\end{aligned}$$
  6) $\mathrm{Split}(\exists_\mathrm{L}\mathrm{x}(c_1), r) = \exists_\mathrm{L}\mathrm{x}(\mathrm{Split}(c_1, r))$

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

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

Observation 2 Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$ where $V_L = \{v_1, \cdots, v_n\}$ and $E_L = \{e_1, \cdots, e_m\}$. Let $c = \exists_\mathrm{E} x(c_1)$ be a condition over L. Then, the following holds:

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

3. If c_1 does not contain the terms s(x) and t(x), $\mathrm{inc}(c_1, r) = \mathrm{Split}(c_1, r)$

Proof.

1. If c_1 does not contain the term s(x), then for any i, j, $c_1^{[\mathrm{s(x)}\mapsto \mathrm{v_i}, \mathrm{t(x)}\mapsto \mathrm{v_j}]} = c_1^{[\mathrm{t(x)}\mapsto \mathrm{v_j}]}$, and $c_1^{[\mathrm{s(x)}]} = c_1$. The first and the third line of $\mathrm{inc}(c_1, r)$ is the disjunction of all possibilities of $(t(x))$ is one of nodes in L while the second and forth line is about $(t(x))$ is outside the match.
2. Analogously to above.
3. If c_1 does not contain the terms s(x) and t(x), it is obvious that $c_1^{[\mathrm{s(x)}\mapsto \mathrm{v_i}, \mathrm{t(x)}\mapsto \mathrm{v_j}]} = c_1^{[\mathrm{t(x)}\mapsto \mathrm{v_j}]} = c_1^{[\mathrm{s(x)}\mapsto \mathrm{v_j}]} = c_1$. □

Example 6 (Transformation Split).

$$\begin{aligned}
\mathrm{Split}(q_1, r_1) &= \neg\ \mathrm{Split}(\exists_{\mathrm{E}}\mathrm{x}(\mathrm{s(x)} = \mathrm{t(x)}), r_1)\\
&= \neg(\mathrm{m_V}(\mathrm{s(e1)}) \neq \mathrm{none} \vee \mathrm{m_V}(\mathrm{s(e2)}) \neq \mathrm{none} \vee\\
&\qquad \exists_{\mathrm{E}}\mathrm{x}(\mathrm{x} \neq \mathrm{e1} \wedge \mathrm{x} \neq \mathrm{e2}\ \wedge\ ((\mathrm{s(x)} = 1 \wedge \mathrm{m_V}(1) \neq \mathrm{none})\\
&\qquad\qquad \vee (\mathrm{s(x)} = 2 \wedge \mathrm{m_V}(2) \neq \mathrm{none})\\
&\qquad\qquad \vee (\mathrm{s(x)} = 3 \wedge \mathrm{m_V}(3) \neq \mathrm{none})\\
&\qquad\qquad \vee (\mathrm{s(x)} \neq 1 \wedge \mathrm{s(x)} \neq 2 \wedge \mathrm{s(x)} \neq 3\\
&\qquad\qquad\quad \wedge \mathrm{m_V}(\mathrm{s(x)}) \neq \mathrm{none}))))\\
\mathrm{Split}(q_2, r_2) &= \neg\mathrm{root}(1) \vee \exists_{\mathrm{V}}\mathrm{x}(\mathrm{x} \neq 1 \wedge \neg\mathrm{root(x)})
\end{aligned}$$

Since $\mathrm{Split}(c, r)$ only disjunct all possibilities of nodes and edges that can be represented by node and edge variables in c, it should not change the semantic of c. However, we transform a condition c to a condition over L such that we may not be able to check satisfaction of $\mathrm{Split}(c, r)$ in G. However, we can always check its satisfaction in $\rho_g(G)$ for some premorphism $g : L \to G$.

Lemma 9. 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 $g : L \to G$ be a premorphism. Then,

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

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

(Base case).

$$\begin{aligned}
G \models c \quad &\overset{\mathrm{L5}}{\Leftrightarrow} \rho_g(G) \models c\\
&\Leftrightarrow \rho_g(G) \models \mathrm{Split}(c, r)
\end{aligned}$$

(Inductive case).

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

$$\begin{aligned}
1)\ G \models c_1 \vee c_2 \quad &\overset{\mathrm{L6}}{\Leftrightarrow} G \models c_1 \vee G \models c_2\\
&\Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \rho_g(G) \models \mathrm{Split}(c_2, r)
\end{aligned}$$

$$
\begin{array}{ll}
 & \overset{\text{L6}}{\Leftrightarrow} \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r) \\
2)\ G \models c_1 \wedge c_2 & \overset{\text{L6}}{\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} \\
 & \overset{\text{L6}}{\Leftrightarrow} \rho_g(G) \models \mathrm{Split}(c_1, r) \vee \mathrm{Split}(c_2, r) \\
3)\ G \models \neg c_1 & \overset{\text{L6}}{\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} \\
 & \overset{\text{L6}}{\Leftrightarrow} \rho_g(G) \models \neg\mathrm{Split}(c_1, r) \\
4)\ G \models \exists_{\mathrm{V}}\mathrm{x}(c_1) & \overset{\text{L7}}{\Leftrightarrow} G \models \bigvee_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{c_1}^{[\mathrm{x}\mapsto \mathrm{v_i}]} \vee \exists_{\mathrm{V}}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{x} \neq \mathrm{v_i} \wedge \mathrm{c_1}) \\
 & \Leftrightarrow \rho_g(G) \models \bigvee_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{Split}(\mathrm{c}_1^{[\mathrm{x}\mapsto \mathrm{v_i}]}, \mathrm{r}) \vee \exists_{\mathrm{V}}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{x} \neq \mathrm{v_i} \wedge \mathrm{Split}(\mathrm{c_1}, \mathrm{r})) \\
5)\ G \models \exists_{\mathrm{E}}\mathrm{x}(c_1) & \overset{\text{L7}}{\Leftrightarrow} G \models \bigvee_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{c_1}^{[\mathrm{x}\mapsto \mathrm{e_i}]} \vee \exists_{\mathrm{V}}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{x} \neq \mathrm{e_i} \wedge \mathrm{c_1}) \\
 & \overset{\text{L7}}{\Leftrightarrow} \rho_g(G) \models \bigvee_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{Split}(\mathrm{c}_1^{[\mathrm{x}\mapsto \mathrm{e_i}]}, \mathrm{r}) \vee \exists_{\mathrm{V}}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{x} \neq \mathrm{v_i} \wedge \mathrm{Split}(\mathrm{c_1}, \mathrm{r})) \\
 & \overset{\text{L7}}{\Leftrightarrow} \rho_g(G) \models \bigvee_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{Split}(\mathrm{c}_1^{[\mathrm{x}\mapsto \mathrm{e_i}]}, \mathrm{r}) \vee \exists_{\mathrm{V}}\mathrm{x}(\bigwedge_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{x} \neq \mathrm{v_i} \wedge \mathrm{inc}(\mathrm{c_1}, \mathrm{r})) \\
6)\ G \models \exists_{\mathrm{L}}\mathrm{x}(c_1) & \overset{\text{L6}}{\Leftrightarrow} G \models c_1 \\
 & \Leftrightarrow \rho_g(G) \models \mathrm{Split}(c_1, r) \\
 & \overset{\text{L6}}{\Leftrightarrow} \rho_g(G) \models \exists_{\mathrm{L}}\mathrm{x}(\mathrm{Split}(c_1, r))
\end{array}
$$

□

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

Definition 32 (Valuation of c). 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. Let also $F = \{\mathrm{s}, \mathrm{t}, \mathrm{l_V}, \mathrm{l_E}, \mathrm{m_V}, \mathrm{m_E}, \mathrm{indeg}, \mathrm{outdeg}, \mathrm{length}$ be the set of function syntax. Let also $y \oplus_L z$ for $\oplus \in \{+, -, *, /, :, .\}$ and $y, z \in \mathbb{L}$ denotes the value of $y \oplus z$ as desribed in Section 3.3, and $f_L(z)$ for a constant z and $f \in F$ denotes the value of $f(y)$ in L. *Valuation of c w.r.t. r*, written $\mathrm{Val}(c, r)$, 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 $f \in F$,

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

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

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

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

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

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

□

Example 7 (Valuation of a graph condition). For rules r_1 and r_2,

1. $\text{Val}(\text{Split}(q, r_1), r_1)$

$$\begin{aligned} &= \neg(\text{none} \neq \text{none} \vee \text{none} \neq \text{none} \vee \\ &\qquad \exists_{\text{E}}\text{x}(\text{x} \neq \text{e1} \wedge \text{x} \neq \text{e2} \wedge ((\text{s(x)} = 1 \wedge \text{none} \neq \text{none}) \\ &\qquad\qquad \vee (\text{s(x)} = 2 \wedge \text{none} \neq \text{none}) \\ &\qquad\qquad \vee (\text{s(x)} = 3 \wedge \text{none} \neq \text{none}) \\ &\qquad\qquad \vee (\text{s(x)} \neq 1 \wedge \text{s(x)} \neq 2 \wedge \text{s(x)} \neq 3 \wedge \text{m}_{\text{V}}(\text{s(x)}) \neq \text{none})))) \\ &\equiv \neg\exists_{\text{E}}\text{x}(\text{x} \neq \text{e1} \wedge \text{x} \neq \text{e2} \wedge \text{s(x)} \neq 1 \wedge \text{s(x)} \neq 2 \wedge \text{s(x)} \neq 3 \wedge \text{m}_{\text{V}}(\text{s(x)}) \neq \text{none}) \end{aligned}$$

Here, we replace the terms s(e1), s(e2) with node constant 1, then replace $\text{m}_\text{V}(1), \text{m}_\text{V}(2), \text{m}_\text{V}(3)$ with none. Then, we simplify the resulting condition by evaluating none $\neq$ none which is equivalent to false.

2. $\text{Val}(\text{Split}(s, r_2), r_1) = \text{false} \vee \exists_\text{V}\text{x}(\text{x} \neq 1 \wedge \neg\text{root(x)})$
$\qquad\qquad\equiv \exists_\text{V}\text{x}(\text{x} \neq 1 \wedge \neg\text{root(x)})$
Here, we substitute false for $\neg root(1)$ since the node 1 in L is a rooted node.
3. $\text{Val}(\Gamma_1, r_1) = \text{d} \geq \text{e}$
For this case, we change nothing.
4. $\text{Val}(\Gamma_2, r_2) = \text{outcon}(1) \neq 0$
In this case, we change outdeg(1) with outcon(1) + 0 because the outdegree of node 1 in L is 0.

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

Lemma 10. Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$, a host graph G, and an injectiva morphism $g : L^\alpha \to G$ for a label assignment α_L. For a graph condition c,

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

Proof. Let us consider the construction of Val(c) step by step. In step 1, we change terms x in c with $T(x)$. Here, we change functions s(e), t(e), l_V(v), m_V(v),

$\mathrm{l_E(e), m_E(e), l_V(s(e)), l_V(t(e)), m_V(s(e)), m_V(t(e))}$ for $e \in E_L$ and $v \in V_L$ with their values in L. Since $L^\alpha \to \rho_g(G)$ is an inclusion, then $s_L(e) = s_{\rho_g(G)}(e)$ and $t_L(e) = t_{\rho_g(G)}(e)$. Also, $(\ell_L(i))^\alpha = \ell_{\rho_g(G)}(i)$, $(m_L(i))^\alpha = m_{\rho_g(G)}(i)$ for all $i \in V_G$ and $i \in E_G$ such that the replacement does not change the satisfaction of c in $\rho_g(G)$. Then for function indeg(v) for $v \in V_L - V_K$, we change it to $indeg_L(x)$ due to the dangling condition, and for $v \in V_K$, we change it to $incon(v) + indeg_L(v)$ which is equivalent to $indeg_G(v) = indeg_{\rho_g(G)}(v)$ because $incon(v) = indeg_G(v) - indeg_L(v)$ (and analogously for outdeg(v)). In step 2, changing Boolean operators whose arguments are constants to their Boolean value clearly does not change the satisfaction in $\rho_g(G)$. Also, by the definition of morphism, $p_L(v) = p_{\rho_g(G)}(v)$ for all $v \in V_L$ so that the Boolean value of root(v) in L is equivalent to the Boolean value of root(v) in $\rho_g(G)$. Finally, in step 3, simplification clearly does not change satisfaction. □

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

Definition 33 (Transformation Lift). Given a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \to R\rangle$. Let c be a precondition. A left application condition w.r.t. c and w, denoted by $\mathrm{Lift}(c, w)$, is the condition over L:

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

□

Example 8 (Transformation Lift).

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

Proposition 2 (Left-application condition). Given a host graph G and a generalised rule $w = \langle r, ac_L, ac_R\rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \to R\rangle$. Let c be a precondition and α_L be a label assignment such that there exists an injective morphism $g : L^\alpha \to G$. 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 9, 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 $g : L^\alpha \to G$. The latter implies the satisfaction of the dangling condition. The satisfaction of the dangling condition implies $\rho_g(G) \vDash \mathrm{Dang}(r)$ based on Observation 1, such that $\rho_g(G) \vDash \mathrm{Split}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r)$, and $\rho_g(G) \vDash \mathrm{Val}(\mathrm{Split}(c, r) \wedge ac_L \wedge \mathrm{Dang}(r), r)^\alpha$ from Lemma 10. □

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

Observation 3 Given a host graph G and a generalised rule $w = \langle r^\alpha, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, and a precondition c. For every node/edge variable x in Lift(c,w), x does not represent any node/edge in L.

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

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

It is obvious that Γ does not have node and edge variable from its syntax. For Dang(r), it clearly only has one edge variable and there exists constraint $\bigwedge_{i \in E_L x \neq i}$ inside the existential quantifier for the variable. Finally for Split(c,r), since c is a closed formula, every node/edge variable must be bounded by existential quantifier, such that from Definition 31, the variable must be bounded by existential quantifier with constraint $\bigwedge_{i \in V_L x \neq i}$ or $\bigwedge_{i \in E_L x \neq i}$ inside. □

4.4 From left to right-application condition

To obtain a right-application condition from a left-application condition, we need to consider what properties could be different in the initial and the result graphs. Recall that in constructing a left-application condition, we evaluate all functions with a node/edge constant argument and change them with constant, including the constant incon(v) and outcon(v) when evaluating indeg(v) and

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

Definition 34 (Adjusment). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$ and a condition c over L. Let c' be a condition over L that is obtained from c by changing every term incon(x) (or outcon(x)) for $x \in V_K$ with indeg(x)$-indeg_R(x)$ (or outdeg(x)$-outdeg_R(x)$). Let also $\{v_1, \cdots, v_n\}$ and $\{e_1, \cdots, e_m\}$ denote the set of all nodes and edges in $R - K$ respectively. The *adjusted* condition of c w.r.t r, denoted by Adj(c,r), is a condition over R that is defined inductively, where c_1, c_2 are conditions over L:

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

□

Example 9.
Let p_1 denotes $\mathrm{Lift}(q_1, \texttt{del}^{\vee})$ and p_2 denotes $\mathrm{Lift}(q_2, \texttt{copy}^{\vee})$.
1. $\mathrm{Adj}(p_1, r_1) = \neg \exists_{\mathrm{E}} \mathrm{x}(\mathrm{x} \neq \mathrm{e1} \wedge \mathrm{s(x)} \neq 1 \wedge \mathrm{s(x)} \neq 2 \wedge \mathrm{m_V(s(x))} \neq \mathrm{none}) \wedge \mathrm{d} \geq \mathrm{e}$
2. $\mathrm{Adj}(p_2, r_2) = \exists_{\mathrm{V}} \mathrm{x}(\mathrm{x} \neq 1 \wedge \mathrm{x} \neq 2 \wedge \neg \mathrm{root(x)}) \wedge \mathrm{outdeg}(1) \neq 1$

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

Lemma 11. Given a host graph G, a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, 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 $g^* : R^\beta \rightarrow H$ where $\beta_R(i) = \alpha_L(i)$ for all common item i in domain β_R and $\alpha_L(i)$. Then,

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

Proof. Note that $\mathrm{Adj}(c, r)$ does not change any term representing label in c such that $\mathrm{Adj}(c^\alpha, r) \equiv \mathrm{Adj}(c, r)^\alpha$ for all label assignment α_L. Also, note that $\mathrm{Adj}(c, r)$ does not contain any variable x in R that does not exist in L. Hence, $\mathrm{Adj}(c, r)^\alpha = \mathrm{Adj}(c, r)^\beta$. Assuming $\rho_g(G) \vDash c^\alpha$ for $c = \mathrm{Lift}(c, w)$, we prove that $\rho_{g^*}(H) \vDash (\mathrm{Adj}(c, r))^\beta$ inductively bellow:
Base case.
1. If c is true or false, it is obvious that the lemma holds as every graph satisfies true and no graph satisfies false
2. If c is the predicate $\mathrm{int(x)}, \mathrm{char(x)}, \mathrm{string(x)}$ or $\mathrm{atom(x)}$ for a list variable x, $c' \equiv c$ and satisfaction of c is independent on the host graph such that $\rho_g(G) \vDash c^\alpha$ implies $\rho_{g^*}(H) \vDash c'^\alpha$ and $c'^\alpha = c'^\beta$.
3. If c is the predicate $\mathrm{root(x)}$ for some term x representing a node, then $x \notin V_L$ (see Definition 32 point 2), x is a variable representing $V_{\rho_g(G)} - (V_L) = V_{\rho_{g^*}(H)} - V_R$ (see Observation 3), or x is the function $\mathrm{s(x)}$ or $\mathrm{t(x)}$ for some

edge variable x representing an edge in $E_{\rho_g(G)} - E_L = E_{\rho_{g^*}(H)} - E_R$ (see Definition 32 point 1(b) and 3). Hence, x representing a node in $\rho_g(G) - L$, which is also in $\rho_{g^*}(H) - R$ so that if root(x) is true in $\rho_g(G)$, root(x) must be true in $\rho_{g^*}(H)$, and label assignment has nothing to do with this.

4. If $c = x_1 \ominus x_2$, if x_1 and x_2 are terms representing lists, then x_1 and x_2 independent to nodes and edges in V_L unless x_1 or x_2 is in the form incon(v)oroutcon(v) for some $v \in V_K$ (see Definition 32 point 1(b) and 3). However, because $outcon(v) = outdeg_{\rho_g(G)}(v) - outdeg_L(v) = outdeg_{\rho_{g^*}(H)}(v) - outdeg_R(v)$, then semantics of outcon(v) in $\rho_g(G)$ is equivalent to semantics of indeg(v)$-indeg_R(v)$ in $\rho_{g^*}(H)$. Hence, c is either independent to nodes and edges in V_L or contain outcon(x) or incon(x), $\rho_g(G) \vDash c$ implies $\rho_{g^*}(H) \vDash c' =Adj(c,r)$, or c. If c is $x_1 = x_2$ and x_1 or x_2 is a constant in $(V_L - V_K)$ or in E_L, it is obvious that there in no node/edge in $\rho_{g^*}(H)$ that is equal to the constant such that $\rho_{g^*}(H) \vDash$ false $=$Adj(c,r). Analogously, if c is $x_1 \neq x_2$ and x_1 or x_2 is a constant in $(V_L - V_K)$ or in E_L, every node/edge in $\rho_{g^*}(H)$ does not equal to the node or edge such that $\rho_{g^*}(H) \vDash$ true $=$Adj(c,r).

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

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

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

Lemma 12. Given 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 d. Let $g : L^\alpha \rightarrow R$ for some label assignment α_L be an injective morphism satisfying the dangling condition. Then

$$\rho_g(G) \vDash \text{Adj}(\text{Adj}(\text{Lift}(d,w),r),r^{-1})^\alpha \text{ if and only if } \rho_g(G) \vDash \text{Lift}(d,w)^\alpha$$

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

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

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

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

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

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

Definition 35 (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 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 10.

$$\begin{aligned}
\mathrm{Shift}(q_1, \mathtt{del}^{\vee}) = {} & \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{x} \neq \mathrm{e1} \wedge \mathrm{s(x)} \neq 1 \wedge \mathrm{s(x)} \neq 2 \wedge \mathrm{m_V(s(x))} \neq \mathrm{none}) \wedge \mathrm{d} \geq \mathrm{e} \\
& \wedge \mathrm{l_V}(1) = \mathrm{a} \wedge \mathrm{l_V}(2) = \mathrm{b} \wedge \mathrm{l_E(e1)} = \mathrm{d + e} \wedge \mathrm{m_V}(1) = \mathrm{red} \\
& \wedge \mathrm{m_V}(2) = \mathrm{none} \wedge \mathrm{m_E(e1)} = \mathrm{none} \wedge \mathrm{s(e1)} = 1 \wedge \mathrm{t(e1)} = 2 \\
& \wedge \neg\mathrm{root}(1) \wedge \neg\mathrm{root}(2) \wedge \mathrm{int(d)} \wedge \mathrm{int(e)} \\
\mathrm{Shift}(q_2, \mathtt{del}^{\vee}) = {} & \exists_{\mathrm{V}}\mathrm{x}(\mathrm{x} \neq 1 \wedge \mathrm{x} \neq 2 \wedge \neg\mathrm{root(x)}) \wedge \mathrm{outdeg}(1) \neq 1 \\
& \wedge \mathrm{l_V}(1) = \mathrm{a} \wedge \mathrm{l_V}(2) = \mathrm{a} \wedge \mathrm{l_E(e1)} = \mathrm{empty} \wedge \mathrm{m_V}(1) = \mathrm{none} \\
& \wedge \mathrm{m_V}(2) = \mathrm{none} \wedge \mathrm{m_E(e1)} = \mathrm{dashed} \wedge \mathrm{s(e1)} = 1 \wedge \mathrm{t(e1)} = 2 \\
& \wedge \neg\mathrm{root}(1) \wedge \mathrm{root}(2) \wedge \mathrm{indeg}(2) = 1 \wedge \mathrm{outdeg}(2) = 0
\end{aligned}$$

Proposition 3 (Shifting). 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 Adj$(\mathrm{Lift}(d, w)), r)^{\beta}$ is implied by Shift$(d, w)^{\beta}$, so now we show that Adj$(\mathrm{Lift}(d, w)), r)^{\beta}$ implies Shift$(d, w)^{\beta}$. That is, $ac_R^{\beta} \wedge \mathrm{Spec}(R)^{\beta} \wedge \mathrm{Dang}(r^{-1})^{\beta}$ is satisfied by $\rho_{g^*}(H)$. From Definition 25, $G \Rightarrow_{w,g,g^*} H$ implies $\rho_{g^*}(H) \vDash ac_R^{\beta}$. From the construction of Spec(R), Spec$(R)^{\beta}$ $\equiv$Spec(R^{β}) such that $\rho_{g^*}(H) \vDash \mathrm{Spec}(R)^{\beta}$ is implied by the injective morphism g^*. Finally, there is no label variable in Dang(r^{-1}) such that Dang(r^{-1}) $\equiv$Dang$(r^{-1})^{\beta}$, which is implied by $G \Rightarrow_{w,g,g^*} H$ because nodes in $R - K$ must not incident to any edge in $\rho_{g^*}(H) - R$ so that their indegree and outdegree in R represents their indegree and outdegree in $\rho_{g^*}(H)$. □

4.5 From right-application condition to postcondition

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

Definition 36 (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 Post(c, w), is the FO formula:

$$\mathrm{Post}(c, w) = \exists_{\mathrm{V}} x_1, \cdots, x_n(\exists_{\mathrm{E}} y_1, \cdots, y_m(\exists_{\mathrm{L}} z_1, \cdots, z_k(\mathrm{Var}(\mathrm{Shift}(c, w))))).$$

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 Var(Shift(c, w)). We then denote by Slp(c, r) the formula Post$(c, r^\vee)$, and Slp(c, r^{-1}) for the formula Post$(c, (r^\vee)^{-1})$. □

To obtain a closed FO 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.

Example 11.

$$
\begin{aligned}
\mathrm{Post}(q_1, \mathtt{del}^\vee) = {} & \exists_\mathrm{V}\mathrm{u}, \mathrm{v}(\mathrm{u} \neq \mathrm{v} \wedge \exists_\mathrm{E}\mathrm{w}(\exists_\mathrm{L}\mathrm{a}, \mathrm{b}, \mathrm{d}, \mathrm{e}(\\
& \neg\exists_\mathrm{E}\mathrm{x}(\mathrm{x} \neq \mathrm{w} \wedge \mathrm{s}(\mathrm{x}) \neq \mathrm{u} \wedge \mathrm{s}(\mathrm{x}) \neq \mathrm{v} \wedge \mathrm{m_V}(\mathrm{s}(\mathrm{w})) \neq \mathrm{none}) \wedge\ \mathrm{d} \geq \mathrm{e} \\
& \wedge \mathrm{l_V}(\mathrm{u}) = \mathrm{a} \wedge \mathrm{l_V}(\mathrm{v}) = \mathrm{b} \wedge \mathrm{l_E}(\mathrm{w}) = \mathrm{d} + \mathrm{e} \wedge \mathrm{m_V}(\mathrm{u}) = \mathrm{red} \\
& \wedge \mathrm{m_V}(\mathrm{v}) = \mathrm{none} \wedge \mathrm{m_E}(\mathrm{w}) = \mathrm{none} \wedge \mathrm{s}(\mathrm{w}) = \mathrm{u} \wedge \mathrm{t}(\mathrm{w}) = \mathrm{v} \\
& \wedge \neg\mathrm{root}(\mathrm{u}) \wedge \neg\mathrm{root}(\mathrm{v}) \wedge \mathrm{int}(\mathrm{d}) \wedge \mathrm{int}(\mathrm{e})))) \\
\mathrm{Post}(q_2, \mathtt{del}^\vee) = {} & \exists_\mathrm{V}\mathrm{u}, \mathrm{v}(\mathrm{u} \neq \mathrm{v} \wedge \exists_\mathrm{E}\mathrm{w}(\exists_\mathrm{L}\mathrm{a}(\\
& \exists_\mathrm{V}\mathrm{x}(\mathrm{x} \neq \mathrm{u} \wedge \mathrm{x} \neq \mathrm{v} \wedge \neg\mathrm{root}(\mathrm{x})) \wedge \mathrm{outdeg}(\mathrm{u}) \neq 1 \\
& \wedge \mathrm{l_V}(\mathrm{u}) = \mathrm{a} \wedge \mathrm{l_V}(\mathrm{v}) = \mathrm{a} \wedge \mathrm{l_E}(\mathrm{w}) = \mathrm{empty} \wedge \mathrm{m_V}(\mathrm{u}) = \mathrm{none} \\
& \wedge \mathrm{m_V}(\mathrm{v}) = \mathrm{none} \wedge \mathrm{m_E}(\mathrm{w}) = \mathrm{dashed} \wedge \mathrm{s}(\mathrm{w}) = \mathrm{u} \wedge \mathrm{t}(\mathrm{w}) = \mathrm{v} \\
& \wedge \neg\mathrm{root}(\mathrm{u}) \wedge \mathrm{root}(\mathrm{v}) \wedge \mathrm{indeg}(\mathrm{v}) = 1 \wedge \mathrm{outdeg}(\mathrm{v}) = 0)))
\end{aligned}
$$

Proposition 4 (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. From Lemma 3, $\rho_{g^*}(H) \vDash (\mathrm{Shift}(c, w))^\beta$ iff $H \vDash \mathrm{Var}(\mathrm{Shift}(c, w))^\beta$. 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 $\mathrm{Var}(\mathrm{Shift}(c, w)))^\beta$ so that there is no additional node (or edge) quantifier for $\mathrm{Var}(\mathrm{Shift}(c, w)))^\beta$. If there exists a node (or edge) in H, then from Lemma 6, adding an existential quantifier will not change its satisfaction on H. Hence, $H \vDash \mathrm{Var}(\mathrm{Shift}(c, w)))^\beta$ iff $H \vDash \mathrm{Post}(c, w)^\beta$. □

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 $H \vDash \mathrm{Post}(c, r^\vee)$, and showing that for all host graph H, $H \vDash \mathrm{Post}(c, r^\vee)$ 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, Post$(c, r^\vee)$ is a strongest liberal postcondition w.r.t. c and r.

Proof. From Lemma 1, $G \Rightarrow_r H$ iff $G \Rightarrow_{w,g,g*} H$ for some injective morphisms $g : L^\alpha \to G$ and $g^* : R^\beta \to H$ with label assignment α_L and β_R where $\beta_R(i) = \alpha_L(i)$ for every variable i in L such that i is in R, and for every node/edge i where $m_L(i) = m_R(i) = \texttt{any}$. From Proposition 2, Lemma 11, Proposition 3, and Proposition 4, $G \vDash c$ and $G \Rightarrow_{r^\vee,g,g^*} H$ implies $\rho_g(G) \vDash (\mathrm{Lift}(c, r^\vee))^\alpha$ implies $\rho_{g^*}(H) \vDash \mathrm{Shift}(c, r^\vee)^\beta$ implies $H \vDash \mathrm{Post}(c, r^\vee)$. Hence, $\mathrm{Post}(c, r^\vee)$ is a liberal postcondition w.r.t. c and r.

To show that $\mathrm{Post}(c, r^\vee)$ is a strongest liberal postcondition, based on Lemma 8, we need to show that for every graph H satisfying $\mathrm{Post}(c, r^\vee)$, there exists a host graph G satisfying c such that $G \Rightarrow_r H$.

Recall the construction of $\mathrm{Shift}(c, r^\vee)$. A graph satisfying $\mathrm{Shift}(c, r^\vee)$ must satisfying $\mathrm{Spec}(R)$ such that $H \vDash (\mathrm{Post}(c, r^\vee))$ implies $H \vDash (\mathrm{Post}(c, r^\vee))^\beta$ for some label assignment β_R, which implies $H \vDash \mathrm{Var}(\mathrm{Spec}(R))^\beta \equiv \mathrm{Var}(\mathrm{Spec}(R^\beta))$. From Lemma 3, this implies the existence of an injective morphism $g^* : R^\beta \to H$. From Proposition 4, $H \vDash \mathrm{Post}(c, r^\vee)^\beta$ implies $\rho_{g^*}(H) \vDash \mathrm{Shift}(c, r^\vee)^\beta$. From the construction of $\mathrm{Shift}(c, r^\vee)$, $\mathrm{Dang}(r^{-1})$ asserts that the dangling condition is satisfied by g^*. Hence, there exists a natural double-pushout bellow where every morphism is inclusion:

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

Since $\rho_{g^*}(H) \vDash \mathrm{Adj}(\mathrm{Lift}(c, r^\vee), r)^\beta$, from Lemma 11 this implies A satisfies $\mathrm{Adj}(\mathrm{Adj}(\mathrm{Lift}(c, r^\vee), r), r^{-1})^\alpha$. From Lemma 12, this implies $A \vDash c^\alpha$. Since direct derivations are invertible, $A \Rightarrow_{r^\vee,g,g^*} H$. Hence, $A \Rightarrow_r H$. $\square$

5 Proof Calculus

In this section, we introduce semantic and syntactic partial correctness calculus. The former consider arbitrary assertion language as pre- and postconditions, while the latter consider conditions (i.e. closed firs-order formulas) as the pre- and postconditions.

5.1 Semantic partial correctness calculus

For a graph program P and assertions c and d, triple $\{c\}\ P\ \{d\}$ is partially correct iff for all graph satisfying c, $H \in [\![P]\!]G$ implies $H \vDash d$.

Definition 37 (Partial correctness [?]). A graph program P is *partially correct* with respect to a precondition c and a postcondition d, denoted by $\vDash \{c\}\ P\ \{d\}$ if for every host graph G and every graph H in $[\![P]\!]G$, $G \models c$ implies $H \models d$. $\square$

To prove that $\vDash \{c\}\ P\ \{d\}$ holds for some assertions c, d, and a graph program P, we use two methods: 1) finding a strongest liberal postcondition w.r.t c and P and prove that the strongest liberal postcondition implies d, and 2) using proof rules for graph programs, create a proof tree to show the partial correctness. The first method has been done in classical programming [7, 11], while the second has been done in graph programming [17] but without the special command `break`.

In the previous section, we have defined a strongest liberal postcondition w.r.t. a precondition and a conditional rule schema. In this section, we extend the definition from conditional rule schemata to graph programs. In addition, we also introduce a weakest liberal precondition over a graph program.

Definition 38 (Strongest liberal postconditions). A condition d is a *liberal postcondition* w.r.t. a precondition c and a graph program P, if for all host graphs G and H,

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

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

Definition 39 (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 postcondition w.r.t. d and P. □

Lemma 13. Given a graph program P and a precondition c. Let d be a liberal postcondition w.r.t. c and P. Then d is a strongest liberal postcondition w.r.t. c and P if and only if for every graph H satisfying d, there exists a host graph G satisfying c such that $H \in [\![P]\!]G$.

Proof.
(If).
Assuming it is true that for every graph H satisfying d, there exists a host graph G satisfying c such that $H \in [\![P]\!]G$. Let H be a host graph satisfying d. From the assumption, there exists a graph G such that $G \vDash c$ and $H \in [\![P]\!]G$. Since $H \in [\![P]\!]G$, $H \vDash a$ for all liberal postcondition a over c and P. Hence, $H \vDash d$ implies $H \vDash a$ for all liberal postcondition a over c, P such that d is a strongest postcondition w.r.t. c and P
(Only if).
Assume that it is not true that for every host graph H, $H \vDash d$ implies there exists a host graph G satisfying c such that $H \in [\![P]\!]G$. We show that a graph

satisfying d can not imply the graph satisfying all liberal postcondition w.r.t r and P. From the assumption, there exists a host graph H such that every host graph G does not satisfy c or $H \notin [\![P]\!]G$. In the case of $H \notin [\![P]\!]G$, we clearly can not guarantee characteristic of H w.r.t. P. Then for the case where G does not satisfy c, we also can not guarantee the satisfaction of any liberal postcondition a over c in H because a is dependent of c. Hence, we can not guarantee that H satisfying all liberal postcondition w.r.t. r and c. $\square$

Lemma 14. Given a graph program P and a postcondition d. Let c be a liberal precondition w.r.t. P and d. Then c is a weakest liberal precondition w.r.t. P and d if and only if for every graph G $G \vDash c$ if and only if for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$

Proof. (If).
Suppose that $G \vDash c$ iff for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$. It implies for all host graphs H, $G \vDash c$ and $H \in [\![P]\!]G$ implies $H \vDash d$. From Definition 39, c is a liberal precondition. Let a be a liberal precondition w.r.t. P and d as well. From Definition 39, for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d$, and from the premise, $G \vDash c$. Hence, c is a weakest liberal precondition. (Only if).
Suppose that c is a weakest liberal precondition. From Definition 39, if $G \vDash c$ then $H \in [\![P]\!]G$ implies $H \vDash d$. Let a be a liberal precondition w.r.t P and d. From Definition 39, $G \vDash a$ implies for all H, $H \in [\![P]\!]G$ implies $H \vDash d$. Since for all a, $G \vDash a$ must imply $G \vDash c$, then $H \in [\![P]\!]G$ implies $H \vDash d$ must imply $G \vDash c$ as well. $\square$

SLP and WLP for a loop $P!$ is not easy to construct because $P!$ may get stuck or diverge. In [14], the divergence is represented by infinite formulas while in [11], it is represented by recursive equation that is not well-defined. In this paper, for practical reason we only consider strongest liberal postconditions over loop-free graph programs.

For the conditional commands $\texttt{if/try} - \texttt{then} - \texttt{else}$, the execution of the command depends on the existence of a proper host graph as a result of executing a graph program. In [17], there is an assertion representing a condition that must be satisfied by a graph such that there exists a path to successful execution, and there is also an assertion representing a condition that must be satisfied by a host graph such that there exist a path to a failure. Here, we define assertion SUCCESS for the former and FAIL for the latter.

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

$$G \vDash \text{SUCCESS}(P) \text{ if and only if there exists a host graph } H \text{ with } H \in [\![P]\!]G.$$

$\square$

Definition 41 (Assertion FAIL). Given a graph program P. FAIL(P) is the predicate on host graphs where for all host graph G,

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

$\square$

Note that for a graph program C, FAIL(C) does not necessarily equivalent to $\neg$SUCCESS(C), e.g. if $C = \{\texttt{nothing}, \texttt{add}\}; \texttt{zero}$ where `nothing` is the rule schema where the left and right-hand graphs are the empty graph, `add` is the rule schema where the left-hand graph is the empty graph and the right-hand graph is a single 0-labelled unmarked and unrooted node, and `zero` is a rule schema that match with the a 0-labelled unmarked and unrooted node. For a host graph G where there is no 0-labelled unmarked unrooted node, there is a derivation $\langle C, G\rangle \rightarrow^* H$ for some host graph H but also a derivation $\langle C, G\rangle \rightarrow^* \texttt{fail}$ such that $G \vDash$ SUCCESS(C) and $G \vDash$ FAIL(C).

Having a strongest liberal postcondition over a loop-free program P w.r.t a precondition c allows us to prove that the triple $\{c\}\, P\, \{d\}$ for an assertion d is partially correct. That is, by showing that d is implied by the strongest liberal postcondition.

Proposition 5 (Strongest liberal postcondition for loop-free programs). Given a precondition c and a loop-free program S. Then, the following holds:

1. If S is a set of rule schemata $\mathcal{R} = \{r_1, \cdots, r_n\}$,
   $$\text{SLP}(c, \mathcal{R}) = \begin{cases} \text{SLP}(c, r_1) \vee \cdots \vee \text{SLP}(c, r_n) & ,\text{, if } n > 0, \\ \text{false} & \text{, otherwise} \end{cases}$$
2. For loop-free programs C, P, and Q,
   (i) If $S = P \texttt{ or } Q$,
   $\text{SLP}(c, S) = \text{SLP}(c, P) \ \vee\ \text{SLP}(c, Q)$
   (ii) If $S = P; Q$,
   $\text{SLP}(c, S) = \text{SLP}(\text{SLP}(c, P), Q)$
   (iii) If $S = \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
   $\text{SLP}(c, S) = \text{SLP}(c \wedge \text{SUCCESS}(C), P) \vee \text{SLP}(c \wedge \text{FAIL}(C), Q)$
   (iv) If $S = \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
   $\text{SLP}(c, S) = \text{SLP}(c \wedge \text{SUCCESS}(C), C; P) \vee \text{SLP}(c \wedge \text{FAIL}(C), Q)$

Computing SLP$(c, \mathcal{R})$ for a set of rule schemata $\mathcal{R}$ is basically disjunct all strongest liberal postcondition w.r.t c and each rule schema in $\mathcal{R}$. If the rule set is empty, then SLP$(c, \mathcal{R})$ is false since there is nothing to disjunct. Computing SLP$(c, P; Q)$ is constructed by having SLP(c, P) and then find strongest liberal postcondition w.r.t. Q and the resulting formula.

The equation for program composition is the same with the equation for program composition in [7, 11]. However, for $\texttt{if} - \texttt{then} - \texttt{else}$ command, the command `if` in classical programming is followed by an assertion while in graph programs it is followed by a graph program. Hence, instead of checking the truth value of the assertion on the input graph, we check the check if the satisfaction of SUCCESS and FAIL of the associated program on the input graph. Then for `try − then − else` command, it does not exist in classical programming, but we have the equation for the command based on its similarity with `if − then − else`.

The execution of if/try commands yields two possibilities of results, so we need to check the strongest liberal postcondition for both possibilities and disjunct them. For the graph program $\texttt{if } C \texttt{ then } P \texttt{ else } Q$, P can be executed if SUCCESS(C) holds and Q can be executed if FAIL(C) holds. Similarly for $\texttt{try } C \texttt{ then } P \texttt{ else } Q$, $C;P$ can be executed if SUCCESS(C) holds and Q can be executed if FAIL(C) holds.

Proof (of Proposition 5). Here, we show that the proposition holds by induction on loop-free programs.
Base case.

1. If $S = \mathcal{R} = \{\}$,
   It is obvious that for all host graph G, $G \not\Rightarrow$ such that every condition is a liberal postcondition w.r.t. c and $\mathcal{R}$, and false is the strongest among all.
2. If $S = \mathcal{R} = \{r_1, \cdots, r_n\}$ where $n > 0$,
   (a) $$\begin{aligned} H \vDash \mathrm{SLP}(c, \mathcal{R}) &\overset{\text{L13}}{\Leftrightarrow} \exists G.G \Rightarrow_{\mathcal{R}} H \wedge G \vDash c \\ &\Leftrightarrow \exists G.(G \Rightarrow_{r_1} H \vee \cdots \vee G \Rightarrow_{r_n} H) \wedge G \vDash c \\ &\Leftrightarrow (\exists G.G \Rightarrow_{r_1} H \wedge G \vDash c) \vee \cdots \vee (\exists G.G \Rightarrow_{r_n} H \wedge G \vDash c) \\ &\overset{\text{L13}}{\Leftrightarrow} H \vDash \mathrm{SLP}(c, r_1) \vee \cdots \vee \mathrm{SLP}(c, r_n) \end{aligned}$$

Inductive case. Assume the proposition holds for loop-free programs C, P, and Q.

1. If $S = P \texttt{ or } Q$,
   $$\begin{aligned} H \vDash \mathrm{SLP}(c, S) &\overset{\text{L13}}{\Leftrightarrow} \exists G.H \in [\![P \texttt{ or } Q]\!]G \wedge G \vDash c \\ &\Leftrightarrow \exists G.(H \in [\![P]\!]G \vee H \in [\![Q]\!]G) \wedge G \vDash c \\ &\Leftrightarrow (\exists G.H \in [\![P]\!]G \wedge G \vDash c) \vee (\exists G.H \in [\![Q]\!]G \wedge G \vDash c) \\ &\overset{\text{L13}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c, P) \vee \mathrm{SLP}(c, Q) \end{aligned}$$
2. If $S = P; Q$,
   $$\begin{aligned} H \vDash \mathrm{SLP}(c, S) &\overset{\text{L13}}{\Leftrightarrow} \exists G.\ H \in [\![P;Q]\!]G \wedge G \vDash c \\ &\Leftrightarrow \exists G, G'.\ G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G' \wedge G \vDash c \\ &\overset{\text{L13}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{SLP}(c, P) \wedge H \in [\![Q]\!]G' \\ &\overset{\text{L13}}{\Leftrightarrow} H \vDash \mathrm{SLP}(\mathrm{SLP}(c, P), Q) \end{aligned}$$
3. If $S = \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
   $$\begin{aligned} & H \vDash \mathrm{SLP}(c, S) \\ &\overset{\text{L13}}{\Leftrightarrow} \exists G.\ G \vDash c \wedge H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G \\ &\Leftrightarrow \exists G.\ G \vDash c \wedge ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \\ &\Leftrightarrow (\exists G.\ G \vDash c \wedge \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (\exists G.\ G \vDash c \wedge \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G) \\ &\overset{\text{L13}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q) \end{aligned}$$

4. If $S = \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
$H \vDash \mathrm{SLP}(c, S)$
$\overset{\text{L13}}{\Leftrightarrow} \exists G.\ G \vDash c \wedge H \in [\![\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$
$\Leftrightarrow (\exists G, G'.\ G \vDash c \wedge G' \in [\![C]\!]G \wedge H \in [\![P]\!]G') \vee (\exists G.\ G \vDash c \wedge \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)$
$\overset{\text{L13}}{\Leftrightarrow} (\exists G'.\ G' \vDash \mathrm{SLP}(c, C) \wedge H \in [\![P]\!]G') \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$
$\overset{\text{L13}}{\Leftrightarrow} G \vDash \mathrm{SLP}(\mathrm{SLP}(c, C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$ □

To prove the triple $\{c\}\ P\ \{d\}$ is partially correct for a graph program P, we only need to show that $\mathrm{SLP}(c, P)$ implies d. However for graph programs P containing a loop, obtaining the assertion $\mathrm{SLP}(c, P)$ is not easy. Alternatively, we can create a proof tree (see Definition 42) with proof rules to show that $\{c\}\ P\ \{d\}$ is partially correct. Before we define the proof rules for partial correctness, we define predicate Break which shows relation between a graph program and assertions.

Definition 42 (Provability; proof tree[17]). Given an proof system I, a triple $\{c\}\ P\ \{d\}$ is provable in I, denoted by $\vdash_I \{c\}\ P\ \{d\}$, 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 $\vdash_I\ \{c\}\ P\ \{d\}$. If $\{c\}\ P\ \{d\}$ can be instantiated from the conclusion of an inference rule Y, and there are proof trees $T_1, \cdots, T_n$ with conclusions that are instances of the n premises of Y, then

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

is a proof tree, and $\vdash_I\ \{c\}\ P\ \{d\}$. □

Definition 43 (Predicate Break). Given a graph program P and assertions c and d. $\mathrm{Break}(c, P, d)$ is the predicate defined by:

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

□

Intuitively, when $\mathrm{Break}(c, P, d)$ holds, the execution of `break` that yields to termination of $P!$ will result a graph satisfying d.

Lemma 15. Given a graph program P with invariant c. If P does not contain the command `break`, then the following triple holds:

$$\{c\}\ P!\ \{c \wedge \mathrm{FAIL}(P)\}$$

Proof. If P does not contain the command `break`, then the derivation $\langle P, G\rangle \to^* \langle \texttt{break}, H\rangle$ must not exist for any host graphs G and H. Hence, $\text{Break}(c, P, d)$ is true for any c and d. Hence, $\text{Break}(c, P, \text{false})$ must be true. Since c in an invariant, $\{c\}\ P\ \{c\}$ is true. If $\langle P!, G\rangle \to^* H$ for some host graph H, from the semantics of graph programs, $\langle P!, G\rangle \to \langle P!, H\rangle \to^+ \texttt{fail}$. H must satisfy c because c is the invariant of P, and H must satisfy $\text{FAIL}(P)$ because $fail \in [\![P]\!]H$. Hence, the triple holds. $\square$

Definition 44 (Semantic partial correctness proof rules). The semantic partial correctness proof rules for core commands, denoted by SEM, is defined in Fig. 11, 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. $\square$

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

$$[\text{ruleapp}]_{\text{wlp}}\ \frac{}{\{\text{WLP}(r,d)\}\ r\ \{d\}}$$

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

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

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

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

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

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

Fig. 11: Calculus SEM of semantic partial correctness proof rules

The inference rule [ruleset] tells us about the application of a set of rule schemata $\mathcal{R}$. The rule set $\mathcal{R}$ is applied to a graph by nondeterministically choose an applicable rule schema from the set and apply it to the input graph. Hence, to derive a triple about $\mathcal{R}$, we need to prove the same triple for each rule schema inside $\mathcal{R}$.

The inference rule [comp] is similar to [comp] in traditional programming. In executing $P;Q$, the graph program Q is not executed until after the execution of P has terminated. So to show a triple about $P;Q$, we need to prove a triple about each P and Q and show that they are connected to some midpoint such that the midpoint is satisfied after the execution of P and before the execution of Q.

Like in conventional Hoare logic [1], the rule [cons] is aimed to strengthen the precondition and weaken the postcondition, or to replace the condition to another condition that semantically equivalent but syntactically different. To show that c' can be strengthened to c, we only need to show that c implies c', and to weaken d' to d, we need to show that d' implies d.

The assertions SUCCESS and FAIL are needed to prove a triple about if command. Recall that in the execution of `if` C `then` P `else` Q, the program C is first executed on a copy of G. If it terminates and yields a proper graph as a result, P is executed on G. If C terminates and results in a fail state, then Q is executed on G.

Similarly, for a triple about try command, we use the two assertions. But for `try` C `then` P `else` Q, C is not executed on a copy of G, but G itself. When the execution of C on G terminates and yields a proper graph, P is executed on the result graph. Hence, the difference with [if] is located in the first of the premises, where we use the sequential composition of C and P.

As in traditional programming, we need an invariant to show a triple about loop $P!$. When we have proven the existence of an invariant for P, the invariant will hold after any number of successful executions of P. If $P!$ terminates, from the semantics of "!" we know that the last execution of P either yields a fail state (see [Loop$_2$] of Fig. 4), such that FAIL(P) must hold, or executing the command `break` (see [Loop$_3$] of Fig. 4). In the former case, it is clear that the invariant and FAIL(P) must hold. Then in the latter case, we use Break(c, P, d) which is defined in Definition 43. The triple for loops is then captured by the rule [alap].

5.2 Syntactic partial correctness calculus

Section 5.1 introduces us to the semantic of partial correctness calculus. Now that we already have first-order formulas for some properties in graph programming, in this section we define the construction of SLP, SUCCESS, and FAIL in first-order formulas. In addition, we also define the syntactic version of partial correctness proof rules where possible (it will turn out that this is not always can be done). First, we define the first-order formula App(r) which should represent the first-order formula of SUCCESS(r).

Definition 45 (App(r)). Given a conditional rule schema $r : \langle L \leftarrow K \rightarrow R,\ \Gamma\rangle$. The formula App($r$) is defined as

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

□

The definition of Var(c), Spec(L), and Dang(r) for a condition c, rule graph L, and rule schema r, can be found in Definition 28, 27, and **??** respectively.

Lemma 16. Given a conditional rule schema $r : \langle L \leftarrow K \rightarrow R,\ \Gamma\rangle$, and a host graph G,

$$G \vDash \mathrm{SUCCESS}(r) \text{ if and only if } G \vDash \mathrm{App}(r).$$

Proof. (If).
$G \vDash \text{App}(r)$ implies $G \vDash \text{Var}(\text{Spec}(L))$, such that from Lemma 4, we know that there exists injective morphism $g : L^\alpha \to G$ for some label assignment α_L. Then from Lemma 3, $G \vDash \text{App}(r)$ implies $\rho_g(G) \vDash \text{Dang}(r)$ and $\rho_g(G) \vDash \Gamma^\alpha$. From Observation 1, $\rho_g(G) \vDash \text{Dang}(r)$ implies g satisfies the dangling condition, and $\rho_g(G) \vDash \Gamma^\alpha$ clearly implies $\Gamma^{\alpha,g}$ is satisfied by G. Hence, from the definition of conditional rule schema application, we know that $G \Rightarrow_{r,g} H$ for some host graph H such that $G \vDash \text{SUCCESS}(r)$.
(Only if).
$G \vDash \text{SUCCESS}(r)$ implies $G \Rightarrow H$ for some host graph H, which implies the existence of injective morphism $g : L^\alpha \to G$ for some label assignment α_L such that g satisfies the dangling condition and $G \vDash \Gamma^{\alpha,g}$. The existence of the injective morphism implies $G \vDash \text{Var}(\text{Spec}(L))$ from Lemma 4, the satisfaction of the dangling condition implies $\rho_g(G) \vDash \text{Dang}(r)$, and the $G \vDash \Gamma^{\alpha,g}$ implies $\rho_g(G) \vDash \Gamma$. Hence, $\rho_g(G) \vDash \text{Spec}(L)$ since $L^\alpha \to \rho_g(G)$ is inclusion (see Proposition 1). Hence, $\rho_g(G) \vDash \text{Spec}(L) \wedge \text{Dang}(r) \wedge \Gamma$ so that from Lemma 3, $G \vDash \text{App}(r)$. □

Defining a first-order formula for SUCCESS(r) with a rule schema r is easier than defining FO formula for SUCCESS(P) with an arbitrary loop-free program P. This is because we need to express properties of the initial graph after checking the existence of derivations. To determine the properties of the initial graph, we introduce the condition $\text{Pre}(P, c)$ for a postcondition c and a loop-free program P. Intuitively, $\text{Pre}(P, c)$ expresses the properties of the initial graph such that we can assert the existence of a host graph H such that $H \vDash c$ and $H \in [\![P]\!]G$. For an example, if there exists host graphs G' and H for a given host graph G and rule schemata r_1 and r_2 such that $G \Rightarrow_{r_1} G' \Rightarrow_{r_2} H$ and $H \vDash$ true (which also means that $G \vDash$ SUCCESS(P)), then G' should satisfy $\text{Pre}(r_2, \text{true})$ and G should satisfy $\text{Pre}(r_1, \text{Pre}(r_2, \text{true}))$ such that $\text{Pre}(r_1, \text{Pre}(r_2, \text{true}))$ can be considered as SUCCESS($r_1; r_2$) in first-order formula. For more general cases, see Definition 46. In the definition, $(r^\vee)^{-1}$ refers to the inverse of the generalised r (see Definition 26).

Definition 46 (Slp, Success, Fail, Pre of a loop-free program). Given a condition c and a loop-free program S. The first-order formulas $\text{Slp}(c, S)$, $\text{Pre}(c, S)$, $\text{Success}(S)$, and $\text{Fail}(S)$ are defined inductively:

1. If S is a set of rule schemata $\mathcal{R} = \{r_1, \cdots, r_n\}$,
   (a) $\text{Slp}(c, S) = \begin{cases} \text{Post}(c, r_1^\vee) \vee \cdots \vee \text{Post}(c, r_n^\vee) & \text{if } n > 0, \\ \text{false} & \text{otherwise} \end{cases}$
   (b) $\text{Pre}(S, c) = \begin{cases} \text{Post}(c, (r_1^\vee)^{-1}) \vee \cdots \vee \text{Post}(c, (r_n^\vee)^{-1}) & \text{if } n > 0, \\ \text{false} & \text{otherwise} \end{cases}$
   (c) $\text{Success}(S) = \begin{cases} \text{App}(r_1) \vee \cdots \vee \text{App}(r_n) & \text{if } n > 0, \\ \text{false} & \text{otherwise} \end{cases}$

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

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

□

For a precondition c and a loop-free program S, $\text{Slp}(c, S)$ is basically constructed based on Proposition 5. For $\text{Pre}(S, c)$, since we want to know the property of the initial graph based on c that is satisfied by the final graph and S, it works similar with constructing a weakest liberal precondition from a given postcondition and a program. Here we use [14] as a reference. However, in the reference the conditional part of `if` − `then` − `else` command contains an assertion instead of a graph program such that if C is an assertion, following their setting we will have $\text{Pre}(C, c) = C \implies \text{Pre}(P, c) \wedge \neg C \implies \text{Pre}(Q, c)$. The difference between assertions and graph programs as condition of a conditional program is, the satisfaction of the assertion on the initial graph implies that Q can not be executed, while in our case, $G \vDash \text{Success}(C)$ does not always imply that Q can not be executed. Hence, we change the equation to what we have in the definition above.

$\text{Success}(S)$ should express the existence of a proper graph as a final result, which means it should express the property of the initial graph based on S and the final graph satisfying true. This is exactly what $\text{Pre}(\text{true}, S)$ should express. Finally, $\text{Fail}(S)$ should express the property of the initial graph where failure is a result of the execution of S. Since we can yield failure anywhere is the subprogram of S, we need to disjunct all possibilities.

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

(a) $\text{Slp}(c, S)$ 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 $G \vDash \text{FAIL}(S)$

Proof. Here, we prove the theorem by induction on loop-free graph programs. Base case.

1. For $\mathcal{R} = \{\}$,
   (a) It is obvious that for all host graph G, $G \not\Rightarrow$ such that every condition is a liberal postcondition w.r.t. c and $\mathcal{R}$, and false is the strongest among all.
   (b) Statement (b) is valid because nothing satisfies false.
   (c) Both $G \vDash \text{Success}(\mathcal{R})$ and $G \vDash \text{SUCCESS}(\mathcal{R})$ always false such that $G \vDash \text{Success}(\mathcal{R})$ iff $G \vDash \text{SUCCESS}(\mathcal{R})$ holds.
   (d) Similarly, this point holds because both $G \vDash \text{Fail}(\mathcal{R})$ and $G \vDash \text{FAIL}(\mathcal{R})$ always true.
2. If $S = \mathcal{R} = \{r_1, \cdots, r_n\}$ where $n > 0$,
   (a) $H \vDash \text{SLP}(c, \mathcal{R}) \overset{\text{P5}}{\Leftrightarrow} H \vDash \text{SLP}(c, r_1) \vee \cdots \vee \text{SLP}(c, r_n)$
   $\overset{\text{T1}}{\Leftrightarrow} H \vDash \text{Post}(c, r_1^\vee) \vee \cdots \vee \text{Post}(c, r_n^\vee)$
   (b) $\exists H.H \in [\![\mathcal{R}]\!]G \wedge H \vDash c$
   $\Leftrightarrow \exists H.(G \Rightarrow_{r_1} H \vee \cdots \vee G \Rightarrow_{r_n} H) \wedge H \vDash c$
   $\Leftrightarrow (\exists H.G \Rightarrow_{r_1} H \wedge H \vDash c) \vee \cdots \vee (\exists H.G \Rightarrow_{r_n} H \wedge H \vDash c)$
   $\overset{\text{D25}}{\Leftrightarrow} (\exists H.G \Rightarrow_{r_1^\vee} H \wedge H \vDash c) \vee \cdots \vee (\exists H.G \Rightarrow_{r_n^\vee} H \wedge H \vDash c)$
   $\overset{\text{L2}}{\Leftrightarrow} (\exists H.H \Rightarrow_{(r_1^\vee)^{-1}} H \wedge H \vDash c) \vee \cdots \vee (\exists H.H \Rightarrow_{(r_n^\vee)^{-1}} G \wedge H \vDash c)$
   $\overset{\text{T1}}{\Leftrightarrow} G \vDash \text{Post}(c, (r_1^\vee)^{-1}) \vee \cdots \vee \text{Post}(c, (r_n^\vee)^{-1})$
   (c) $H \vDash \text{SUCCESS}(\mathcal{R}) \Leftrightarrow \exists H.H \in [\![\mathcal{R}]\!]G$
   $\Leftrightarrow \exists H.G \Rightarrow_{r_1} H \vee \cdots \vee G \Rightarrow_{r_n} H$
   $\Leftrightarrow (\exists H.G \Rightarrow_{r_1} H) \vee \cdots \vee (\exists H.G \Rightarrow_{r_n} H)$
   $\overset{\text{D40}}{\Leftrightarrow} G \vDash \text{SUCCESS}(r_1) \vee \cdots \vee \text{SUCCESS}(r_n)$
   $\overset{\text{L16}}{\Leftrightarrow} G \vDash \text{App}(r_1) \vee \cdots \vee \text{App}(r_n)$
   (d) $G \vDash \text{FAIL}(\mathcal{R}) \Leftrightarrow \text{fail} \in [\![\mathcal{R}]\!]G$
   $\Leftrightarrow (\neg\exists H.G \Rightarrow_{r_1} H) \wedge \cdots \wedge (\neg\exists H.G \Rightarrow_{r_n} H)$
   $\overset{\text{D40}}{\Leftrightarrow} G \vDash \neg(\text{SUCCESS}(r_1) \vee \cdots \wedge \text{SUCCESS}(r_n))$
   $\overset{\text{L16}}{\Leftrightarrow} G \vDash \neg(\text{App}(r_1) \vee \cdots \vee \text{App}(r_n))$

Inductive case. Assume (a), (b), (c), and (d) hold for loop-free programs C, P, and Q.

1. If $S = P$ `or` Q,
   (a) $H \vDash \text{SLP}(c, S) \overset{\text{P5}}{\Leftrightarrow} G \vDash \text{SLP}(c, P) \vee \text{SLP}(c, Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Slp}(c, P) \vee \text{Slp}(c, Q)$

(b) $\exists H. H \in [\![S]\!]G \wedge H \vDash c \Leftrightarrow \exists H.(H \in [\![P]\!]G \vee H \in [\![Q]\!]G) \wedge H \vDash c$

$\Leftrightarrow (\exists H. H \in [\![P]\!]G \wedge H \vDash c) \vee (\exists H. H \in [\![Q]\!]G \wedge H \vDash c)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, c) \vee \mathrm{Pre}(Q, c)$

(c) $G \vDash \mathrm{SUCCESS}(S) \Leftrightarrow \exists H. H \in [\![P \texttt{ or } Q]\!]G$

$\Leftrightarrow \exists H. H \in [\![P]\!]G \vee H \in [\![Q]\!]G$

$\overset{\text{D40}}{\Leftrightarrow} G \vDash \mathrm{SUCCESS}(P) \vee \mathrm{SUCCESS}(Q)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Success}(P) \vee \mathrm{Success}(Q)$

(d) $G \vDash \mathrm{FAIL}(S) \Leftrightarrow \mathrm{fail} \in [\![P \texttt{ or } Q]\!]G$

$\Leftrightarrow \mathrm{fail} \in [\![P]\!]G \vee \mathrm{fail} \in [\![Q]\!]G$

$\overset{\text{D41}}{\Leftrightarrow} G \vDash \mathrm{FAIL}(P) \vee \mathrm{FAIL}(Q)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Fail}(P) \vee \mathrm{Fail}(Q)$

2. If $S = P; Q$,

(a) $H \vDash \mathrm{SLP}(c, S) \overset{\text{P5}}{\Leftrightarrow} H \vDash \mathrm{SLP}(\mathrm{SLP}(c, P), Q)$

$\overset{\text{Ind.}}{\Leftrightarrow} H \vDash \mathrm{Slp}(\mathrm{Slp}(c, P), Q)$

(b) $\exists H. H \in [\![S]\!]G \wedge H \vDash c \Leftrightarrow \exists H, G'.\ G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G' \wedge H \vDash c$

$\overset{\text{Ind.}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{Pre}(Q, c) \wedge G' \in [\![P]\!]G$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, \mathrm{Pre}(Q, c))$

(c) $G \vDash \mathrm{SUCCESS}(S) \Leftrightarrow \exists H. H \in [\![P; Q]\!]G$

$\Leftrightarrow \exists H, G'.\ G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G'$

$\overset{\text{D40}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{SUCCESS}(Q) \wedge G' \in [\![P]\!]G$

$\overset{\text{Ind.}}{\Leftrightarrow} \exists G'.\ G' \vDash \mathrm{Success}(Q) \wedge G' \in [\![P]\!]G$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Pre}(P, \mathrm{Success}(Q))$

(d) $G \vDash \mathrm{FAIL}(S) \Leftrightarrow \mathrm{fail} \in [\![P; Q]\!]G$

$\Leftrightarrow \mathrm{fail} \in [\![P]\!]G \vee \exists H.\ H \in [\![P]\!]G \wedge \mathrm{fail} \in [\![Q]\!]H$

$\overset{\text{D41}}{\Leftrightarrow} G \vDash \mathrm{FAIL}(P) \vee \exists H.\ H \in [\![P]\!]G \wedge H \vDash \mathrm{FAIL}(Q)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Fail}(P) \vee \mathrm{Pre}(P, \mathrm{Fail}(Q))$

3. If $S = \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,

(a) $H \vDash \mathrm{SLP}(c, S)$

$\overset{\text{P5}}{\Leftrightarrow} G \vDash \mathrm{SLP}(c \wedge \mathrm{SUCCESS}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{FAIL}(C), Q)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \mathrm{Slp}(c \wedge \mathrm{Success}(C), P) \vee \mathrm{SLP}(c \wedge \mathrm{Fail}(C), Q)$

(b) $\exists H. H \in [\![S]\!]G \wedge H \vDash c$

$\Leftrightarrow \exists H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \wedge H \vDash c$

$\Leftrightarrow (\exists H.\ G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G \wedge H \vDash c)$
$\vee (\exists H.\ G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \wedge H \vDash c)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\mathrm{Success}(C) \wedge \mathrm{Pre}(P, c)) \vee (\mathrm{Fail}(C) \wedge \mathrm{Pre}(Q, c))$

(c) $G \vDash \mathrm{SUCCESS}(S)$

$\Leftrightarrow \exists H. H \in [\![S]\!]G$

$\Leftrightarrow \exists H.\ (G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)$

$\Leftrightarrow (\exists H.\ G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (\exists H.\ G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)))$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\mathrm{Success}(C) \wedge \mathrm{Success}(P)) \vee (\mathrm{Fail}(C) \wedge \mathrm{Success}(Q))$

(d) $G \vDash \mathrm{FAIL}(S)$

$\Leftrightarrow \mathrm{fail} \in [\![S]\!]G$

$\Leftrightarrow (G \vDash \mathrm{SUCCESS}(C) \wedge \mathrm{fail} \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge \mathrm{fail} \in [\![Q]\!]G)$

$\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\mathrm{Success}(C) \wedge \mathrm{Fail}(P)) \vee (\mathrm{Fail}(C) \wedge \mathrm{Fail}(Q))$

4. If $S = \texttt{try}\, C \,\texttt{then}\, P \,\texttt{else}\, Q$,
   (a) $H \vDash \text{SLP}(c, S)$
   $\overset{\text{P5}}{\Leftrightarrow} G \vDash \text{SLP}(\text{SLP}(c, C), P) \vee \text{SLP}(c \wedge \text{FAIL}(C), Q)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Slp}(\text{Slp}(c, C), P) \vee \text{Slp}(c \wedge \text{Fail}(C), Q)$
   (b) $\exists H. H \in [\![S]\!]G \wedge H \vDash c$
   $\Leftrightarrow (\exists H, G'.\ H \vDash c \wedge G' \in [\![C]\!]G \wedge H \in [\![P]\!]G') \vee (\exists H.\ H \vDash c \wedge \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
   $\overset{\text{Ind.}}{\Leftrightarrow} (\exists G'.\ G' \vDash \text{Pre}(P, c) \wedge G' \in [\![C]\!]G) \vee (\exists H.\ G \vDash \text{Fail}(C) \wedge H \in [\![Q]\!]G)) \wedge H \vDash c)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Pre}(C, \text{Pre}(P, c)) \vee (\text{Fail}(C) \wedge \text{Pre}(Q, c))$
   (c) $G \vDash \text{SUCCESS}(S)$
   $\Leftrightarrow \exists H. H \in [\![S]\!]G$
   $\Leftrightarrow (\exists H, G'.\ G' \in [\![C]\!]G \wedge H \in [\![P]\!]G') \vee (\exists H.\ \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
   $\overset{\text{D40}}{\Leftrightarrow} (\exists G'.\ G' \in [\![C]\!]G \wedge G' \vDash \text{SUCCESS}(P)) \vee (\exists H.\ \text{FAIL}(C) \wedge H \in [\![Q]\!]G)$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash \text{Pre}(C, \text{Success}(P)) \vee (\text{Fail}(C) \wedge \text{Success}(Q))$
   (d) $G \vDash \text{FAIL}(S)$
   $\Leftrightarrow \text{fail} \in [\![S]\!]G$
   $\Leftrightarrow (\exists G'.\ G' \in [\![C]\!]G \wedge \text{fail} \in [\![P]\!]G) \vee (G \vDash \text{FAIL}(C) \wedge \text{fail} \in [\![Q]\!]G)$
   $\overset{\text{D40,41}}{\Leftrightarrow} G \vDash (\text{SUCCESS}(C) \wedge \text{FAIL}(P)) \vee (\text{FAIL}(C) \wedge \text{FAIL}(Q))$
   $\overset{\text{Ind.}}{\Leftrightarrow} G \vDash (\text{Success}(C) \wedge \text{Fail}(P)) \vee (\text{Fail}(C) \wedge \text{Fail}(Q))$ $\square$

For any loop-free program P, we now can find the first order formula of SLP, SUCCESS, and FAIL. However, constructing SLP and SUCCESS of a loop is a challenging task because a loop may diverge. However, constructing a FO formula for FAIL of a graph program with loops is not as challenging if we only consider some forms of graph programs. In [2], Bak introduced a class of commands that cannot fail. Hence, we can always conclude that $\text{Fail}(P) = \text{false}$ if P is a command that cannot fail. Here, we introduce the class of non-failing commands.

Definition 47 (Non-failing commands). The class of *non-failing commands* is inductively defined as follows:
Base case:

1. `break` and `skip` are non-failing commands
2. Every call of a rule schema with the empty graph as its left-hand graph is a non-failing command
3. Every rule set call $\{r_1, \cdots, r_n\}$ for $n \geq 1$ where each r_i has the empty graph as its left-hand graph, is a non-failing command
4. Every command P! is a non-failing command

Inductive case:

1. $P; Q$ is a non-failing command if P and Q are non-failing commands
2. $\texttt{if}\, C \,\texttt{then}\, P \,\texttt{else}\, Q$ is a non-failing command if P and Q are non-failing commands

3. $\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$ is a non-failing command if P and Q are non-failing commands

$\square$

Recall the inference rule [alap] of SEM. To obtain a triple $\{c\}\ P!\ \{d\}$ for some precondition c, postcondition d, and a graph program $P!$, we need to find $\mathrm{Fail}(P)$. We now can construct $\mathrm{Fail}(P)$ if P is a loo-free program as in Definition 46, or if P is a non-failing command.

Now, let us consider P in the form $C; Q$. For any host graph G, $\mathrm{fail} \in [\![C; Q]\!]G$ iff $\mathrm{fail} \in [\![C]\!]G$ or $H \in [\![C]\!]G \wedge \mathrm{fail} \in [\![Q]\!]H$ for some host graph H, which means $G \vDash \mathrm{FAIL}(C) \vee (\mathrm{SUCCESS}(C) \wedge \mathrm{FAIL}(Q))$. We can construct both $\mathrm{Fail}(C)$ and $\mathrm{Success}(C)$ if C is a loop-free program, and we can construct $\mathrm{Fail}(Q)$ if Q is a loop-free program or a non-failing command. Here, we introduce the class of *iteration commands* which is the class of commands where we can obtain Fail of the commands.

Definition 48 (Iteration commands). The class of iteration commands is inductively defined as follows:

1. Every loop-free program is an iteration command
2. Every non-failing command is an iteration command
3. A command of the form $C; P$ is an iteration command if C is a loop-free program and P is an iteration command

$\square$

Definition 49 (Fail of iteration commands).
Given an iteration command S. Then,

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

$\square$

Theorem 3. Given an iteration command S. Then,

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

Proof. Here, we prove the theorem case by case.

1. It is obvious that if S is a non-failing command, then for any host graph G, $\mathrm{fail} \notin [\![S]\!]G$. Hence, there is no graph satisfying $\mathrm{FAIL}(S)$ such that $G \vDash \text{false}$ iff $G \vDash \mathrm{FAIL}(S)$ holds.
2. If S is a loop-free program, $G \vDash \mathrm{Fail}(S)$ iff $G \vDash \mathrm{FAIL}(S)$ holds based on Theorem 2.

3. If S is in the form $C; P$ for a loop-free program C and non-failing command P, then
$$\begin{aligned} G \vDash \mathrm{FAIL}(C;P) &\text{ iff fail} \in [\![C;P]\!]G \\ &\text{ iff fail} \in [\![C]\!]G \vee \exists G'.G' \in [\![C]\!]G \wedge \text{ fail} \in [\![P]\!]G' \\ &\text{ iff fail} \in [\![C]\!]G \\ &\text{ iff } G \vDash \mathrm{FAIL}(C) \\ &\text{ iff } G \vDash \mathrm{Fail}(C) \end{aligned}$$

Now let us consider the proof calculus SEM. There is the assertion SUCCESS(C) and FAIL(C) where C is the condition of a branching statement, and FAIL(S) for a loop body S. Since we are only able to construct Success(C) for a loop-free program C and FAIL(S) for an iteration command S, we do not define the syntactic version for arbitrary graph programs. Hence, we require a loop-free program as the condition of every branching statement and an iteration command as every loop body. For the axiom $[\text{ruleapp}]_{\text{wlp}}$, we follow the construction in [8] where a weakest liberal precondition can be constructed using the construction of Slp.

Definition 50 (Control programs). A *control command* is a command where the condition of every branching command is loop-free and every loop body is an iteration command. Similarly, a graph program is a *control program* if the condition of every branching command is loop-free and every loop body is an iteration command. □

Lemma 17. Given a conditional rule schema r and a closed first-order formula d. Let $\mathrm{Wlp}(r,d) = \neg\mathrm{Slp}(\neg d, r^{-1})$. Then for all host graphs G,

$$G \vDash \mathrm{Wlp}(r,d) \text{ if and only if } G \vDash \mathrm{WLP}(r,d).$$

Proof.
$$\begin{aligned} G \vDash \mathrm{Wlp}(r,d) &\text{ iff } G \vDash \neg\mathrm{Post}(\neg d, (r^{\vee})^{-1}) \\ &\text{ iff } \neg(\exists H, g, g^*. H \Rightarrow_{(r^{\vee})^{-1}, g^*, g} G \wedge H \vDash \neg d) && \text{(Lemma 8)} \\ &\text{ iff } \neg(\exists H, g, g^*. G \Rightarrow_{(r^{\vee}), g, g^*} H \wedge H \vDash \neg d) && \text{(Lemma 2)} \\ &\text{ iff } \neg(\exists H. G \Rightarrow_r H \wedge H \vDash \neg d) && \text{(Def. 25)} \\ &\text{ iff } \forall H. G \Rightarrow_r H \text{ implies } H \vDash d) && \text{(Def. implication)} \\ &\text{ iff } G \vDash \mathrm{WLP}(r,d) && \text{(Lemma 13)} \quad \square \end{aligned}$$

Now we know the FO formula for WLP(r, c), SLP(c, r), also SUCCESS(P) and FAIL(P) for some form of P. Finally, we define a syntactic partial correctness proof for control programs.

Definition 51 (Syntactic partial correctness proof rules). The syntactic partial correctness proof rules, denoted by SYN, is defined in Fig. 12, 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`. □

In the following section, we give an example of graph program verification using the calculus SYN we define above.

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

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

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

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

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

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

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

$$[\text{alap}] \; \frac{\{c\}\ S\ \{c\} \qquad \text{Break}(c, S, d)}{\{c\}\ S!\ \{(c \wedge \text{Fail}(S)) \vee d\}}$$

Fig. 12: Calculus SYN of syntactic partial correctness 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 partially correct. In addition, we also show the relative completeness of the proof calculi.

6.1 Soundness

To proof the soundness, we use structural induction on proof tree as defined in Definition 52.

Definition 52 (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. □

When we prove that a triple $\{c\}P\{d\}$ for assertions c, d and a graph program P is partially correct by showing that $\text{SLP}(c, P)$ implies d, it is obviously sound because of the definition of a strongest liberal postcondition itself. Then if c and d are first-order formulas and P is a loop-free program, showing that $\text{Slp}(c, P)$ implies d implies that $\{c\}P\{d\}$ is partially correct from Theorem 2. Then we also need to prove the soundness of proof calculus as summarised in Fig. 11 and Fig. 12.

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

$$\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\} \text{ implies } \vDash \{c\}\ P\ \{d\}.$$

Proof. To prove the soundness, we show that the implication holds for each axiom and inference rule in the proof rule w.r.t. the semantics of graph programs by structural induction on proof trees.

1. Base case :
   (a) $[\text{ruleapp}]_{\text{slp}}$. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs H, $H \vDash d$ iff $H \vDash \text{SLP}(c, r)$. Suppose that $G \vDash c$. From Definition 29, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash \{c\}\ P\ \{d\}$.
   (b) $[\text{ruleapp}]_{\text{slp}}$. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for a (conditional) rule schema r where for all graphs G, $G \vDash c$ iff $H \vDash WLP(r, c)$. Suppose that $G \vDash c$. From Definition 39, $G \Rightarrow_r H$ implies $H \vDash d$ so that $\vDash \{c\}\ P\ \{d\}$.
2. Inductive case.
   Assume that *Prop* holds for each premise of inference rules in Definition 44 for a set of rule schemata $\mathcal{R}$, assertions c, d, e, c', d', inv, host graphs G, G', H, H', and graph programs C, P, Q.
   (a) [ruleset]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \mathcal{R}\ \{d\}$ and $G \vDash c$. Since we can have a proof tree where $\{c\}\ \mathcal{R}\ \{d\}$ is the root, then $\vdash_{\mathsf{SEM}} \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From point 1, this means that $\vDash \{c\}\ r\ \{d\}$ for all $r \in \mathcal{R}$. From the semantics of graph programs, $H \in [\![R]\!]G$ iff $H \in [\![r]\!]G$ for some $r \in \mathcal{R}$. Since for any $r \in \mathcal{R}$, $H \in [\![r]\!]G$ implies $H \vDash d$, $H \in [\![\mathcal{R}]\!]G$ implies $H \vDash d$ as well so that $\vDash \{c\}\ \mathcal{R}\ \{d\}$.
   (b) [comp]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P; Q\ \{d\}$ and $G \vDash c$. $\vdash_{\mathsf{SEM}} \{c\}\ P; Q\ \{d\}$, implies $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{e\}$ and $\vdash_{\mathsf{SEM}} \{e\}\ Q\ \{d\}$. From the semantic of graph programs, $H \in [\![P; Q]\!]G$ iff there exists G' such that $G' \in [\![P]\!]G$ and $H \in [\![Q]\!]G'$. In addition to the assumption, $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{e\}$ implies $G' \vDash e$, and $\vdash_{\mathsf{SEM}} \{e\}\ Q\ \{d\}$ implies $H \vDash d$ so that $\vDash \{c\}\ P; Q\ \{d\}$.
   (c) [cons]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$ and $G \vDash c$. From the inference rule, we know that $\vdash \{c'\}\ P\ \{d'\}$, c implies c' (so that $G \vDash c'$), and d' implies d. From $\vdash \{c'\}\ P\ \{d'\}$, we get that for all host graphs H, $H \in [\![P]\!]G$ implies $H \vDash d'$ so that $H \vDash d$. Hence, $\vDash \{c\}\ P\ \{d\}$.
   (d) [if]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$. From $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$, we get $\vdash_{\mathsf{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ and $\vdash_{\mathsf{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From the former we know that for all host graphs H, if $G \vDash \text{SUCCESS}(C)$ and $H \in [\![P]\!]G$ then $H \vDash d$, while from the latter we know that for all host graphs H, if $G \vDash \text{FAIL}(C)$ and $H \in [\![Q]\!]G$ then $H \vDash d$. Recall that from the semantic of graph programs, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ iff $G \vDash \text{SUCCESS}(C) \wedge H \in [\![P]\!]G$ or $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$. Since both $G \vDash \text{SUCCESS}(C) \wedge H \in [\![P]\!]G$ and $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$ implies $H \vDash d$, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ implies $H \vDash d$ such that $\vDash \{c\}\ \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.
   (e) [try]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ and $G \vDash c$. $\vdash_{\mathsf{SEM}}$ $\{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$ implies $\vdash_{\mathsf{SEM}} \{c \wedge \text{SUCCESS}(C)\}\ C; P\ \{d\}$

and $\vdash_{\mathsf{SEM}} \{c \wedge \text{FAIL}(C)\}\ Q\ \{d\}$. From the former we know that for all host graphs H, if $G \vDash \text{SUCCESS}(C)$ and $H \in [\![C;P]\!]G$ then $H \vDash d$, while from the latter we know that for all host graphs H, if $G \vDash \text{FAIL}(C)$ and $H \in [\![Q]\!]G$ then $H \vDash d$. Recall that from the semantic of graph programs, $H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ iff $G \vDash \text{SUCCESS}(C) \wedge H \in [\![C;P]\!]G$ or $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$. Since both $G \vDash \text{SUCCESS}(C) \wedge H \in [\![C;P]\!]G$ and $G \vDash \text{FAIL}(C) \wedge H \in [\![Q]\!]G$ implies $H \vDash d$, $H \in [\![\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G$ implies $H \vDash d$ such that $\vDash \{c\}\ \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q\ \{d\}$.

(f) [alap]. Suppose that $\vdash_{\mathsf{SEM}} \{c\}\ P!\ \{d\}$ and $G \vDash c$. From $\vdash_{\mathsf{SEM}} \{c\}\ P!\ \{d\}$, we know that $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$ and $\text{Break}(c,P,d)$ holds. From $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$, we get that for all host graph H, $H \in [\![P]\!]G$ implies $H \vDash c$, while from Definition 43 and the true value of $\text{Break}(c,P,d)$ we know that for all hsot graphs H, $G \vDash c$ and $\langle P, G\rangle \to^* \langle \texttt{break;H}\rangle$ implies $H \vDash d$. From the semantic of graph programs, $H \in [\![P!]\!]G$ iff there exist derivation $\langle P, G\rangle \to^* \langle \texttt{break;H}\rangle$ or $\langle P!, G\rangle \to^* \langle P!, H\rangle$ and $\langle P!, H\rangle \to^+$ `fail`. The first case yields $H \vDash d$ because of $\text{Break}(c,P,d)$. Note that $\langle P!, G\rangle \to^* \langle P!, H\rangle$ sdone by having (probably) multiple execution of P on host graphs, so that from $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{c\}$ we know that $H \vDash c$. Then since $\langle P!, H\rangle \to^+ \texttt{fail}$, $H \vDash \text{FAIL}(P)$ so that $H \vDash c \wedge \text{FAIL}(C)$. Hence, $H \in [\![P!]\!]G$ implies $H \vDash d \vee (c \wedge \text{FAIL}(P))$ so that $\vDash \{c\}\ P!\ \{d\}$. □

Theorem 5 (Soundness of SYN). Let P be a restricted graph program i.e. graph programs where for every subprogram in the form $\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, $\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$, or $C!$, C is a loop-free program. Let also c and d be first-order formulas. Then,

$$\vdash_{\mathsf{SYN}} \{c\}\ P\ \{d\} \text{ implies } \vDash \{c\}\ P\ \{d\}.$$

Proof. The soundness of $[\text{ruleapp}]_{\text{slp}}$ follows from Theorem 1 and Theorem 4, while the soundness of $[\text{ruleapp}]_{\text{wlp}}$ follows from Theorem 1 and Lemma 17. The soundness of [ruleset], [comp], [cons], [if], and [try] follows from Theorem 4 and Theorem 2 about defining SUCCESS and FAIL in first-order formulas. Finally, the soundness of the inference rule [alap] follows from Theorem 4 and Theorem 3. □

6.2 Relative completeness

A proof calculus is complete when anytime we can denote a triple is valid according to partial correctness, then we can prove the correctness with the proof calculus. However, a completeness really depends on assertions we used because the ability to prove that d can be implied by c for some assertions c and d depends on language of the assertions. Hence, here we show the relative completeness instead of completeness, where we can separate the incompleteness due to the axioms and inference rules from any incompleteness in deducing valid assertions [4].

Before showing that SEM is relative complete, we first show that for any postcondition d and graph program P, we can show that $\vdash_{\mathsf{SEM}} \{\text{WLP}(P,d)\}\ P\ \{d\}$.

Lemma 18. Given a graph program S and a postcondition d. Then,

$$\vdash_{\mathsf{SEM}} \{\text{WLP}(S,d)\}\ S\ \{d\}$$

Proof. Here we prove the lemma by induction on graph programs.

Base case. If S is a (conditional) rule schema r,
$\vdash_{\mathsf{SEM}} \{\text{WLP}(S,d)\}\ S\ \{d\}$ automatically follows from the axiom $[\text{ruleapp}]_{\text{wlp}}$.
Inductive case.
Assume that for graph programs C, P, and Q, $\vdash_{\mathsf{SEM}} \{\text{WLP}(C,d)\}\ C\ \{d\}$, $\vdash_{\mathsf{SEM}} \{\text{WLP}(P,d)\}\ P\ \{d\}$, and $\vdash_{\mathsf{SEM}} \{\text{WLP}(Q,d)\}\ Q\ \{d\}$.

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

$$\begin{aligned}
&G \vDash \text{WLP}(\mathcal{R},d)\\
&\overset{\text{L14}}{\Leftrightarrow} \forall H. H \in [\![\mathcal{R}]\!]G \Rightarrow H \vDash d\\
&\Leftrightarrow \forall H.(H \in [\![r_1]\!]G \vee \cdots \vee H \in [\![r_n]\!]G) \Rightarrow H \vDash d\\
&\Leftrightarrow \forall H.(H \in [\![r_1]\!]G \Rightarrow H \vDash d) \wedge \cdots \wedge (H \in [\![r_n]\!]G \Rightarrow H \vDash d)\\
&\overset{\text{L14}}{\Leftrightarrow} G \vDash \text{WLP}(r_1,d) \wedge \cdots \wedge \text{WLP}(r_n,d)
\end{aligned}$$

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

$$\begin{aligned}
&G \vDash \text{WLP}(P;Q,d)\\
&\overset{\text{L14}}{\Leftrightarrow} \forall H. H \in [\![P;Q]\!]G \Rightarrow H \vDash d\\
&\Leftrightarrow \forall H, G'.(G' \in [\![P]\!]G \wedge H \in [\![Q]\!]G' \Rightarrow H \vDash d\\
&\Leftrightarrow \forall G'.(G' \in [\![P]\!]G \Rightarrow (\forall H. H \in [\![Q]\!]G' \Rightarrow H \vDash d)\\
&\overset{\text{L14}}{\Leftrightarrow} \forall G'.(G' \in [\![P]\!]G \Rightarrow G' \vDash \text{WLP}(Q,d)\\
&\overset{\text{L14}}{\Leftrightarrow} G \vDash \text{WLP}(P, \text{WLP}(Q,d))
\end{aligned}$$

(c) If $S = \texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,
Both $\vdash_{\mathsf{SEM}} \{\text{WLP}(P,d)\}\ P\ \{d\}$ and $\vdash_{\mathsf{SEM}} \{\text{WLP}(Q,d)\}\ Q\ \{d\}$ follow from the assumption. By [cons], we have:
$\vdash_{\mathsf{SEM}} \{\text{WLP}(P,d) \wedge (\text{FAIL}(C) \Rightarrow \text{WLP}(Q,d))\}\ P\ \{d\}$ and
$\vdash_{\mathsf{SEM}} \{\text{WLP}(Q,d) \wedge (\text{SUCCESS}(C) \Rightarrow \text{WLP}(P,d))\}\ Q\ \{d\}$.
Let e denotes $(\text{SUCCESS}(C) \Rightarrow \text{WLP}(P,d)) \wedge (\text{FAIL}(C) \Rightarrow \text{WLP}(Q,d))$ so that by [cons], we have:
$\vdash_{\mathsf{SEM}} \{e \wedge \text{SUCCESS}(C)\}\ P\ \{d\}$ and $\vdash_{\mathsf{SEM}} \{e \wedge \text{FAIL}(C)\}\ Q\ \{d\}$.

By [if] we then get that $\vdash_{\mathsf{SEM}} \{e\}\, S\, \{d\}$, and finally by [cons] we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S, d)\}\, S\, \{d\}$ because

$$\begin{aligned}
& G \vDash \mathrm{WLP}(\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q, d) \\
& \overset{\text{L14}}{\Leftrightarrow} \forall H.\ H \in [\![\texttt{if}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G \Rightarrow H \vDash d \\
& \Leftrightarrow \forall H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \Rightarrow H \vDash d \\
& \Leftrightarrow (\forall H.\ (G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![P]\!]G) \Rightarrow H \vDash d) \\
& \quad \wedge (\forall H.\ (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G) \Rightarrow H \vDash d) \\
& \Leftrightarrow G \vDash \mathrm{SUCCESS}(C) \Rightarrow (\forall H.\ H \in [\![P]\!]G \Rightarrow H \vDash d) \\
& \quad \wedge G \vDash \mathrm{FAIL}(C) \Rightarrow (\forall H.\ H \in [\![Q]\!]G \Rightarrow H \vDash d) \\
& \overset{\text{L14}}{\Leftrightarrow} G \vDash (\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(P, d) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q, d)
\end{aligned}$$

(d) If $S = \texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q$,

Let e denotes $\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(C; P, d) \wedge \mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q, d)$. Similar to point (c), from the assumption we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(Q, d)\}\, Q\, \{d\}$, which imply $\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{FAIL}(C)\}\, Q\, \{d\}$. Also from the assumption, we have both $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(C, \mathrm{WLP}(P, d))\}\, P\, \{\mathrm{WLP}(P, d)\}$ and also $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P, d)\}\, P\, \{d\}$. By [comp] and [cons] as case $S = P; Q$, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(C; P, d)\}\, C; P\, \{d\}$. Then by [cons] as in $\texttt{if} - \texttt{then} - \texttt{try}$ case, $\vdash_{\mathsf{SEM}} \{e \wedge \mathrm{SUCCESS}(C)\}\, C; P\, \{d\}$ such that by the inference rule [try] we have $\vdash_{\mathsf{SEM}} \{e\}\, S\, \{d\}$. Finnaly by [cons], $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(S, d)\}\, S\, \{d\}$ because

$$\begin{aligned}
& G \vDash \mathrm{WLP}(\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q, d) \\
& \overset{\text{L14}}{\Leftrightarrow} \forall H.\ H \in [\![\texttt{try}\, C\, \texttt{then}\, P\, \texttt{else}\, Q]\!]G \Rightarrow H \vDash d \\
& \Leftrightarrow \forall H.\ ((G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![C; P]\!]G) \vee (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G)) \\
& \qquad\qquad \Rightarrow H \vDash d \\
& \Leftrightarrow (\forall H.\ (G \vDash \mathrm{SUCCESS}(C) \wedge H \in [\![C; P]\!]G) \Rightarrow H \vDash d) \\
& \quad \wedge (\forall H.\ (G \vDash \mathrm{FAIL}(C) \wedge H \in [\![Q]\!]G) \Rightarrow H \vDash d) \\
& \Leftrightarrow G \vDash \mathrm{SUCCESS}(C) \Rightarrow (\forall H.\ H \in [\![C; P]\!]G \Rightarrow H \vDash d) \\
& \quad \wedge G \vDash \mathrm{FAIL}(C) \Rightarrow (\forall H.\ H \in [\![Q]\!]G \Rightarrow H \vDash d) \\
& \overset{\text{L14}}{\Leftrightarrow} G \vDash (\mathrm{SUCCESS}(C) \Rightarrow \mathrm{WLP}(C; P, d) \wedge (\mathrm{FAIL}(C) \Rightarrow \mathrm{WLP}(Q, d)
\end{aligned}$$

(d) If $S = P!$,

From the assumption, $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P, \mathrm{WLP}(P!, d))\}\, P\, \{\mathrm{WLP}(P!, d)\}$. By [cons] as in $P; Q$ case, we get $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P; P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$ such that by [cons] we know that $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$. Note that from Theorem 4, this implies $\vDash \{\mathrm{WLP}(P!, d)\}\, P\, \{\mathrm{WLP}(P!, d)\}$ such that for all host graphs $G_1, \cdots, G_n$, and H where $G_2 \in [\![P]\!]G_1, \cdots, G_n \in [\![P]\!]G_{n-1}$, and $\langle P, G_n\rangle \to^* \langle \texttt{break}, H\rangle$, $G \vDash \mathrm{WLP}(P!.d)$ implies $G' \vDash \mathrm{WLP}(P!.d)$ and $H \vDash d$. Hence, $\mathrm{Break}(\mathrm{WLP}(P!.d), P, d)$ holds. Then by the inference rule [alap], we have $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{(\mathrm{WLP}(P!, d) \wedge \mathrm{FAIL}(P)) \vee d\}$ such that by [cons], $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P!, d)\}\, P\, \{d\}$ because

$$\begin{aligned}
H \vDash \mathrm{WLP}(P!, d) \wedge \mathrm{FAIL}(P) & \overset{\text{L14}}{\Leftrightarrow} \mathrm{fail} \in [\![P]\!]H \wedge \forall H'.\ H' \in [\![P!]\!]H \Rightarrow H' \vDash d \\
& \Rightarrow H \in [\![P!]\!]H \wedge \forall H'.\ H' \in [\![P!]\!]H \Rightarrow H' \vDash d \\
& \Rightarrow H \vDash d.
\end{aligned}$$

□

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

$$\models \{c\}\ P\ \{d\} \text{ implies } \vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}.$$

Proof. From Lemma 18, we know that for all $\vdash_{\mathsf{SEM}} \{\mathrm{WLP}(P,d)\}\ P\ \{d\}$ and from Theorem 4, we get that $\mathrm{WLP}(P,d)$ is a weakest liberal precondition over P and d. Hence, if $\models \{c\}\ P\ \{d\}$, c must imply $\mathrm{WLP}(P,d)$ so that by [cons] we get that $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$. □

Conjecture 1. The proof calculus SYN is not relative complete.

In Theorem 6, we show the relative completeness of our semantic partial correctness calculus. This proof, however, assumes that the assertion language was expressive, i.e. able to express strongest liberal postcondition relative to arbitrary programs and preconditions. However, there are limitations in properties that can be expressed by first-order logic. We believe that FO logic can not express that a graph has an even number of nodes [12]. Although we do not have proof of the incompleteness of SYN, we believe that the calculus is not relative complete due to the expressiveness of FO formulas.

There is one example we have in mind that supports our hypothesis. That is when we have the following rule schema:

`Main = mark!; del!`

`mark(a : list)` `del(a : list)`

and consider a precondition expressing all nodes are unmarked and isolated ($c = \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V}(\mathrm{x}) = \mathrm{none} \vee \neg\exists_{\mathrm{E}}\mathrm{y}(\mathrm{s}(\mathrm{y}) = \mathrm{x} \vee \mathrm{t}(\mathrm{y}) = \mathrm{x}))$) and postcondition expressing the empty graph ($d = \forall_{\mathrm{V}}\mathrm{x}(\mathrm{false})$).

It is obvious that $\models \{c\}$ `mark!; del!!`$\{d\}$ holds because `mark!` doubled the nodes while marking the nodes with grey such that all nodes are grey and isolated, and we have an even number of grey nodes. Then `del!` deletes even number of grey nodes as many as it can, such that the resulting graph must be an empty graph.

However, SYN only can show that $\vdash \{c\}$ `mark!` $\{e\}$ where e expresses that all nodes are grey and isolated, without expressing that the number of the nodes are even. Hence, SYN can show $\vdash \{e\}$ `del!` $\{f\}$ where f expresses the graph is empty, or there exists one isolated grey node. From here, we cannot show that f implies d such that we cannot prove $\vdash \{c\}$ `mark!; del!` $\{d\}$.

7 Verification Example

In this section, we show an example of graph program verification with first-order logic. Here we consider the program `2colouring` that can be seen in Fig. 13.

Given a host graph where all nodes are unmarked and unrooted and all edges are unmarked. If the input graph is two-colourable, then all nodes in the resulting graph should marked with blue or red such that no two adjacent nodes have the same colour.

```
Main = (init; Colour!)!; if Illegal then unmark!
Colour = {col_blue, col_red}
Illegal = {ill_blue, ill_red}
```

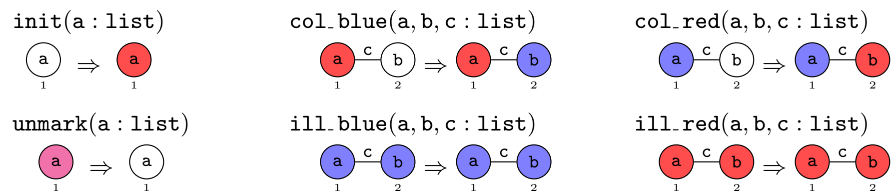


Fig. 13: Graph program `2colouring` for computing a 2-colouring graph

Then, let us consider the following pre- and postcondition:
Precondition "every node and edge is unmarked and every node is unrooted"
Postcondition "the precondition holds or every node is marked with blue or red, and no two adjacent nodes marked with the same colour"

Let c and $c \vee d$ be the FO formulas expressing pre- and postcondition respectively. We define c and d as follows:
$c \equiv \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x)} = \mathrm{none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x)} = \mathrm{none})$
$d \equiv \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x)} = \mathrm{red} \vee \mathrm{m_V(x)} = \mathrm{blue})) \wedge \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{s(x)} \neq \mathrm{t(x)} \wedge \mathrm{m_V(s(x))} = \mathrm{m_V(t(x))})$

A proof tree for the partial correctness for $\mathtt{2-colouring}$ with respect to c and $c \vee d$ is provided in Fig. 14. The conditions in the tree are defined in Table 4.

Note that there is no command `break` in the program, so $\mathrm{Break}(c, P, \mathrm{false})$ holds for any precondition c and sub-command P of the program `2colouring`. For this reason and for simplicity, we omit premise $\mathrm{Break}(c, P, \mathrm{false})$ in the inference rule [alap] of the proof tree.

As we can see in the proof tree of Fig. 14, we apply some inference rule [cons] which means we need to give proof of implications applied to the rules. Some implications are obvious, e.g. c implies $c \vee d$, so that for those obvious implications, we do not give any argument about them. Otherwise, we show that the implications hold:

1. *Proof of c implies f.*
$$\begin{aligned} G \vDash c \Leftrightarrow{} & G \vDash \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x)} = \mathrm{none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x)} = \mathrm{none}) \\ \Rightarrow{} & G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x)} = \mathrm{none} \vee \mathrm{m_V(x)} = \mathrm{blue} \vee \mathrm{m_V(x)} = \mathrm{red}) \wedge \neg\mathrm{root(x)}) \\ & \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x)} = \mathrm{none}) \end{aligned}$$

$$[\text{cons}]\,\dfrac{[\text{comp}]\,\dfrac{\text{Subtree I} \qquad \text{Subtree II}}{\{f\}\ \texttt{2colouring}\ \{c \vee d\}}}{\{c\}\ \texttt{2colouring}\ \{c \vee d\}}$$

where subtree I is:

$$[\text{cons}]\,\dfrac{[\text{alap}]\,\dfrac{[\text{comp}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{f\}\texttt{init}\{\text{Slp}(f,\texttt{init})\}}}{\{f\}\texttt{init}\{f\}} \qquad \text{subtree I.a}}{\{f\}\ \texttt{init};\texttt{Colour!}\ \{f\}}}{\{f\}\ (\texttt{init};\texttt{Colour!})!\ \{f \wedge \text{Fail}(\texttt{init};\texttt{Colour!})\}}}{\{f\}\ (\texttt{init};\texttt{Colour!})!\ \{e\}}$$

with subtree I.a:

$$[\text{cons}]\,\dfrac{[\text{alap}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{f\}\ \texttt{c\_blue}\ \{\text{Slp}(f,\texttt{c\_blue})\}}}{\{f\}\ \texttt{c\_blue}\ \{f\}} \qquad [\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{f\}\ \texttt{c\_red}\ \{\text{Slp}(f,\texttt{c\_red})\}}}{\{f\}\ \texttt{c\_red}\ \{f\}}}{\{f\}\ \texttt{Colour}\ \{f\}}}{\{f\}\ \texttt{Colour!}\ \{f \wedge \text{Fail}(\texttt{Colour})\}}}{\{f\}\ \texttt{Colour!}\ \{f\}}$$

and subtree II is:

$$[\text{if}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{alap}]\,\dfrac{[\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{f\}\ \texttt{unmark}\ \{\text{Slp}(f,\texttt{unmark})\}}}{\{f\}\ \texttt{unmark}\ \{f\}}}{\{f\}\ \texttt{unmark!}\ \{f \wedge \text{Fail}(\texttt{unmark})\}}}{\{e \wedge \text{Success}(\texttt{Illegal})\}\ \texttt{unmark!}\ \{c \vee d\}} \qquad [\text{cons}]\,\dfrac{[\text{ruleapp}]_{\text{slp}}\,\dfrac{}{\{d\}\ \texttt{skip}\ \{d\})}}{\{e \wedge \text{Fail}(\texttt{Illegal})\}\ \texttt{skip}\ \{c \vee d\}}}{\{e\}\ \texttt{if Illegal then umark!}\ \{c \vee d\}}$$

Fig. 14: Proof tree for partial correctness of `2colouring`

Table 4: Assertions inside proof tree of $2-$ `colouring`

symbol and its first-order formulas
$c \equiv \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$d \equiv \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue})) \wedge \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{s(x)} \neq \mathrm{t(x)} \wedge \mathrm{m_V(s(x)) = m_V(t(x))})$
$e \equiv \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue}) \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$f \equiv \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none})) \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\mathrm{Slp}(f, \texttt{init})$ $\equiv \exists_{\mathrm{V}}\mathrm{y}(\forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = y} \vee ((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}) \wedge \neg\mathrm{root(x)}))$ $\wedge\, \mathrm{m_V(y) = red} \wedge \neg\mathrm{root(y)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\mathrm{Slp}(f, \texttt{c\_blue}) = \mathrm{Slp}(f, \texttt{c\_red})$ $\equiv \exists_{\mathrm{V}}\mathrm{u, v}(\forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = u} \vee \mathrm{x = v} \vee ((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}) \wedge \neg\mathrm{root(x)}))$ $\wedge\, \mathrm{m_V(u) = red} \wedge \mathrm{m_V(v) = blue} \wedge \neg\mathrm{root(u)} \wedge \neg\mathrm{root(v)}$ $\wedge\, \exists_{\mathrm{E}}\mathrm{y}((\mathrm{s(y) = u} \wedge \mathrm{t(y) = v}) \vee (\mathrm{t(y) = u} \wedge \mathrm{s(y) = v}))) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\mathrm{Slp}(f, \texttt{unmark})$ $\equiv \exists_{\mathrm{V}}\mathrm{y}(\forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = y} \vee ((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}) \wedge \neg\mathrm{root(x)}))$ $\wedge\, \mathrm{m_V(y) = none} \wedge \neg\mathrm{root(y)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\mathrm{Fail}(\texttt{Colour})$ $\equiv \neg\exists_{\mathrm{E}}\mathrm{x}((((\mathrm{m_V(s(x)) = red} \vee \mathrm{m_V(s(x)) = blue}) \wedge \mathrm{m_V(t(x)) = none})$ $\vee\, ((\mathrm{m_V(t(x)) = red} \vee \mathrm{m_V(t(x)) = blue}) \wedge \mathrm{m_V(s(x)) = none}))$ $\wedge\, \neg\mathrm{root(s(x))} \wedge \neg\mathrm{root(t(x))})$
$\mathrm{Fail}(\texttt{init}; \texttt{Colour!}) \equiv \neg\exists_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)})$
$\mathrm{Fail}(\texttt{unmark}) \equiv \neg\exists_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x)} \neq \mathrm{none} \wedge \neg\mathrm{root(x)})$
$\mathrm{Fail}(\texttt{Illegal})$ $\equiv \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{s(x)} \neq \mathrm{t(x)}$ $\wedge\, ((\mathrm{m_V(s(x)) = red} \wedge \mathrm{m_V(t(x)) = red}) \vee (\mathrm{m_V(s(x)) = blue} \wedge \mathrm{m_V(t(x)) = blue})))$
$\mathrm{Success}(\texttt{Illegal})$ $\equiv \exists_{\mathrm{E}}\mathrm{x}(\mathrm{s(x)} \neq \mathrm{t(x)}$ $\wedge\, ((\mathrm{m_V(s(x)) = red} \wedge \mathrm{m_V(t(x)) = red}) \vee (\mathrm{m_V(s(x)) = blue} \wedge \mathrm{m_V(t(x)) = blue})))$

2. *Proof of* $\mathrm{Slp}(f, \texttt{init})$ *implies* f.

$$
\begin{aligned}
&G \vDash \mathrm{Slp}(f, \texttt{init})\\
&\Leftrightarrow G \vDash \exists_{\mathrm{V}}\mathrm{y}(\forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = y} \vee (\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}))\\
&\text{‘} \qquad\qquad \wedge\, \mathrm{m_V(y) = red} \wedge \neg\mathrm{root(y)})\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})\\
&\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \wedge \neg\mathrm{root(x)})\\
&\qquad\qquad \vee (\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}))\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})\\
&\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})
\end{aligned}
$$

3. *Proof of* $\mathrm{Slp}(f, \texttt{c\_blue})$ *implies* f.

$$
\begin{aligned}
&G \vDash \mathrm{Slp}(f, \texttt{c\_blue})\\
&\Leftrightarrow G \vDash \exists_{\mathrm{V}}\mathrm{u, v}(\mathrm{u} \neq \mathrm{v} \wedge \forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = u} \vee \mathrm{x = v}\\
&\qquad\qquad\qquad \vee ((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)}))\\
&\qquad\qquad \wedge\, \mathrm{m_V(u) = red} \wedge \mathrm{m_V(v) = blue} \wedge \neg\mathrm{root(u)} \wedge \neg\mathrm{root(v)}\\
&\qquad\qquad \wedge\, \exists_{\mathrm{E}}\mathrm{y}((\mathrm{s(y) = u} \wedge \mathrm{t(y) = v}) \vee (\mathrm{t(y) = u} \wedge \mathrm{s(y) = v})))\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})\\
&\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \wedge \neg\mathrm{root(x)}) \vee (\mathrm{m_V(x) = blue} \wedge \neg\mathrm{root(x)})\\
&\qquad\qquad \vee (\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})\\
&\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})\\
&\qquad\quad \wedge\, \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})
\end{aligned}
$$

4. *Proof of* Slp(f, `col_red`) *implies* f.
$G \vDash \text{Slp}(f, \texttt{c\_red})$
$\Leftrightarrow G \vDash \text{Slp}(f, \texttt{c\_blue})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})$
$\wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$

5. *Proof of* Slp(f, `unmark`) *implies* f.
$G \vDash \text{Slp}(f, \texttt{unmark})$
$\Leftrightarrow G \vDash \exists_{\mathrm{V}}\mathrm{y}(\forall_{\mathrm{V}}\mathrm{x}(\mathrm{x = y} \vee ((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}) \wedge \neg\mathrm{root(x)}))$
$\wedge\, \mathrm{m_V(y) = none} \wedge \neg\mathrm{root(y)})$
$\wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)})$
$\vee ((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = none}) \wedge \neg\mathrm{root(x)}))$
$\wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})$
$\wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$

6. *Proof of* $f \wedge$Fail(`init`; `Colour!`) *implies* e.
$G \vDash f \wedge \text{Fail}(\texttt{init}; \texttt{Colour!})$
$\Leftrightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})$
$\wedge \neg\exists_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$

7. *Proof of* $f \wedge$Fail(`unmark`) *implies* $c \vee d$.
$G \vDash f \wedge \text{Fail}(\texttt{unmark})$
$\Leftrightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})$
$\wedge \neg\exists_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) \neq none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash (\forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = none} \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})) \vee d$

8. *Proof of* $e \wedge$Fail(`Illegal`) *implies* d.
$G \vDash e \wedge \text{Fail}(\texttt{Illegal})$
$\Leftrightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\wedge \neg\exists_{\mathrm{E}}\mathrm{x}(((\mathrm{m_V(s(x)) = red} \wedge \mathrm{m_V(t(x)) = red})$
$\vee (\mathrm{m_V(s(x)) = blue} \wedge \mathrm{m_V(t(x)) = blue})) \wedge \mathrm{s(x) \neq t(x)})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}(\mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red})$
$\wedge \neg\exists_{\mathrm{E}}\mathrm{x}(((\mathrm{m_V(s(x)) = red} \wedge \mathrm{m_V(t(x)) = red})$
$\vee (\mathrm{m_V(s(x)) = blue} \wedge \mathrm{m_V(t(x)) = blue})) \wedge \mathrm{s(x) \neq t(x)})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = red} \vee \mathrm{m_V(x) = blue}))$
$\wedge \neg\exists_{\mathrm{E}}\mathrm{x}(\mathrm{m_V(s(x)) = m_V(t(x))} \wedge \mathrm{s(x) \neq t(x)})$

9. *Proof of* $e \wedge$Success(`Illegal`) *implies* f.
$G \vDash e \wedge \text{Success}(\texttt{Illegal})$
$\Rightarrow G \vDash e$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)}) \wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$
$\Rightarrow G \vDash \forall_{\mathrm{V}}\mathrm{x}((\mathrm{m_V(x) = none} \vee \mathrm{m_V(x) = blue} \vee \mathrm{m_V(x) = red}) \wedge \neg\mathrm{root(x)})$
$\wedge \forall_{\mathrm{E}}\mathrm{x}(\mathrm{m_E(x) = none})$

8 Related Work

Graph program verification using Hoare style verification with first-order logic has been done in [18, 17]. However, they do not consider the command `break` and the rootedness of nodes. Also, their calculus only able to proof graph programs where the conditions of every branching command and every loop-body is a rule-set call. Here, we introduce a calculus that is able to handle more form of graph programs. That is, graph programs where the conditions of every branching command is a loop-free program, and every loop body is an iteration command. This allows us to verify graph programs with nested loops, which are used a lot in practice.

In addition, our first-order formula is more natural such that it can be easier to understand by the general reader. The ability to specify marks in the text is also an advantage to express a property in a straightforward way. For an example, if we want to express "all nodes are unmarked", we can express it as $\forall_{\mathrm{V}}\mathrm{x}(\mathrm{m}_{\mathrm{V}} = \mathrm{none})$ by our first-order formula. However in [18, 17], the simplest way to express it is: $\neg\exists(\text{a})\wedge\neg\exists(\text{a})\wedge\neg\exists(\text{a})\wedge\neg\exists(\text{a})$.

Expressing Success(P), Wlp(P, c), and Slp(c, P) for a condition c and a graph program P with loops remains as an open problem in this paper. In [14], they have an equation of Wlp$(c, P!)$ by using an infinite formula. Here, we do not use a similar trick for a practical reason. In [7, 11], they have properties for SLP$(c, P!)$ and WLP$(c, P!)$ but they also cannot have a syntactic equation to construct Slp$(c, P!)$ and Wlp$(c, P!)$. Alternatively, they construct an approximate strongest liberal postcondition over a loop, which results in conjunction of invariant and Fail(P) for a loop-body P. This is similar to our inference rule [alap], but in our case, we need to consider the command `break` that may exist in a loop body.

9 Conclusion and Future Work

We have shown the construction of a strongest liberal postcondition for a given precondition and a conditional rule schema. From this construction, we have shown that we can obtain a strongest liberal postcondition over a loop-free program, and construct a first-order formula for SUCCESS(C) for a loop-free program C. Moreover, we can construct a first-order formula for FAIL(P) for an iteration command P such that we have a proof calculus that can handle a nested loop in some forms.

However, the expressiveness of a first-order formula is limited. Hence, in the near future, we will extend our formulas to monadic second-order formulas to overcome some limitation in the expressiveness of the formulas. Since there is a lot of literature on standard logic, especially in monadic second-order formulas, we also have possibilities to extend our approach based on literature to overcome some limitation we may have. For example, the current approach in graph program verification has a limitation in expressing isomorphism between initial and final graph [19]. In standard logic, monadic second-order transduction can be used to express the connection between initial and final state [5] by expressing the final state by elements in the initial state. For a future goal, we aim to be able to use our logic to overcome the limitation by using monadic second-order transductions.

References

1. K. R. Apt and E.-R. Olderog. Fifty years of hoare's logic, 2019.
2. C. Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015.
3. G. Campbell. Efficient graph rewriting. *CoRR*, abs/1906.05170, 2019.
4. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.
5. 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.
6. 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.
7. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990.
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. 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.
11. C. B. Jones, A. Roscoe, and K. R. Wood. *Reflections on the Work of C.A.R. Hoare*. Springer Publishing Company, Incorporated, 1st edition, 2010.
12. L. Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
13. T. Nipkow. *Hoare Logics in Isabelle/HOL*, pages 341–367. Springer Netherlands, Dordrecht, 2002.
14. K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2009.
15. 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.
16. 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.
17. C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013.
18. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
19. G. S. Wulandari and D. Plump. Verifying a copying garbage collector in GP 2. In *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, pages 479–494, 2018.