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

Various Hoare-style proof systems for the graph programming language GP 2 have been developed by Poskitt and Plump, see for example [16,14]. These calculi use so-called E-conditions as assertions which extend nested graph conditions [11] with support for expressions. However, a drawback of E-conditions and nested graph conditions is that they are not easy to understand by average programmers who are typically used to write formal specifications in standard first-order logic. To give a simple example, the following E-condition expresses that every node is labelled by an integer: $\forall($ $, \exists($ $\mid \texttt{int(a)})) \wedge \forall($ $, \exists($  $\mid \texttt{int(a)})) \wedge \forall($ $, \exists($ $\mid \texttt{int(a)})) \wedge \forall($ $, \exists($ $\mid \texttt{int(a)})) \wedge \forall($ $, \exists($ $\mid \texttt{int(a)}))$. Having to write two quantifiers that refer to the *same* object appears unnatural from the perspective of standard predicate logic where a single universal quantifier would suffice. In the logic we introduce in this paper, the above condition is simply written as $\forall_{\mathsf{V}} x(\mathsf{int}(\mathsf{l}_{\mathsf{V}}(x)))$. In addition, the E-condition in this example is lengthy as the property has to be repeated for each of GP 2's node marks.

In this paper we use assertions which are conventional first-order formulas enriched with GP 2 expressions. We believe that these assertions are easier to comprehend by programmers than E-conditions and also offer the prospect of reusing the large range of tools available for first-order logic.

* Supported by Indonesia Endowment Fund for Education (LPDP)

To use our assertions in Hoare-style verification, we show how to construct a strongest liberal postcondition $\mathrm{Slp}(c, r)$ for a given rule schema r and a precondition c. Based on this construction, we are able to construct a strongest liberal postcondition for any loop-free graph programs and preconditions. In addition, we are able to give syntactic conditions on host graphs which for any loop-free program express successful execution resp. the existence of a failing execution. With these results we obtain a verification calculus that can handle considerably more programs than the calculi in [16,14]. In particular, many programs with nested loops can now be formally verified, which has been impossible so far.

Nevertheless, our proof calculus is not relatively complete because first-order logic is not powerful enough to express all necessary assertions. Therefore we present a semantic version of the calculus which turns out to be relatively complete. The space available for this paper does not allow us to present all technical details or the proofs of our results. These can be found in the long version [18].

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

2 The Graph Programming Language GP 2

In this section, we briefly review the graph programming language GP 2 which was introduced in [12].

2.1 GP 2 graphs

A graph in GP 2 is a system comprising a finite set of nodes, a finite set of edges, labelling functions, and a rootedness function. A label is a pair consisting of an expression and a mark.

Definition 1 (Graphs in GP 2). Let $\mathbb{L}$ be the set of lists whose entries are integers and character strings, as defined by the grammar of Fig. 5a in the Appendix. Let $\mathbb{M}_V = \{\texttt{none}, \texttt{red}, \texttt{blue}, \texttt{green}, \texttt{grey}\}$ be the set of *node marks* and $\mathbb{M}_E = \{\texttt{none}, \texttt{red}, \texttt{blue}, \texttt{green}, \texttt{dashed}\}$ be the set of *edge marks*. Moreover, let $\mathbb{E}$ be the set of *expressions* defined by grammar Fig. 5b. Note that expressions may contain variables of various types.

A rule graph 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 $l_G = \langle \ell_G^V, m_G^V \rangle$ where $\ell_G^V : V_G \to \mathbb{E}$ and $m_G^V : V_G \to \mathbb{M}_V \cup \{\texttt{any}\}$, an edge labelling function $m_G = \langle \ell_G^E, m_G^E \rangle$ $\ell_G^E : E_G \to \mathbb{E}$ and $m_G^E : E_G \to \mathbb{M}_E \cup \{\texttt{any}\}$, and a partial rootedness function $p_G : V_G \to \{0, 1\}$. A *totally labelled rule graph* is a rule graph where all of its functions are total.

A *host graph* is a totally labelled rule graph with a node labelling function $l_G = \langle \ell_G^V, m_G^V \rangle$ where $\ell_G^V : V_G \to \mathbb{L}$ and $m_G^V : V_G \to \mathbb{M}_V$, and an edge labelling function $m_G = \langle \ell_G^E, m_G^E \rangle$ where $\ell_G^E : E_G \to \mathbb{L}$ and $m_G^E : E_G \to \mathbb{M}_E$. □

Node labelling and rootedness functions are partial to allow relabelling of nodes [2]. Partial node labelling function here means that for all node (or edge) x in G, $\ell_G^V(x)$ (or $\ell_G^E(x)$) is defined if and only if $m_G^V(x)$ (or $m_G^E(x)$) is defined. In a host graph, a list consists of (list of) integers and strings which have five type, that are: `list`, `atom`, `int`, `string`, and `char`. The domain of each type (respectively) is $\mathbb{Z} \cup \text{Char}^*)^*, \mathbb{Z} \cup \text{Char}^*, \mathbb{Z}$, $\{\text{Char}\}^*$, and Char, and they have hierarchical type system as usual (see Fig. 6).

As in traditional graph transformation, we use graph morphism to show connection between two graphs. Here, we use definition of graph morphisms and premorphisms as in [2], where we only consider the preservation of defined labels and rootedness (so that undefined labels and rootedness are always preserved). Injective (or surjective) graph morphisms and inclusions are defined as usual. A graph morphism g is an isomorphism if g is both injective and surjective, but every item with undefined label (or rootedness) can only be mapped with an item with undefined label (or rootedness). 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.

2.2 Conditional rule schemata

As usual in double-pushout approach, rules in GP 2 (called rule schemata) consists of a left-hand graph, an interface graph, and a right-hand graph. In addition, GP 2 also allows a condition for the left-hand graph.

Definition 2 (Rule schemata; conditional rule schemata). A *rule schema* $r = \langle L \leftarrow K \to R \rangle$ comprises totally labelled rule graphs L and R, a graph K containing only unlabelled nodes with undefined rootedness, and inclusions $K \to L$ and $K \to R$. All list expressions in L are simple (i.e. no arithmetic operators, at most one occurrence of a list variable, and at most one occurrence of a string variable in each occurrence of a string sub-expression). 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. A *conditional rule schema* is a pair $\langle r, \Gamma \rangle$ with r a rule schema and Γ a condition (see Fig. 7) where 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 3 (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 $\alpha_{\mathbb{L}} : X \to \mathbb{L}$ 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 $\mu_{\mathbb{M}} = \langle \mu_V : V_L \to \mathbb{M}_V \backslash \{\mathtt{none}\}, \mu_E : E_L \to \mathbb{M}_E \backslash \{\mathtt{none}\} \rangle$ is a partial function assigning marks to each node (or edge) whose mark is `any`. □

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

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 defined as usual (see [18]). 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 [7] for unrooted graph programming.

Note that we require natural double-pushout in rule schema application. We use a natural pushout to have a unique pushout complement up to isomorphism in relabelling graph transformation[7,8]. In [1], a graph morphism preserves rooted nodes while here we require a morphism to preserve unrooted nodes as well. We require the preservation of unrooted nodes to prevent a non-natural pushout[2]. In addition, we need a natural double-pushout because we want to have invertible direct derivations.

Definition 4 (Conditional rule schema application). Given a conditional rule schema $r = \langle L \leftarrow K \rightarrow R,\ \Gamma \rangle$, and host graphs G, H. G directly derives r, denoted by $G \Rightarrow_{r,g} H$ (or $G \Rightarrow_r H$), if there exists a premorphism $g : L \to G$ 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) $G \Rightarrow_{r^\alpha,g} H$.

where $G \Rightarrow_{r^{g,\alpha},g} H$ denotes the existence of natural double-pushout below:

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

□

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.

2.3 Commands and semantics of graph programs

There are some commands that is considered as the *core commands* of GP 2, that are: rule set call, sequential command, if-then-else command, `try` − `then` − `else` command, and loop command (!). Rule set call is a command to apply the given set of rules, while other commands work similar to most programming languages. However, the condition for the branching commands (if, try, and loop) is a graph program, where its 'truth value' depends on the existence of successful or failure execution of the program. Also, the condition of a loop command is the loop body of the command. For more details, see Fig. 8 for the abstract syntax of GP 2, and Fig. 9 for the inference rule of core commands.

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 $\bot$. The value `fail` indicates a failed program run while $\bot$ indicates a run that does not terminate or gets stuck. Program P can diverge from G if there is an infinite sequence $\langle P, G\rangle \to \langle P_1, G_1\rangle \to \langle P_2, G_2\rangle \to \ldots$. Also, P can get stuck from G if there is a terminal configuration $\langle Q, H\rangle$ such that $\langle P, G\rangle \to^* \langle Q, H\rangle$.

3 First-Order Formulas for Graph Programs

In this section, we define first-order formulas which specify classes of GP 2 graphs. We also show how to represent concrete GP 2 graphs in rule schema applications.

3.1 Syntax of first-order formulas

To be able to express GP 2 graphs, we need to be able to express properties of a graph and GP 2 rule schema conditions. A graph in GP 2 is a system that consists of a finite set of nodes and edges, where each edge has source and target nodes, a label of a node or an edge consists of a list and a mark, and nodes may be rooted. For more detail on GP 2, see Appendix. For our first-order formula, we use the same type of variables used in GP 2 rule graphs with additional node and edge variables as seen in Table 1.

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

kind of variables	Node	Edge	List	Atom	Int	String	Character
domain	V_G	E_G	$(\mathbb{Z} \cup (\text{Char})^*)^*$	$\mathbb{Z} \cup \text{Char}^*$	$\mathbb{Z}$	Char^*	Char

The syntax of FO formulas is given by the grammar of Figure 1. 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 list, atom, int, string, and char respectively. The nonterminals Character and Digit in the syntax represent the fixed character set of GP 2, and the digit set $\{0, \ldots, 9\}$ respectively.

```
Formula ::= true | false | Cond | Equal
          | Formula ('∧' | '∨') Formula | '¬'Formula | '('Formula')'
          | '∃V' (NodeVar) '('Formula')'
          | '∃E' (EdgeVar) '('Formula')'
          | '∃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 | lV '('Node')' | lE '('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 | mV'('Node')' ('=' | '≠') VMark
          | mE'('EdgeVar')' ('=' | '≠') EMark
```

Fig. 1: Syntax of first-order formulas

The quantifiers $\exists_V, \exists_E$, and $\exists_L$ in the grammar are reserved for variables of nodes, edges, and labels respectively. The function symbols indeg, outdeg and length returns indegree, outdegree, and length of the given argument. In addition, we also have unary functions s, t, l_V, l_E, m_V, and m_E, which takes the argument and respectively return the value of its source, target, node label, edge label, node mark, and edge mark. 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 $\neg\exists_V x(\neg c)$ and $\exists_V x_1, \ldots, x_n(c)$ for $\exists_V x_1(\exists_V x_2(\ldots\exists_V x_n(c)\ldots))$ (also for edge and label quantifiers). Also, we define 'term' as the set of variables, constants, and functions in first-order formulas.

The satisfaction of a FO formula c in a host graph G relies on *assignments*. An assignment α of a formula c on G is a pair $\langle \alpha_G, \alpha_{\mathbb{L}} \rangle$ where α_G is function that maps every free node (or edge) variable to a node (or edge) in G, and $\alpha_{\mathbb{L}}$ is a function that maps every free char, string, integer, atom, and list variable in c to a member of its domain based on Table 1. From an assignment α, we can obtain c^α by replacing every free variable x with $\alpha(x)$, and evaluate the functions based on the semantics of their associated GP 2 syntax. G satisfies c by assignment α, denotes by $G \vDash^\alpha c$ if and only if c^α is true in G.

The truth value of c^α is evaluated just like in standard logic, with respect to the semantics of the predicates as described above, where $(\mathsf{root}(x))^\alpha$ is true in G if x^α is rooted, or false otherwise. We then write $G \vDash c$ if there exists an assignment α such that $G \vDash^\alpha c$.

3.2 Conditions for rule schema applications

First-order formulas as defined above do not contain node or edge constants because we want to be able to check the satisfaction of formulas on arbitrary host graphs. However, for rule schema applications we will need to express properties of specific nodes and edges of the graphs in the rule schema. For this, we define a *condition over a graph* that can be obtained from a first-order formula and an assignment.

Definition 5 (Conditions). A *condition* is a first-order formula without free node and edge variables. A *condition over a graph* G is a first-order formula where every free node and edge variable is replaced with node and edge identifiers in G. That is, if c is a FO formula and α_G is an assignment of free node and edge variables of c on G, then c^{α_G} is a condition over G. □

Checking if a graph satisfies a condition c over a graph is essentially the same as checking satisfaction of a FO formula in a graph. However, the satisfaction of c in a graph G can be defined only if c is a condition over G. For a rule schema $\langle L \leftarrow K \rightarrow R \rangle$, the satisfaction of a condition over L may not be defined for some host graphs G. However, we can rename nodes and edges in the host graphs so that we can check the satisfaction.

Definition 6 (Replacement graph). Given an injective morphism $g : L \rightarrow G$ for host graphs L and G. Graph $\rho_g(G)$ is a replacement graph of G w.r.t. g if $\rho_g(G)$ is isomorphic to G with L as a subgraph. □

A conditional rule schema is not invertible because of the restrictions on the variables and the existence of the rule schema condition that is reserved only for the left-hand graph. However, an invertible rule is sometimes needed to be able to derives properties from output graph to the input graph. Hence, we define a generalisation of a rule schema.

Definition 7 (Generalised rule schema). Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. *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 call ac_L the left application condition and ac_R the right application condition. The inverse of w, written w^{-1}, is then defined as the tuple $\langle r^{-1}, ac_R, ac_L \rangle$ where $r^{-1} = \langle R \leftarrow K \rightarrow L \rangle$. □

The application of a generalised rule schema is essentially the same as the application of a rule schema. However in a generalised version, we need to consider the satisfaction of both left and right-application condition in the replacement graph of input and output graphs.

For a conditional rule schema $r = \langle \langle L \leftarrow K \rightarrow R \rangle, \Gamma \rangle$, we denote by $r^\vee$ the general version of r, that is the generalised rule schema $r^\vee = \langle \langle L \leftarrow K \rightarrow R \rangle, \Gamma^\vee, \texttt{true} \rangle$ where $\Gamma^\vee$ is obtained from Γ by replacing the notations !=, `not`, `and`, `or`, # with $\neq, \neg, \wedge, \vee, ',$ (comma symbol) respectively.

4 Constructing a Strongest Liberal Postcondition

In this section, we show how to construct a strongest liberal postcondition for a conditional rule schema and a precondition. The condition expresses properties that must be satisfied by every graph resulting from the application of the rule schema to a graph satisfying the given precondition [5].

Definition 8 (Strongest liberal postcondition). An assertion d is a *liberal postcondition* with respect to a precondition c and a graph program P, if for all host graphs G and H, ($G \vDash c$ and $H \in [\![P]\!]G$) 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. □

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. $\mathrm{SLP}(c, r)$ 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. $r^{\vee}$, which is then transformed into a right-application condition by Shift. Finally, the transformation Post transforms the right-application condition to $\mathrm{SLP}(c, r)$. A simple example of the construction can be seen in Section 6.

4.1 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, denoted by $\mathrm{Split}(c, r)$; 2) Express the dangling condition as a condition over L, denoted by $\mathrm{Dang}(r)$; 3) Evaluate terms and Boolean expression in $\mathrm{Split}(c, r)$, $\mathrm{Dang}(r)$, and $\Gamma^{\vee}$, 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 $c = \exists_{\mathsf{V}} x(c_1)$ 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. $\mathrm{Split}(c, r)$ is obtained from the disjunction of all these possibilities.

Since we only concern about node and edge variables and the fact that a precondition is a closed formula, $\mathrm{Split}(c, r)$ can be obtained from c by changing every subformula in the form $\exists_{\mathsf{V}} x(c_1)$ to $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \mathrm{Split}(c_1^{[x \mapsto v_i]}, r)) \vee \exists_{\mathsf{V}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{x} \neq \mathsf{v_i} \wedge c_1)$ and $\exists_{\mathsf{E}} x(c_1)$ to $(\bigvee_{\mathsf{i}=1}^{\mathsf{m}} \mathrm{Split}(c_1^{[x \mapsto e_i]}, r)) \vee \exists_{\mathsf{E}} \mathsf{x}(\bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{x} \neq \mathsf{e_i} \wedge \mathrm{inc}(c_1, r, x))$, where

$$\mathrm{inc}(c_1, r, x) = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} (\bigvee_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s}(\mathsf{x}) = \mathsf{v_i} \wedge \mathsf{t}(\mathsf{x}) = \mathsf{v_j} \wedge \mathrm{Split}(c_1^{[\mathsf{s}(\mathsf{x}) \mapsto \mathsf{v_i}, \mathsf{t}(\mathsf{x}) \mapsto \mathsf{v_j}]}, r))$$

$$\vee\,(\mathsf{s(x)} = \mathsf{v_i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge \mathrm{Split}(c_1^{[\mathsf{s(x)}\mapsto\mathsf{v_i}]}, r))$$
$$\vee\,(\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_j} \wedge \mathsf{t(x)} = \mathsf{v_i} \wedge \mathrm{Split}(c_1^{[\mathsf{t(x)}\mapsto\mathsf{v_i}]}, r))$$
$$\vee\,(\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{s(x)} \neq \mathsf{v_i} \wedge \bigwedge_{\mathsf{j}=1}^{\mathsf{n}} \mathsf{t(x)} \neq \mathsf{v_j} \wedge c_1).$$

When c_1 contains a node/edge quantifier, we change them from the innermost to the outermost node/edge quantifier.

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

Beside obtaining $\mathrm{Split}(c, r)$, we also need to express the dangling condition as a condition over L. 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. $\rho_g(G)$ satisfies the dangling condition if every node $v \in L - K$ and every edge not in L are not incident. This means that the indegree and outdegree of every node $v \in L - K$ in L represent the indegree and outdegree of v in $\rho_g(G)$ as well. Hence, if we have $V_L - V_K = \{v_1, \ldots, v_n\}$, we can have:
(i) $\mathrm{Dang}(r) = \mathsf{true}$ if $V_L - V_K = \emptyset$, and
(ii) $\mathrm{Dang}(r) = \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{indeg(v_i)} = \mathit{indeg}_L(v_i) \wedge \mathsf{outdeg(v_i)} = \mathit{outdeg}_L(v_i)$ otherwise.

The last step to obtain a left-application condition is to evaluate $\mathrm{Split}(c, r)$, $\mathrm{Dang}(r)$, and $\Gamma^{\vee}$ with respect to the left-hand graph (note that those three are conditions over L). Intuitively, $\mathrm{Val}(d, r)$ of a condition d over L where L is the left-hand graph of r, is a condition that is obtained from d by replacing every term with its value in L where possible. Possible here means if the argument of the term contains a constant. We then simplify the resulting condition so that there is no subformulas in the form $\neg\,\mathsf{true}$,$\neg(\neg\, a)$ $\neg(a \vee b)$, $\neg(a \wedge b)$ for some conditions a, b. We can simplify them to $\mathsf{false}, a, \neg a \wedge \neg b, \neg a \vee \neg b$ respectively.

There is a special case when the term is in the form $\mathsf{indeg(x)}$ or $\mathsf{outdeg(x)}$ because unlike the other terms, their value in L is different with their value in the replacement graph of the input graph. For more information about handling this case, we refer reader to [18].

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 9 (Transformation Lift). For a precondition c and a generalised rule $w = \langle r, ac_L, ac_R\rangle$ with an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R\rangle$,

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

4.2 From left to right-application condition

To obtain a right-application condition from the obtained 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.

The Boolean value for $\mathsf{x} = \mathsf{i}$ for any node/edge variable x and node/edge constant i not in R must be false in the resulting graph. Analogously, $\mathsf{x} \neq \mathsf{i}$ is always true. Also, all variables in the left-application condition should not represent any new nodes and edges in the right-hand side. Hence, to obtain the right-application condition $\mathrm{Shift}(c, w)$, we have some adjustment to the obtained left-application condition, denoted by $\mathrm{Adj}(d, r)$ where $d = \mathrm{Lift}(c, w)$.

To obtain $\mathrm{Adj}(d, r)$, we follow the following steps: 1) replace term representing indegree or outdegree if any (see [18] for detail), 2) replace every subformula in the form $x_1 \neq x_2$ with true and $x_1 = x_2$ with false if x_1 or x_2 is in $V_L - V_K$ or $E_L - E_K$, 3) replace every $\exists_{\mathsf{V}} x(c_1)$ in c' with $\exists_{\mathsf{V}} x(x \neq v_1 \wedge \ldots \wedge x \neq v_n \wedge c_1)$ and every $\exists_{\mathsf{E}} x(c_1)$ with $\exists_{\mathsf{E}} x(x \neq e_1 \wedge \ldots \wedge x \neq e_m \wedge c_1)$ for $V_R - V_K = \{v_1, \ldots, v_n\}$ and $E_R - E_K = \{e_1, \ldots, e_n\}$.

Actually, $\mathrm{Adj}(\mathrm{Lift}(c, w), r)$ can be considered as a right-application condition. However, to have a stronger condition, we add a condition over R expressing the specification of the right-hand graph. A *specification of a graph* R, denoted by $\mathrm{Spec}(R)$, can be easily obtained by forming conjunction of predicates and equality of functions and its value in R that can give us all information about R and type of label variables in R.

Lemma 1. For every totally labelled rule graph R, there exists a condition $\mathrm{Spec}(R)$ such that for every host graph G, $G \vDash \mathrm{Spec}(R)$ if and only if there exists assignment $\alpha_{\mathbb{L}}$ such that $g : R^{\alpha_{\mathbb{L}}} \to G$ is an inclusion.

Definition 10 (Shifting). Given a generalised rule $w = \langle r, ac_L, ac_R \rangle$ for an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$, and a precondition c. Right application condition w.r.t. c and w, denoted by $\mathrm{Shift}(c, w)$, is defined as:

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

4.3 From right-application condition to postcondition

The right-application condition we obtain from transformation Shift is strong enough to express properties of the replacement graph of any resulting graph. However, it is a condition over R, so we need to change it to a FO formula. This can be done by replacing every node and edge constant to a fresh variable and state that each new variable is not equal to other new variables.

Lemma 2. For a rule graph G and a condition c over G, there exists a first-order formula $\mathrm{Var}(c)$ so that for every graph H that is isomorphic to G, $G \vDash c$ implies $H \vDash \mathrm{Var}(c)$.

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. In [18], we show that the obtained formula defines a strongest liberal postcondition.

Definition 11 (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. Let $\{x_1, \ldots, x_n\}$, $\{y_1, \ldots, y_m\}$, and $\{z_1, \ldots, z_k\}$ denote the set of free node, edge, and label (resp.) variables in $\mathrm{Var}(\mathrm{Shift}(c, w))$. We define $\mathrm{Post}(c, w)$ as the FO formula:
$\mathrm{Post}(c, w) \equiv \exists_{\mathsf{V}} x_1, \ldots, x_n (\exists_{\mathsf{E}} y_1, \ldots, y_m (\exists_{\mathsf{L}} z_1, \ldots, z_k (\mathrm{Var}(\mathrm{Shift}(c, w)))))$.
We denote by $\mathrm{Slp}(c, r)$ the formula $\mathrm{Post}(c, r^{\vee})$ for a rule schema r, and $\mathrm{Slp}(c, r^{-1})$ the formula $\mathrm{Post}(c, (r^{\vee})^{-1})$. □

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, $\mathrm{Slp}(c, r)$ is a strongest liberal postcondition w.r.t. c and r.

5 Proof Calculi

In this section, we introduce a semantic and a syntactic partial correctness calculus. For pre- and postconditions, we consider arbitrary assertions (without giving their syntax) for the former, and first-order formulas for the latter.

Given a graph program P and assertions c and d, a triple $\{c\}\, P\, \{d\}$ is *partially correct*, denoted by $\vDash \{c\}\ P\ \{d\}$, if for every graph G satisfying c, all graphs in $[\![P]\!]G$ satisfy d [15].

5.1 Semantic partial correctness calculus

In addition to a strongest liberal postcondition we defined in the previous section, here we define a weakest liberal precondition.

Definition 12 (Weakest liberal precondition). 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, $G \vDash c$ and $H \in [\![P]\!]G$ implies $H \vDash d$.
A *weakest liberal precondition* w.r.t. P, d, written $\mathrm{WLP}(P, d)$, is a liberal precondition w.r.t. P, d that is implied by all liberal postconditions w.r.t. P, d. □

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 or $\mathrm{WLP}(P, d)$ implies c. However if P contains a loop, obtaining the $\mathrm{SLP}(c, P)$ or $\mathrm{WLP}(P, d)$ may not be easy because $P!$ may get stuck or diverge. In [11], the divergence is represented by infinite formulas while in [9], they use an approximation. However, we take different approach by only considering loop-free programs in obtaining SLP or WLP.

For programs with loops, we can create a proof tree 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 over the command `break`. Also, we define assertions that can express properties of host graphs that asserts the existence of result graph, which is important for the execution of branching commands `if`/`try` − `then` − `else`.

Definition 13 (Assertions SUCCESS, FAIL). For a graph program P, SUCCESS(P) and FAIL(P) are the predicates on host graphs where for all host graph G, $G \vDash \text{SUCCESS}(P)$ if and only if there exists a host graph H with $H \in [\![P]\!]G$, and $G \vDash \text{FAIL}(P)$ if and only if fail $\in [\![P]\!]G$. □

Definition 14 (Predicate Break). Given a graph program P and assertions c and d. Break(c, P, d) is the predicate defined by:
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 Break(c, P, d) holds, the execution of `break` that yields to termination of $P!$ will result in a graph satisfying d.

Definition 15 (Semantic partial correctness proof rules). The semantic partial correctness proof rules for GP 2 commands, denoted by SEM, are defined in Fig. 2a, where c, d, and d' are assertions, r is a conditional rule schema, $\mathcal{R}$ is a set of rule schemata, and C, P, and Q are graph programs. □

The assertions SUCCESS and FAIL are needed to prove a triple about an `if` command, because P may be executed on G if $G \vDash \text{SUCCESS}(C)$, and Q may be executed on G if $G \vDash \text{FAIL}(C)$. Similarly, for a `try` command, P may be executed on a graph C' if $G \vDash \text{SUCCESS}(C)$ and $C' \in [\![C]\!]G$, and Q may be executed on G if $G \vDash \text{FAIL}(C)$. Finally the execution of a loop $P!$, it terminates if at some point the execution of P yields failure, or reaches the command `break`.

5.2 Syntactic partial correctness calculus

Defining a first-order formula for SUCCESS(r) with a rule schema r is easier than defining FO formula for SUCCESS(P) with a program P with loops. This is because the existence of a result graph can be known after some execution of P, which really depends on the program. Moreover, it may diverge. However if we consider loop-free programs, we can construct a first-order formula for SUCCESS, FAIL and SLP. In addition, we can construct a FO formula of FAIL(P) for bigger class of programs because some commands cannot fail (see [1]).

Definition 16 (Non-failing commands). The class of *non-failing commands* is inductively defined as follows:

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

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

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

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

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

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

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

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

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

(a) Calculus SEM (b) Calculus SYN

Fig. 2: Semantic (a) and syntactic (b) partial correctness proof calculus, where $\text{S}(C)$ is $\text{SUCCESS}(C)$ and $\text{F}(C)$ is $\text{FAIL}(C)$

3. Every rule set call $\{r_1, \ldots, r_n\}$ for $n \geq 1$ where each r_i has the empty graph as its left-hand graph, is a non-failing command
4. Every command P! is a non-failing command
5. if P and Q are non-failing commands, then $P;Q$, $\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q$, and $\texttt{try}\ C\ \texttt{then}\ P\ \texttt{else}\ Q$ are non-failing commands. □

Now, let us consider P in the form $C;Q$. For any host graph G, $\text{fail} \in [\![C;Q]\!]G$ iff $\text{fail} \in [\![C]\!]G$ or $H \in [\![C]\!]G \wedge \text{fail} \in [\![Q]\!]H$ for some host graph H, which means $G \vDash \text{FAIL}(C) \vee (\text{SUCCESS}(C) \wedge \text{FAIL}(Q))$. We can construct both $\text{Fail}(C)$ and $\text{Success}(C)$ if C is a loop-free program (see [18] for the detail of construction), and we can construct $\text{Fail}(Q)$ if Q is a loop-free program or a non-failing command. Here, we introduce the class of *iteration commands* for which we can obtain Fail of the commands.

Definition 17 (Iteration commands). The class of iteration commands is inductively defined as follows: 1) every loop-free program and non-failing command is an iteration command, and 2) a command in the form $C;P$ is an iteration command if C is a loop-free program and P is an iteration command. □

Definition 18. Let $\text{Fail}_{\text{lf}}(C)$ denotes the formula $\text{Fail}(C)$ for a loop-free program C. For any iteration command S,

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

□

Theorem 2. For any loop-free program P and precondition c, there exists first-order formula $\text{Success}(P)$ and $\text{Slp}(c,P)$ such that $G \vDash \text{Success}(P)$ if and only if $G \vDash \text{SUCCESS}(P)$ and $G \vDash \text{Slp}(c,P)$ if and only if $G \vDash \text{SLP}(c,P)$. Also, for any iteration command S, $G \vDash \text{Fail}(S)$ if and only if $G \vDash \text{FAIL}(S)$.

The construction of $\mathrm{Slp}(c, P)$ and $\mathrm{Success}(P)$ to show that Theorem 2 holds can be found in [18]. Since we only have a construction for $\mathrm{Success}(C)$ for a loop-free program C and $\mathrm{Fail}(S)$ for an iteration command S, we cannot define the syntactic proof calculus for arbitrary graph programs. We call the class of programs we can handle by our syntactical calculus as control programs.

Definition 19 (Control programs). A *control command* is a command where the condition of every branching command is loop-free and every loop body is an iteration command. Similarly, a graph program is a *control program* if all its command are control commands. □

As in [6], a First-order formula of $\mathrm{WLP}(r, d)$ of a postcondition d and a rule schema r can be easily constructed from the construction of a strongest liberal postcondition.

Lemma 3. Given a closed FO formula d and a rule schema r. Then for all host graphs G, $G \vDash \neg\mathrm{Slp}(\neg d, r^{-1})$ if and only if $G \vDash \mathrm{WLP}(r, d)$.

Definition 20 (Syntactic partial correctness proof rules). The syntactic partial correctness proof rules, denoted by SYN, are defined in Fig. 2b, where c, d, and d' are conditions, r is a conditional rule schema, $\mathcal{R}$ is a set of rule schemata, C is a loop-free program, P and Q are control commands, and S is an iteration command. □

In the following section, we give a graph verification example using the calculus SYN we defined in this section.

6 Example: Verifying a 2-Colouring Program

In this section, we show how to verify the 2-colouring graph program given in Fig. 3. The 2-colouring problem is the problem to assign to each node of a graph one of two colours such that each two adjacent nodes have different colours.

The program expects input graphs without any roots or marks. It starts by marking any unmarked node with red, then repeatedly colours uncoloured nodes adjacent to a coloured node with the other colour. Finally, the program checks if the produced graph contains two adjacent nodes with the same colour. If that is the case, the program unmarks all nodes to restore the input graph. Note the nested loop which allows to process disconnected graphs, by colouring each connected component in turn. This program cannot be verified with the proof calculi in [16,14] as there exists a nested loop in the program.

Let us consider the precondition "every node and edge is unmarked and every node is unrooted" and the postcondition "the precondition holds or every node is marked with blue or red, and no two adjacent nodes marked with the same colour", that can be represented by c and $c \vee d$ where
$c = \forall_{\mathsf{V}}\mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none} \wedge \neg\mathsf{root}(\mathsf{x})) \wedge \forall_{\mathsf{E}}\mathsf{x}(\mathsf{m}_{\mathsf{E}}(\mathsf{x}) = \mathsf{none})$, and
$d = \forall_{\mathsf{V}}\mathsf{x}((\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{red} \vee \mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{blue})) \wedge \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{s}(\mathsf{x}) \neq \mathsf{t}(\mathsf{x}) \wedge \mathsf{m}_{\mathsf{V}}(\mathsf{s}(\mathsf{x})) = \mathsf{m}_{\mathsf{V}}(\mathsf{t}(\mathsf{x})))$

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

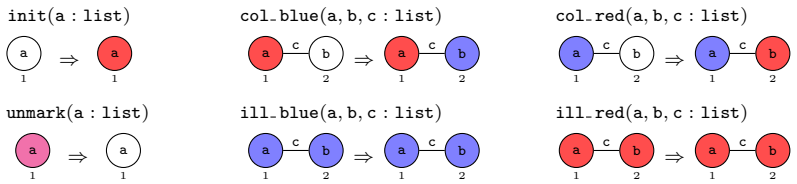


Fig. 3: Graph program `2-colouring`

By using the conditions in Table 2, we then have a proof tree as in Fig. 4 for the partial correctness of $2-$`colouring` with respect to c and $c \vee d$.

Table 2: Conditions inside proof tree of $2-$`colouring`

symbol and its first-order formulas
$c \equiv \forall_V x(m_V(x) = \text{none} \wedge \neg\text{root}(x)) \wedge \forall_E x(m_E(x) = \text{none})$
$d \equiv \forall_V x((m_V(x) = \text{red} \vee m_V(x) = \text{blue})) \wedge \neg\exists_E x(s(x) \neq t(x) \wedge m_V(s(x)) = m_V(t(x)))$
$e \equiv \forall_V x((m_V(x) = \text{red} \vee m_V(x) = \text{blue}) \wedge \neg\text{root}(x)) \wedge \forall_E x(m_E(x) = \text{none})$
$f \equiv \forall_V x((m_V(x) = \text{red} \vee m_V(x) = \text{blue} \vee m_V(x) = \text{none})) \wedge \neg\text{root}(x)) \wedge \forall_E x(m_E(x) = \text{none})$
$\text{Slp}(f,$ `init`$)$ $\equiv \exists_V y(\forall_V x(x = y \vee ((m_V(x) = \text{red} \vee m_V(x) = \text{blue} \vee m_V(x) = \text{none}) \wedge \neg\text{root}(x)))$ $\wedge\, m_V(y) = \text{red} \wedge \neg\text{root}(y)) \wedge \forall_E x(m_E(x) = \text{none})$
$\text{Slp}(f,$ `c_blue`$) = \text{Slp}(f,$ `c_red`$)$ $\equiv \exists_V u, v(\forall_V x(x = u \vee x = v \vee ((m_V(x) = \text{red} \vee m_V(x) = \text{blue} \vee m_V(x) = \text{none}) \wedge \neg\text{root}(x)))$ $\wedge\, m_V(u) = \text{red} \wedge m_V(v) = \text{blue} \wedge \neg\text{root}(u) \wedge \neg\text{root}(v)$ $\wedge\, \exists_E y((s(y) = u \wedge t(y) = v) \vee (t(y) = u \wedge s(y) = v))) \wedge \forall_E x(m_E(x) = \text{none})$
$\text{Slp}($f,unmark$)$ $\equiv \exists_V y(\forall_V x(x = y \vee ((m_V(x) = \text{red} \vee m_V(x) = \text{blue} \vee m_V(x) = \text{none}) \wedge \neg\text{root}(x)))$ $\wedge\, m_V(y) = \text{none} \wedge \neg\text{root}(y)) \wedge \forall_E x(m_E(x) = \text{none})$
$\text{Fail}($`Colour`$)$ $\equiv \neg\exists_E x((((m_V(s(x)) = \text{red} \vee m_V(s(x)) = \text{blue}) \wedge m_V(t(x)) = \text{none})$ $\vee\, ((m_V(t(x)) = \text{red} \vee m_V(t(x)) = \text{blue}) \wedge m_V(s(x)) = \text{none}))$ $\wedge\, \neg\text{root}(s(x)) \wedge \neg\text{root}(t(x)))$
$\text{Fail}($`init; Colour!`$) \equiv \neg\exists_V x(m_V(x) = \text{none} \wedge \neg\text{root}(x))$
$\text{Fail}($`unmark`$) \equiv \neg\exists_V x(m_V(x) \neq \text{none} \wedge \neg\text{root}(x))$
$\text{Fail}($`Illegal`$)$ $\equiv \neg\exists_E x(s(x) \neq t(x)$ $\wedge\, ((m_V(s(x)) = \text{red} \wedge m_V(t(x)) = \text{red}) \vee (m_V(s(x)) = \text{blue} \wedge m_V(t(x)) = \text{blue})))$
$\text{Success}($`Illegal`$)$ $\equiv \exists_E x(s(x) \neq t(x)$ $\wedge\, ((m_V(s(x)) = \text{red} \wedge m_V(t(x)) = \text{red}) \vee (m_V(s(x)) = \text{blue} \wedge m_V(t(x)) = \text{blue})))$

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

For an example of constructing Slp, let us consider the rule $r =$ `init` of program $2-$`colouring` and the formula f of Table 2. Note that $\forall x(c)$ is an abbreviation of $\neg\exists x(\neg c)$ so that we need to change universal quantifiers to existential quantifiers.

$$\begin{aligned}\text{Split}(f, r) = {} & \neg((m_V(1) \neq \text{red} \wedge m_V(1) \neq \text{blue} \wedge m_V(1) \neq \text{none}) \vee \text{root}(1)) \\ & \wedge\neg\exists_V x(x \neq 1 \wedge (m_V(x) \neq \text{red} \wedge m_V(x) \neq \text{blue} \wedge m_V(x) \neq \text{none}) \vee \text{root}(x)) \\ & \wedge\neg\exists_E x(m_E(x) \neq \text{none})\end{aligned}$$

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

where subtree I is:

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

with subtree I.a:

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

and subtree II is:

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

Fig. 4: Proof tree for partial correctness of `2colouring`

$$\text{Dang}(r) = \mathsf{true}$$
$$\text{Lift}(f, r^{\vee}) = \neg\exists_{\mathsf{V}}\mathsf{x}(\mathsf{x} \neq 1 \wedge (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{red} \wedge \mathsf{m_V}(\mathsf{x}) \neq \mathsf{blue} \wedge \mathsf{m_V}(\mathsf{x}) \neq \mathsf{none}) \vee \mathsf{root}(\mathsf{x})) \wedge \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) \neq \mathsf{none})$$
$$\text{Adj}(\text{Lift}(f, r^{\vee}), r) = \text{Lift}(f, r^{\vee})$$
$$\text{Shift}(f, r^{\vee}) = \text{Lift}(f, r^{\vee}) \wedge \mathsf{l_V}(1) = \mathsf{a} \wedge \mathsf{m_V}(1) = \mathsf{red} \wedge \neg\mathsf{root}(1)$$
$$\text{Slp}(f, r) \equiv \exists_{\mathsf{V}}\mathsf{y}(\neg\exists_{\mathsf{V}}\mathsf{x}(\mathsf{x} \neq \mathsf{y} \wedge (\mathsf{m_V}(\mathsf{x}) \neq \mathsf{red} \wedge \mathsf{m_V}(\mathsf{x}) \neq \mathsf{blue} \wedge \mathsf{m_V}(\mathsf{x}) \neq \mathsf{none}) \vee \mathsf{root}(\mathsf{x})) \wedge \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{m_E}(\mathsf{x}) = \mathsf{none}) \wedge \exists_{\mathsf{L}}\mathsf{a}(\mathsf{l_V}(\mathsf{y}) = \mathsf{a}) \wedge \mathsf{m_V}(\mathsf{y}) = \mathsf{red} \wedge \neg\mathsf{root}(\mathsf{y}))$$

In the proof tree of Fig. 4, 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 $c \vee d$. Other implications, are also obvious if we check their formulas. The implications have the form $\exists y(\forall x((x = y \vee c) \wedge x = y \Rightarrow c))$ for some variables x, y and FO formula c with no variable y, which implies $\forall x(c)$. For an example, $\text{Post}(f, \texttt{init})$ expresses that there exists an unrooted red node y, labelled with a list, where all nodes beside y are unmarked or marked red or blue, which implies all nodes are unmarked or marked red or blue, such that f holds. Other proof of implications use similar method (see [18]).

7 Soundness and Completeness of the Proof Calculi

In [18], we show that both SEM and SYN are sound. That is, if a triple $\{c\}\ P\ \{d\}$ can be proven by SEM or SYN (denoted by $\vdash_{\mathsf{SEM}}$ or $\vdash_{\mathsf{SYN}}$), then the triple is partially correct.

Theorem 3 (Soundness). Given a graph program P and assertions c, d. Then, $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$ implies $\vDash \{c\}\ P\ \{d\}$. Moreover, if c and d are first-order formulas, $\vdash_{\mathsf{SYN}} \{c\}\ P\ \{d\}$ implies $\vDash \{c\}\ P\ \{d\}$.

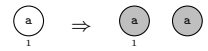
A proof calculus is complete if every partially correct triple can be proved by the calculus. Neither SEM nor SYN are complete because GP 2's expressions include Peano arithmetic which is known to be incomplete. However, the notion of relative completeness allows to separate the incompleteness in proving valid assertions from the power of the inference rules for programming constructs [3]. That means, we assume that the implications in the [cons] rules of SEM and SYN can be proved outside the calculi.

Theorem 4 (Relative completeness of SEM). Given a graph program P and assertions c, d. Then, $\vDash \{c\}\ P\ \{d\}$ implies $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$.

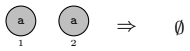
The proof of Theorem 4 can be seen in [18]. The proof relies on $\mathrm{WLP}(P, c)$ for arbitrary program P and assertion c. Even if we omit $[\mathrm{ruleapp}_{\mathrm{slp}}]$ from the calculus, SEM is still relative complete. However, for SYN to be relative complete, it would be necessary to express these assertions in FO formula. There is strong evidence that this is impossible. For example, consider the triple $\{c\}\ P\ \{d\}$ with $c = \forall_{\mathsf{V}} \mathsf{x}(\mathsf{m}_{\mathsf{V}}(\mathsf{x}) = \mathsf{none} \wedge \neg\exists_{\mathsf{E}} \mathsf{y}(\mathsf{s}(\mathsf{y}) = \mathsf{x} \vee \mathsf{t}(\mathsf{y}) = \mathsf{x}))$ (all nodes are unmarked and isolated), $d = \forall_{\mathsf{V}} \mathsf{x}(\mathsf{false})$ (the graph is empty), and the following program:

`Main` $=$ `duplicate!; delete!`

`duplicate(a : list)`



`delete(a : list)`



It is obvious that $\vDash \{c\}$ `duplicate!; delete!` $\{d\}$ holds: `duplicate!` duplicates the number of nodes while marking the nodes grey, hence its result graph consists of an even number of isolated grey nodes . Then `delete!` deletes pairs of grey nodes as long as possible, so the overall result must be the empty graph.

Using SYN one can prove $\vdash \{c\}$ `duplicate!` $\{e\}$ where e expresses that all nodes are grey and isolated. However, we believe that our logic cannot express that a graph has an even number of nodes. This is because pure first-order logic (without built-in operations) cannot express this property [10] and it is likely that this inexpressiveness carries over to our logic. As a consequence, one can only prove $\vdash \{e\}$ `delete!` $\{f\}$ where f expresses that the graph contains at most one node (because otherwise `delete` would be applicable). But we cannot use SYN to prove $\vdash \{c\}$ `duplicate!; delete!` $\{d\}$.

8 Related Work

Hoare style verification of graph programs was introduced in [16,14], using E-conditions which generalise nested graph conditions. Neither rooted rules nor

the `break` command are considered. In addition, that approach can only handle programs in which the conditions of branching commands and loop bodies are rule set calls. Here, we introduce a calculus that is able to handle a larger class of graph programs. That is, graph programs where the condition 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. In addition, we believe that first-order formulas are easier to comprehend by programmers than some form of nested graph conditions.

Expressing Success(P), Wlp(P, c), and Slp(c, P) for a condition c and a graph program P with loops remains an open problem for now. In [11], there is a construction of Wlp($c, P!$) by using an infinite formula. Here, we do not use a similar trick but stick to standard finitary logic. In [5,9], there are no constructions for syntactical strongest liberal postcondition and weakest liberal postcondition either. But similar to our inference rule [alap], the conjunction of a loop invariant and a negated loop condition is considered as an approximate strongest liberal postcondition.

9 Conclusion and Future Work

We have shown how to construct a strongest liberal postcondition for a given conditional rule schema and a precondition in the form of a first-order formula. Using 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. Altogether, this gives us a proof calculus that can handle more programs than previous calculi in the literature, in particular we can now handle certain nested loops.

However, the expressiveness of first-order formulas over the domain of graphs is quite limited. For example, one cannot specify that a graph is connected by a first-order formula. Hence, in the near future, we will extend our formulas to monadic second-order formulas to overcome such limitations [4].

Another limitation in current approaches to graph program verification is the inability to specify isomorphisms between the initial and final graphs [17]. Monadic second-order transductions can link initial and final states by expressing the final state through elements of the initial state [4]. We plan to adopt this technique for graph program verification in the future.

References

1. C. Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015.
2. G. Campbell. Efficient graph rewriting. *CoRR*, abs/1906.05170, 2019.
3. S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

4. B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.
5. E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990.
6. 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.
7. 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.
8. 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.
9. 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.
10. L. Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
11. K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2009.
12. 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.
13. 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.
14. C. M. Poskitt. *Verification of Graph Programs*. PhD thesis, The University of York, 2013.
15. C. M. Poskitt and D. Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, volume 6372 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2010.
16. C. M. Poskitt and D. Plump. Hoare-style verification of graph programs. *Fundamenta Informaticae*, 118(1-2):135–175, 2012.
17. 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.
18. G. S. Wulandari and D. Plump. Verifying graph programs with first-order logic (long version). Technical report, 2020. https://github.com/UoYCS-plasma/GP2/.

A Appendix: Grammars and inference rules for Section 2

This appendix gives some definitions omitted in Section 2. As defined in Definition 1, a label in a GP 2 graph consists of an expression and a mark. The set of expressions $\mathbb{E}$ is defined by the grammar of Fig. 5b. The set $\mathbb{L}$ of lists is a subset of $\mathbb{E}$ and is defined by the grammar of Fig. 5a.

```
𝕃          ::= empty | GraphExp | 𝕃 ':' 𝕃
GraphExp   ::= ['-'] Digit {Digit} | GraphStr
GraphStr   ::= ' " ' {Character} ' " ' | GraphStr '.' GraphStr
```

(a) Abstract syntax for $\mathbb{L}$

```
 𝔼       ::= 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
```

(b) Abstract syntax for $\mathbb{E}$

Fig. 5: Abstract syntax for GP 2 labels

Here, ListVar, AtomVar, IntVar, StringVar, and CharVar represent variables of type `list`, `atom`, `int`, `string`, and `char` respectively. Character is the set of all printable characters except '"' (i.e. ASCII characters 32, 33, and 35-126), while Digit is the digit set $\{0, \ldots, 9\}$.

The colon operator ':' is used to concatenate expressions while the dot operator '.' is used to concatenate strings. The empty list is signified by the keyword `empty`. The function `indeg` (or `outdeg`) takes a node as its argument and returns the indegree (or outdegree) of the node. The function `length` takes a list variable as an argument and returns the length of the list represented by the variable. The mark `blue`, `red`, `green`, and `grey` are graphically represented by the colour itself, while the mark `any` is represented by the colour magenta.

Lists in GP 2 are typed, that are: `list`, `atom`, `int`, `string`, and `char`. Based on the domain of the types, Fig. 6 shows the type hierarchy of lists in GP 2.

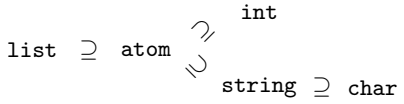


Fig. 6: GP 2 type hierarchy

Rules in GP 2 allows conditions to give additional specification of the left-hand graph. The set of rule schema conditions is described in Fig. 7.

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

```
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
```

Fig. 7: Abstract syntax of rule schema conditions

more details on the abstract syntax of GP 2 programs, see Fig. 8, 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. 8: Abstract syntax of GP 2 programs

Configurations in GP 2 represents a program state of program execution in any stage. Configurations are given by $(\texttt{ComSeq} \times \mathcal{G}(\mathbb{L})) \cup \mathcal{G}(\mathbb{L}) \cup (\texttt{fail}))$, where $\mathcal{G}(\mathbb{L})$ consists of all host graphs. This means that a configuration consists either of unfinished computations, represented by command sequence together with current graph; only a graph, which means all commands have been executed; or the special element `fail` that represents a failure state. A small step transition relation $\rightarrow$ on configuration is inductively defined by inference rules shown in Fig. 9 where $\mathcal{R}$ is a rule set call; C, P, P', and Q are command sequences; and G and H are host graphs.

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

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

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

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

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

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

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

Fig. 9: Inference rules for core commands [13]