
Proceedings Paper:
Wulandari, Gia and Plump, Detlef (2020) Verifying Graph Programs with First-Order Logic. In: Graph Computation Models (GCM 2020), Revised Selected Papers. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 181-200.

https://doi.org/10.4204/EPTCS.330.11

Verifying Graph Programs with First-Order Logic

Gia S. Wulandari*
University of York
York, United Kingdom
Telkom University
Bandung, Indonesia
gsw511@york.ac.uk

Detlef Plump
University of York
York, United Kingdom
detlef.plump@york.ac.uk

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 [17, 15]. These calculi use so-called E-conditions as assertions which extend nested graph conditions [13] 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 first-order logic. To give a simple example, the following E-condition expresses that every node is labelled by an integer: $\forall(\,{}_1\,\textcircled{a}, \exists(\,{}_1\,\textcircled{a} \mid \texttt{int(a)})) \wedge \forall(\,{}_1\,\textcircled{a}, \exists(\,{}_1\,\textcircled{a} \mid \texttt{int(a)})) \wedge \forall(\,{}_1\,\textcircled{a}, \exists(\,{}_1\,\textcircled{a} \mid \texttt{int(a)})) \wedge \forall(\,{}_1\,\textcircled{a}, \exists(\,{}_1\,\textcircled{a} \mid \texttt{int(a)})) \wedge \forall(\,{}_1\,\textcircled{a}, \exists(\,{}_1\,\textcircled{a} \mid \texttt{int(a)}))$. 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 $\forall_V x(\mathsf{int}(\mathsf{l}_V(x)))$. Both E-conditions and first-order formulas tend to get lengthy in examples, but our concern with nested graph conditions is that they require a non-standard interpretation. We believe that programmers cannot be expected to think in terms of morphisms and commuting diagrams, but should be allowed to work with a type of logic that they are familiar with.

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.

To use our assertions in Hoare-style verification, we show how to construct a strongest liberal postcondition $\mathrm{Slp}(c, r)$ for a given conditional rule schema r and a precondition c. Based on this construction, we can define strongest liberal postconditions for arbitrary loop-free graph programs and preconditions. Moreover, for loop-free programs we give syntactic conditions on host graphs which express successful

*Supported by the Indonesia Endowment Fund for Education (LPDP)

B. Hoffmann and M. Minas (Eds.): Eleventh International Workshop on Graph Computation Models (GCM 2020)
EPTCS 330, 2020, pp. 181–200, doi:10.4204/EPTCS.330.11

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 [17, 15]. 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. Then, 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.

2 The Graph Programming Language GP 2

In this section, we briefly review the graph programming language GP 2 which was introduced in [14].

2.1 GP 2 Graphs

A label in a GP 2 graph consists of a list expression and an optional mark. The set $\mathbb{E}$ of expressions is defined by the grammar of Figure 1a. The set $\mathbb{L}$ of host graph lists is a subset of $\mathbb{E}$ and is defined by the grammar of Figure 1b.

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

(a) Expressions (rule graph lists)

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

(b) Host graph lists

Figure 1: Abstract syntax of GP 2 lists

Here Digit is the set $\{0,\ldots,9\}$ and Character is the set of all printable characters except "" (i.e. the ASCII characters 32, 33, and 35-126). The variable sets ListVar, AtomVar, IntVar, StringVar, and CharVar contain variables of type `list`, `atom`, `int`, `string`, and `char`, respectively. The domains of `int` and `string` are the integers $\mathbb{Z}$ and the set Character*, respectively, while `atom` represents the union $\mathbb{Z} \cup$ Character*. The domain of `list` is $(\mathbb{Z} \cup \text{Character}^*)^*$, the set of heterogeneous lists of integers and character strings. We identify lists and strings of length one with their contents and hence have the following subtype relationships: `list` $\supset$ `atom` $\supset$ `string` $\supset$ `char` and `atom` $\supset$ `int`.

The colon operator ':' is used to concatenate lists while the dot operator '.' is used to concatenate strings. The keyword `empty` represents the empty list. The functions `indeg` and `outdeg` take a node as

argument and return the indegree resp. outdegree of the node. The function `length` takes a list or string variable as argument and returns the length of the list resp. string represented by the variable.

Definition 1 (Rule graph) Let $\mathbb{M}_V = \{\texttt{none}, \texttt{red}, \texttt{green}, \texttt{blue}, \texttt{grey}\}$ be the set of *node marks* and $\mathbb{M}_E = \{\texttt{none}, \texttt{red}, \texttt{green}, \texttt{blue}, \texttt{dashed}\}$ be the set of *edge marks*.

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 $s_G, t_G \colon E_G \to V_G$, partial node labelling functions $\ell_G^V \colon V_G \to \mathbb{E}$ and $m_G^V \colon V_G \to \mathbb{M}_V \cup \{\texttt{any}\}$, edge labelling functions $\ell_G^E \colon E_G \to \mathbb{E}$ and $m_G^E \colon E_G \to \mathbb{M}_E \cup \{\texttt{any}\}$, and a partial root function $p_G \colon V_G \to \{0,1\}$. A rule graph is *total* if all of its functions are total functions. □

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

Node labels are undefined only in the interface graphs of rule schemata (see below). This allows rules to relabel nodes. Similarly, the root function is undefined only for the nodes of interface graphs. The purpose of root nodes is to speed up the matching of rule schemata [1, 2].

Given a node v in a graph G, we require that $\ell_G^V(v)$ is defined if and only if $m_G^V(v)$ is defined.

Definition 2 (Host graph) A *host graph* is a total rule graph G satisfying $\ell_G^V(V_G) \subseteq \mathbb{L}$, $\ell_G^E(E_G) \subseteq \mathbb{L}$, $m_G^V(V_G) \subseteq \mathbb{M}_V$ and $m_G^E(E_G) \subseteq \mathbb{M}_E$. □

A *graph morphism* $g : G \to H$ maps nodes to nodes and edges to edges such that sources, targets and labels are preserved. We also require that both roots and non-roots are preserved (see [4] for the root-reflecting mode of the GP 2 compiler). A *premorphism* is defined like a graph morphism except that labels need not be preserved.

2.2 Conditional Rule Schemata

The basic computational unit in GP 2 are graph transformation rules labelled with expressions from $\mathbb{E}$, so-called rule schemata. They allow to modify the structure of host graphs and to perform computations on labels, such as arithmetic or list manipulations. Rule schemata can be equipped with application conditions to increase their expressiveness.

Definition 3 (Conditional rule schema) A *rule schema* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two total rule graphs L and R, and inclusion morphisms $K \to L$ and $K \to R$. Graph K is the *interface* of r and consists of nodes only, with labels and roots undefined. All expressions in L must be *simple*, that is, they do not contain arithmetic operators, contain at most one occurrence of a list variable, and contain at most one occurrence of a string variable in each occurrence of a string subexpression. Moreover, all variables in R must also occur in L. A *conditional rule schema* $\langle r, \Gamma \rangle$ consists of a rule schema r and an application condition Γ according to the grammar of Figure 2, where all variables occurring in Γ also occur in the left-hand graph of r. □

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

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

Figure 2: Application conditions for rule schemata

Definition 4 (Label assignment) Consider a rule graph L and the set X of all variables occurring in L. For each $x \in X$, let $\mathrm{dom}(x)$ denote the domain of x associated with the type of x. A *label assignment* for L is a triple $\alpha = \langle \alpha_{\mathbb{L}}, \mu_V, \mu_E \rangle$ where $\alpha_{\mathbb{L}} \colon X \to \mathbb{L}$ is a function such that for each $x \in X$, $\alpha_{\mathbb{L}}(x) \in \mathrm{dom}(x)$, and $\mu_V \colon V_L \to \mathbb{M}_V \backslash \{\texttt{none}\}$ and $\mu_E \colon E_L \to \mathbb{M}_E \backslash \{\texttt{none}\}$ are partial functions assigning a mark to each node and edge marked with `any`. □

Given a rule graph M, a host graph G, an injective premorphism $g \colon M \to G$, and a label assignment $\alpha = \langle \alpha_{\mathbb{L}}, \mu_V, \mu_E \rangle$ for M, the *instance* $M^{g,\alpha}$ is obtained as follows: (1) replace each variable x in a list expression with $\alpha_{\mathbb{L}}(x)$; (2) replace each `any` mark of a node v or edge e with $\mu_V(v)$ resp. $\mu_E(e)$; (3) replace each node identifier n in a list expression with $g(n)$; (4) evaluate all resulting list expressions according to the meaning of the operators in Figure 1a (see [1] for details). Note that $M^{g,\alpha}$ is a host graph.

The instance $\Gamma^{g,\alpha}$ of an application condition Γ is obtained by applying steps (1) and (3), and evaluating the resulting condition according to the meaning of the operators in Figure 2 (see [1] for details). Note that $\Gamma^{g,\alpha}$ is either "true" or "false".

Definition 5 (Conditional rule schema application) Consider a conditional rule schema $r = \langle L \leftarrow K \rightarrow R, \Gamma \rangle$, host graphs G and H, and an injective premorphism $g \colon L \to G$. Then G *directly derives* H by r and g, denoted by $G \Rightarrow_{r,g} H$, if there exists a label assignment α for L such that

(i) $g \colon L^{g,\alpha} \to G$ is a label preserving graph morphism,
(ii) $\Gamma^{g,\alpha}$ is true,
(iii) $G \Rightarrow_{r^{g,\alpha},g} H$.

Here $G \Rightarrow_{r^{g,\alpha},g} H$ denotes the existence of the following natural double-pushout:[1]

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

□

Given r and g such that (i) and (ii) are satisfied, there exists a natural double-pushout as above if and only if g satisfies the *dangling condition*: no node in $g(L-K)$ must be incident to an edge in $G - g(L)$.

In graph transformations, usually a derivation do not require the double-pushouts to be natural [8]. Here, we require them to be natural due to relabelling (see [1, 4] for the motivation of using natural double-pushouts and for their construction).

A rule schema r without application condition can be considered as the conditional rule schema $\langle r, \Delta \rangle$ where Δ is a condition that is always true (such as `0=0`). In this case, point (ii) in the above definition is trivially satisfied.

[1] A pushout is *natural* if it is also a pullback.

2.3 Syntax and Semantics of Programs

A graph program consists of declarations of conditional rule schemata and procedures, and exactly one declaration of a main command sequence, which is a distinct procedure named `Main`. Procedures must be non-recursive, they can be seen as macros. The syntax of GP 2 programs is defined by the grammar in Figure 3 (where we omit the syntax of rule schema declarations). In the following we describe the main control constructs.

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

Figure 3: Abstract syntax of GP 2 programs

The call of a rule set $\{r_1, \ldots, r_n\}$ non-deterministically applies one of the rules whose left-hand graph matches a subgraph of the host graph such that the dangling condition and the rule's application condition are satisfied. The call *fails* if none of the rules is applicable to the host graph.

The command `if` C `then` P `else` Q is executed on a host graph G by first executing C on a copy of G. If this results in a graph, P is executed on the original graph G; otherwise, if C fails, Q is executed on G. The `try` command has a similar effect, except that P is executed on the result of C's execution.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body was entered for the last time. The `break` command inside a loop terminates that loop and transfers control to the command following the loop.

In general, the execution of a program on a host graph may result in different graphs, fail, or diverge. The operational semantics of GP 2 is defined by the inference rules of Figure 4, where $\mathscr{R}$ stands for a rule set call; C, P, P', and Q stand for command sequences; and G and H stand for host graphs. Given a program P, the rules induce a semantic function which maps each host graph G to the set $[\![P]\!]G$ of all possible outcomes of executing P on G. The result set may contain proper results in the form of graphs and the special values "fail" and $\bot$. The value "fail" indicates a failed program run while $\bot$ indicates a run that diverges. Hence the set of all configurations is $(\mathsf{ComSeq} \times \mathscr{G}(\mathbb{L})) \cup \mathscr{G}(\mathbb{L}) \cup \{\mathsf{fail}\}$, where ComSeq is the set of command sequences as defined in Figure 3 and $\mathscr{G}(\mathbb{L})$ is the set of all host graphs.

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.

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

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

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

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

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

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

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

Figure 4: Semantic inference rules for GP 2 core commands

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. Here, we only consider totally labelled graphs. Lists in GP 2 graphs can be expressed by variables. In our first-order formulas, variables may express nodes or edges as well (see 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 first-order (FO) formulas is given by the grammar of Figure 5. 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 $\mathsf{list}, \mathsf{atom}, \mathsf{int}, \mathsf{string}$, and char respectively. The nonterminals Character and Digit in the syntax represent the fixed character set of GP 2 characters, and the digit set $\{0, \ldots, 9\}$ respectively, as what we have in the syntax of Figure 1.

The quantifiers $\exists_{\mathsf{V}}, \exists_{\mathsf{E}}$, and $\exists_{\mathsf{L}}$ in the grammar are reserved for variables of nodes, edges, and labels respectively. The function symbols $\mathsf{indeg}, \mathsf{outdeg}$ and length return indegree, outdegree, and length of the given argument. Also, we have unary functions $\mathsf{s}, \mathsf{t}, \mathsf{l_V}, \mathsf{l_E}, \mathsf{m_V}$, and $\mathsf{m_E}$, 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 $\mathsf{int}, \mathsf{char}, \mathsf{string}, \mathsf{atom}$ 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 $\forall_{\mathsf{V}} \mathsf{x}(c)$ for $\neg\exists_{\mathsf{V}} x(\neg c)$ and $\exists_{\mathsf{V}} \mathsf{x}_1, \ldots, \mathsf{x}_\mathsf{n}(c)$ for $\exists_{\mathsf{V}} \mathsf{x}_1(\exists_{\mathsf{V}} \mathsf{x}_2(\ldots \exists_{\mathsf{V}} \mathsf{x}_\mathsf{n}(c) \ldots))$ (also for edge and label quantifiers). Also, we define 'terms' 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

```
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
```

Figure 5: Syntax of first-order formulas

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 $(\mathsf{root}(\mathsf{x}))^\alpha$ 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 6 (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 similar to 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.

Given a rule schema $\langle L \leftarrow K \rightarrow R \rangle$ and an injective morphism $g : L \rightarrow G$ for some host graph G. The satisfaction of a condition c over L may not be defined in G. However, we can rename some nodes and edges in G with respect to g so that c is a condition over the graph (with renamed nodes and edges).

Definition 7 (Replacement graph) Given an injective morphism $g : L \to G$ for host graphs L and G. Graph $\rho_g(G)$ is a replacement graph of G w.r.t. g if $\rho_g(G)$ is isomorphic to G with L as a subgraph. □

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 derive properties from output graph to the input graph. Hence, we define a generalisation of a rule schema. Here, we define an *unrestricted rule schema* as a rule schema without any restriction on the occurring labels.

Definition 8 (Generalised rule) 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 is essentially similar to 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 $r^\vee = \langle \langle L \leftarrow K \rightarrow R \rangle, \Gamma^\vee, \texttt{true} \rangle$ 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 from a given 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 [7]. Here, a precondition is limited to a closed FO formula.

Definition 9 (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,

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

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. □

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 $\mathrm{SLP}(c, r)$. Similar approach has been used in [15, 9, 13] for constructing a weakest liberal precondition from a given postcondition.

To give a better idea of the transformations we define in this section, we show a running example for the construction. We use the conditional rule schema `del` of Figure 6 and the preconditions $q = \neg\exists \mathsf{E}\mathsf{x}(\mathsf{m}_\mathsf{V}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none})$ for the running example. We denote by Γ_1 the GP 2 rule schema condition $\mathsf{d} \geq \mathsf{e}$. In addition, a simple example of the construction can be seen in Section 6.

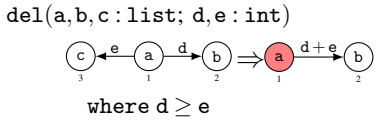


Figure 6: GP 2 conditional rule schema `del`

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 $\text{Split}(c,r)$;
2. Express the dangling condition as a condition over L, denoted by $\text{Dang}(r)$;
3. Evaluate terms and Boolean expression in $\text{Split}(c,r)$, $\text{Dang}(r)$, 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 a way 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. 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 10 (Transformation Split) Given an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$. where $V_L = \{v_1, \ldots, v_n\}$ and $E_L = \{e_1, \ldots, e_m\}$. Let c be a condition over L sharing no variables with r (note that it is always possible to replace the label variables in c with new variables that are distinct from variables in r). We define the condition $\text{Split}(c,r)$ over L inductively as follows:

- Base case.

  If c is true, false, a predicate $\mathsf{int(t)}, \mathsf{char(t)}, \mathsf{string(t)}, \mathsf{atom(t)}, \mathsf{root(t)}$ for some term t, or in the form $\mathsf{t}_1 \ominus \mathsf{t}_2$ for $\ominus \in \{=, ., \neq, ., <, \leq, >, \geq\}$ and some terms $\mathsf{t}_1, \mathsf{t}_2$,

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

- Inductive case.

  Let c_1 and c_2 be conditions over L.

  1) $\text{Split}(c_1 \vee c_2, r) = \text{Split}(c_1, r) \vee \text{Split}(c_2, r)$,

  2) $\text{Split}(c_1 \wedge c_2, r) = \text{Split}(c_1, r) \wedge \text{Split}(c_2, r)$,

  3) $\text{Split}(\neg c_1, r) = \neg \text{Split}(c_1, r)$,

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

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

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

  6) $\text{Split}(\exists_{\mathsf{L}} \mathsf{x}(c_1), r) = \exists_{\mathsf{L}} \mathsf{x}(\text{Split}(c_1, r))$

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

In constructing $\mathrm{Split}(c,r)$, 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, if the condition contains the term $\mathsf{s(x)}$ or $\mathsf{t(x)}$, it may represent a node in the image of the match. Hence, we need to check these possibilities as well.

Example 1 (Transformation Split)

$$\begin{aligned}\mathrm{Split}(q,\mathtt{del}) = \neg(\mathsf{m_V(s(e1))} \neq \mathsf{none} \vee \mathsf{m_V(s(e2))} \neq \mathsf{none}\\ \vee\ \exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge ((\mathsf{s(x)} = 1 \wedge \mathsf{m_V}(1) \neq \mathsf{none}) \vee (\mathsf{s(x)} = 2 \wedge \mathsf{m_V}(2) \neq \mathsf{none})\\ \vee (\mathsf{s(x)} = 3 \wedge \mathsf{m_V}(3) \neq \mathsf{none})\\ \vee (\mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{s(x)} \neq 3 \wedge \mathsf{m_V(s(x))} \neq \mathsf{none}))))\end{aligned}$$

Besides 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. For every node $v \in L - K$, the dangling condition is satisfied if and only if v is not incident to any edge not in L. Therefore, the indegree and outdegree of v in $\rho_g(G)$ must be equal to the indegree and outdegree of v in L. 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.

Example 2 (Dangling Condition) $\mathrm{Dang}(\mathtt{del}) = \mathsf{indeg(3)} = 1 \wedge \mathsf{outdeg(3)} = 0$

Since we have information about some properties of L from the rule, we can put the information in the condition by evaluating the condition we obtained from Split and Dang with respect to L. For this, we construct of $\mathrm{Val}(d,r)$ for a condition d over L where L is the left-hand graph of r. Intuitively, $\mathrm{Val}(d,r)$ 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 subformula 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 readers to [18].

Example 3 (Valuation of a Graph Condition)

1. $\mathrm{Val}(\mathrm{Split}(q,\mathtt{del}),\mathtt{del})$

$$\begin{aligned}&= \neg(\mathsf{none} \neq \mathsf{none} \vee \mathsf{none} \neq \mathsf{none}\\ &\quad \vee\ \exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge ((\mathsf{s(x)} = 1 \wedge \mathsf{none} \neq \mathsf{none}) \vee (\mathsf{s(x)} = 2 \wedge \mathsf{none} \neq \mathsf{none})\\ &\quad \vee (\mathsf{s(x)} = 3 \wedge \mathsf{none} \neq \mathsf{none})\\ &\quad \vee (\mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{s(x)} \neq 3 \wedge \mathsf{m_V(s(x))} \neq \mathsf{none}))))\\ &\equiv \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge \mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{s(x)} \neq 3 \wedge \mathsf{m_V(s(x))} \neq \mathsf{none})\end{aligned}$$

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

2. $\mathrm{Val}(\Gamma_1,\mathtt{del}) = \mathsf{d} \geq \mathsf{e}$ (for this case, we change nothing.)

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

Definition 11 (Transformation Lift) For a precondition c and a generalised rule $w = \langle r, ac_L, ac_R \rangle$ with an unrestricted rule schema $r = \langle L \leftarrow K \rightarrow R \rangle$,

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

Example 4 (Transformation Lift)
$\text{Lift}(q, \texttt{del}^{\vee}) = \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{x} \neq \mathsf{e2} \wedge \mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{s}(\mathsf{x}) \neq 3 \wedge \mathsf{m}_{\mathsf{V}}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e}$

4.2 From Left to Right-Application Condition

To obtain a right-application condition from the obtained left-application condition, we need to consider properties that could be different in the initial and result graphs. Recall that in constructing a left-application condition, we evaluate all functions with a node/edge constant argument so that the satisfaction of the condition is no longer independent of the properties of the left-hand graph.

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 node and edge in the right-hand side. Hence, to obtain the right-application condition $\text{Shift}(c, w)$, we have some adjustment to the obtained left-application condition, denoted by $\text{Adj}(d, r)$ where $d = \text{Lift}(c, w)$.

To obtain $\text{Adj}(d, r)$, we follow the following steps:

1. Replace every 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 $\exists_{\mathsf{V}}\mathsf{x}(c_1)$ with $\exists_{\mathsf{V}}\mathsf{x}(x \neq v_1 \wedge \ldots \wedge x \neq v_n \wedge c_1)$ and every $\exists_{\mathsf{E}}\mathsf{x}(c_1)$ with $\exists_{\mathsf{E}}\mathsf{x}(x \neq e_1 \wedge \ldots \wedge x \neq e_m \wedge c_1)$ for $V_R - V_K = \{v_1, \ldots, v_n\}$ and $E_R - E_K = \{e_1, \ldots, e_n\}$.

Definition 12 (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 $\mathsf{incon}(\mathsf{x})$ (or $\mathsf{outcon}(\mathsf{x})$) for $x \in V_K$ with $\mathsf{indeg}(\mathsf{x}) - \mathit{indeg}_R(x)$ (or $\mathsf{outdeg}(\mathsf{x}) - \mathit{outdeg}_R(x)$). Let also $\{v_1, \ldots, v_n\}$ and $\{e_1, \ldots, e_m\}$ denote the set of all nodes and edges in $R - K$ respectively. The *adjusted* condition of c w.r.t r, denoted by $\text{Adj}(c, r)$, 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, $\text{Adj}(c, r) = c'$;
2. If c is the predicates $\mathsf{int}(\mathsf{x}), \mathsf{char}(\mathsf{x}), \mathsf{string}(\mathsf{x})$ or $\mathsf{atom}(\mathsf{x})$ for a list variable x, $\text{Adj}(c, r) = c'$;
3. If $c = \mathsf{root}(\mathsf{x})$ for some term x representing a node, $\text{Adj}(c, r) = c'$
4. If $c = x_1 \ominus x_2$ for some terms x_1, x_2 and $\ominus \in \{=, \neq, <, \leq, >, \geq\}$,
$$\text{Adj}(c, r) = \begin{cases} \mathsf{false} & \text{, if } \ominus \in \{=\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ \mathsf{true} & \text{, if } \ominus \in \{\neq\} \text{ and } x_1 \in V_L - V_K \cup E_L \text{ or } x_2 \in V_L - V_K \cup E_L, \\ c' & \text{, otherwise} \end{cases}$$
5. $\text{Adj}(c_1 \vee c_2, r) = \text{Adj}(c_1, r) \vee \text{Adj}(c_2, r)$
6. $\text{Adj}(c_1 \wedge c_2, r) = \text{Adj}(c_1, r) \wedge \text{Adj}(c_2, r)$
7. $\text{Adj}(\neg c_1, r) = \neg\text{Adj}(c_1, r)$
8. $\text{Adj}(\exists_{\mathsf{V}}\mathsf{x}(c_1), r) = \exists_{\mathsf{V}}\mathsf{x}(x \neq v_1 \wedge \ldots \wedge x \neq v_n \wedge \text{Adj}(c_1, r))$
9. $\text{Adj}(\exists_{\mathsf{E}}\mathsf{x}(c_1), r) = \exists_{\mathsf{E}}\mathsf{x}(x \neq e_1 \wedge \ldots \wedge x \neq e_m \wedge \text{Adj}(c_1, r))$
10. $\text{Adj}(\exists_{\mathsf{L}}\mathsf{x}(c_1), r) = \exists_{\mathsf{L}}\mathsf{x}(\text{Adj}(c_1, r))$ □

Example 5 (Adjusment)
Let p denotes $\text{Lift}(q, \texttt{del}^{\vee})$. Then,
$\text{Adj}(p, \texttt{del}) = \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{s}(\mathsf{x}) \neq 1 \wedge \mathsf{s}(\mathsf{x}) \neq 2 \wedge \mathsf{m}_{\mathsf{V}}(\mathsf{s}(\mathsf{x})) \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e}$

Although $\mathrm{Adj}(\mathrm{Lift}(c,w),r)$ can be considered as a right-application condition, we need a stronger condition to have a strongest liberal postcondition. Hence, 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, equality of functions and their value in R, and type of label variables in R.

Definition 13 (Specifying a Totally Labelled Graph) Given a totally labelled graph R with the set of nodes $V_R = \{v_1,\ldots,v_n\}$ and the set of edges $E_R = \{e_1,\ldots,e_m\}$. Let $X = \{x_1,\ldots,x_k\}$ be the set of all list variables in R, and $\mathrm{Type}(x)$ for $x \in X$ is $\mathsf{int(x)}$, $\mathsf{char(x)}$, $\mathsf{string(x)}$, $\mathsf{atom(x)}$, or true if x is an integer, char, string, atom, or list variable respectively. Let also $\mathrm{Root}_R(v)$ for $v \in V_R$ be a function such that $\mathrm{Root}_R(v) = \mathsf{root(v)}$ if $p_R(v) = 1$, and $\mathrm{Root}_R(v) = \neg\mathsf{root(v)}$ otherwise. *A specification of R*, denoted by $\mathrm{Spec}(R)$, is the condition over R:

$$\bigwedge_{\mathsf{i}=1}^{\mathsf{k}} \mathrm{Type}(x_i) \;\wedge\; \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{l_V(v_i)} = \ell_R(v_i) \;\wedge\; \mathsf{m_V(v_i)} = m_R(v_i) \;\wedge\; \mathrm{Root}_R(v_i)$$
$$\wedge\; \bigwedge_{\mathsf{i}=1}^{\mathsf{m}} \mathsf{s(e_i)} = s_L(e_i) \;\wedge\; \mathsf{t(e_i)} = t_R(e_i) \;\wedge\; \mathsf{l_E(e_i)} = \ell_L(e_i) \;\wedge\; \mathsf{m_E(e_i)} = m_R(e_i) \qquad \square$$

Basically, $\mathrm{Spec}(R)$ explicitly shows us node and edge identifiers in R, label, mark, and rootedness of each node in R (if defined), also the source, target, label, and mark of each edge 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, $G \models \mathrm{Spec}(R)$ if and only if there exists assignment $\alpha_{\mathbb{L}}$ such that $g : R^{\alpha_{\mathbb{L}}} \to G$ is an inclusion.

Definition 14 (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:

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

Example 6 (Obtaining Right-Application Condition)

$$\begin{aligned}\mathrm{Shift}(q, \mathtt{del}^{\vee}) \;=\; & \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{x} \neq \mathsf{e1} \wedge \mathsf{s(x)} \neq 1 \wedge \mathsf{s(x)} \neq 2 \wedge \mathsf{m_V(s(x))} \neq \mathsf{none}) \wedge \mathsf{d} \geq \mathsf{e} \\ & \wedge \mathsf{l_V}(1) = \mathsf{a} \wedge \mathsf{l_V}(2) = \mathsf{b} \wedge \mathsf{l_E(e1)} = \mathsf{d} + \mathsf{e} \wedge \mathsf{m_V}(1) = \mathsf{red} \\ & \wedge \mathsf{m_V}(2) = \mathsf{none} \wedge \mathsf{m_E(e1)} = \mathsf{none} \wedge \mathsf{s(e1)} = 1 \wedge \mathsf{t(e1)} = 2 \\ & \wedge \neg\mathsf{root}(1) \wedge \neg\mathsf{root}(2) \wedge \mathsf{int(d)} \wedge \mathsf{int(e)}\end{aligned}$$

4.3 From Right-Application Condition to Postcondition

The right-application condition we obtained from transformation Shift is strong enough to express properties of the replacement graph of any resulting graph. To be able to check the satisfaction of the condition in the resulting graph, 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, $G \models c$ implies $H \models \mathrm{Var}(c)$.

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 15 (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:

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

For a rule schema r, we denote by $\mathrm{Slp}(c,r)$ and $\mathrm{Slp}(c,r^{-1})$ the formulas $\mathrm{Post}(c,r^{\vee})$ and $\mathrm{Post}(c,(r^{\vee})^{-1})$ respectively. □

Example 7 (Obtaining Strongest Liberal Postcondition)

$$\begin{aligned}\mathrm{Slp}(q,\texttt{del}) \;=\; & \exists_{\mathsf{V}}\mathsf{u},\mathsf{v}(\mathsf{u}\neq\mathsf{v}\wedge\exists_{\mathsf{E}}\mathsf{w}(\exists_{\mathsf{L}}\mathsf{a},\mathsf{b},\mathsf{d},\mathsf{e}(\\ & \neg\exists_{\mathsf{E}}\mathsf{x}(\mathsf{x}\neq\mathsf{w}\wedge\mathsf{s}(\mathsf{x})\neq\mathsf{u}\wedge\mathsf{s}(\mathsf{x})\neq\mathsf{v}\wedge\mathsf{m}_{\mathsf{V}}(\mathsf{s}(\mathsf{w}))\neq\mathsf{none})\wedge\mathsf{d}\geq\mathsf{e}\\ & \wedge\mathsf{l}_{\mathsf{V}}(\mathsf{u})=\mathsf{a}\wedge\mathsf{l}_{\mathsf{V}}(\mathsf{v})=\mathsf{b}\wedge\mathsf{l}_{\mathsf{E}}(\mathsf{w})=\mathsf{d}+\mathsf{e}\wedge\mathsf{m}_{\mathsf{V}}(\mathsf{u})=\mathsf{red}\\ & \wedge\mathsf{m}_{\mathsf{V}}(\mathsf{v})=\mathsf{none}\wedge\mathsf{m}_{\mathsf{E}}(\mathsf{w})=\mathsf{none}\wedge\mathsf{s}(\mathsf{w})=\mathsf{u}\wedge\mathsf{t}(\mathsf{w})=\mathsf{v}\\ & \wedge\neg\mathsf{root}(\mathsf{u})\wedge\neg\mathsf{root}(\mathsf{v})\wedge\mathsf{int}(\mathsf{d})\wedge\mathsf{int}(\mathsf{e}))))\end{aligned}$$

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. As pre- and post-conditions, we use arbitrary assertions 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 $\models\{c\}\ P\ \{d\}$, if for every graph G satisfying c, all graphs in $[\![P]\!]G$ satisfy d [16].

5.1 Semantic Partial Correctness Calculus

Besides strongest liberal postconditions, it will be useful to consider weakest liberal preconditions.

Definition 16 (Weakest liberal precondition) An assertion c is a *liberal precondition* with respect to a graph program P and a postcondition d, if for all host graphs G and H,

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

A *weakest liberal precondition* w.r.t. P and d, written $\mathrm{WLP}(P,d)$, is a liberal precondition w.r.t. P and d that is implied by all liberal postconditions w.r.t. P and d. □

To prove that a triple $\{c\}\ P\ \{d\}$ is partially correct, 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 $\mathrm{SLP}(c,P)$ or $\mathrm{WLP}(P,d)$ may be difficult because P may diverge. In [9, 13], divergence is represented by infinite formulas while in [10] approximations of these assertions are used. We take a different approach by considering SLP and WLP only for loop-free programs. Programs with loops are verified using the proof rule [alap] in the calculi introduced below.

Before we define our proof rules, we define assertions expressing that a program can produce a result graph or may fail, respectively. These assertions are needed in the proof rules for the branching commands `if_then_else` and `try_then_else`.

Definition 17 (Assertions SUCCESS and FAIL) For a graph program P, $\mathrm{SUCCESS}(P)$ and $\mathrm{FAIL}(P)$ are the predicates defined on all host graphs G by
$G\models\mathrm{SUCCESS}(P)$ if and only if there exists a host graph H with $H\in[\![P]\!]G$, and
$G\models\mathrm{FAIL}(P)$ if and only if $\mathrm{fail}\in[\![P]\!]G$. □

We also define a predicate Break to deal with loops containing the `break` command.

Definition 18 (Predicate Break) Given a graph program P and assertions c and d, $\mathrm{Break}(c,P,d)$ holds if and only if for all derivations $\langle P,G\rangle\rightarrow^*\langle\texttt{break},H\rangle$, $G\models c$ implies $H\models d$. □

Here P is a loop body whose execution on graph G encounters the `break` command, and H is the graph that has been reached at that point.

Definition 19 (Semantic partial correctness proof rules) The semantic partial correctness proof rules for GP 2 commands, denoted by SEM, are defined in Figure 7a, where c, d, and d' are assertions, r is a conditional rule schema, $\mathscr{R}$ 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`.

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

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

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

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

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

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

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

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

(a) Calculus SEM

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

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

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

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

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

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

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

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

(b) Calculus SYN

Figure 7: Semantic (a) and syntactic (b) partial correctness proof calculus, where $\text{S}(C)$ is $\text{SUCCESS}(C)$ and $\text{F}(C)$ is $\text{FAIL}(C)$

5.2 Syntactic Partial Correctness Calculus

Defining a first-order formula for $\text{SUCCESS}(r)$ with a rule schema r is easier than defining FO formula for $\text{SUCCESS}(P)$ 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 $\text{FAIL}(P)$ for bigger class of programs because some commands cannot fail (see [1]).

Definition 20 (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
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, fail $\in [\![C;Q]\!]G$ iff fail $\in [\![C]\!]G$ or $H \in [\![C]\!]G \wedge$ fail $\in [\![Q]\!]H$ for some host graph H, which means $G \vDash \text{FAIL}(C) \vee (\text{SUCCESS}(C) \wedge \text{FAIL}(Q))$. We can construct both Fail(C) and Success(C) if C is a loop-free program (see [18] for the detail of construction), and we can construct Fail(Q) if Q is a loop-free program or a non-failing command. Here, we introduce the class of *iteration commands* for which we can obtain Fail of the commands.

Definition 21 (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. □

If S is a loop-free program, we can construct Fail(S) as stated above (see the full construction in [18]. Meanwhile, if S is a non-failing command, there is no graph G such that fail$\in [\![S]\!]G$, so we can conclude that Fail$(S) \equiv$ false. If S is in the form of $C;P$ for a loop-free program C and a non-failing program P, fail$\in [\![S]\!]G$ for a graph G only if fail$\in [\![C]\!]G$ (because P cannot fail), so that Fail$(S) \equiv$ Fail(C).

Definition 22 Let $\text{Fail}_{\text{lf}}(C)$ denotes the formula 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 Success(P) and Slp(c,P) such that $G \vDash$ Success(P) if and only if $G \vDash$ SUCCESS(P) and $G \vDash$ Slp(c,P) if and only if $G \vDash$ SLP(c,P). Also, for any iteration command S, $G \vDash$ Fail(S) if and only if $G \vDash$ FAIL(S).

The construction of Slp(c,P) and Success(P) to show that Theorem 2 holds can be found in [18]. Since we only have a construction for Success(C) for a loop-free program C and 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 23 (Control programs) A *control command* is a command where the condition of every branching command (e.g. the command C of $\texttt{if}\ C\ \texttt{then}\ P\ \texttt{else}\ Q$) 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 [9], a First-order formula of 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, $G \vDash \neg\text{Slp}(\neg d, r^{-1})$ if and only if $G \vDash$ WLP(r,d).

Definition 24 (Syntactic partial correctness proof rules) The syntactic partial correctness proof rules, denoted by SYN, are defined in Figure 7b, where c, d, and d' are conditions, r is a conditional rule schema, $\mathscr{R}$ 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.

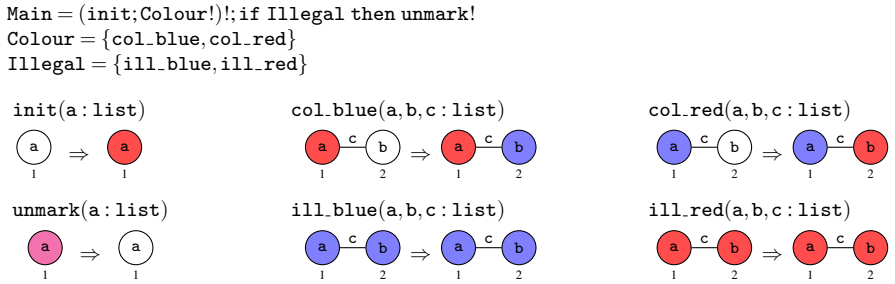


Figure 8: Graph program 2-colouring

Table 2: Conditions inside proof tree of $2-\texttt{colouring}$

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

6 Example: Verifying a 2-Colouring Program

In this section, we show how to verify the 2-colouring graph program given in Figure 8. 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 [17, 15] 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

$c = \forall_V x(m_V(x) = \text{none} \wedge \neg\text{root}(x)) \wedge \forall_E x(m_E(x) = \text{none})$, and

$d = \forall_V x((m_V(x) = \text{red} \vee m_V(x) = \text{blue})) \wedge \neg\exists_E x(s(x) \neq t(x) \wedge m_V(s(x)) = m_V(t(x)))$

By using the conditions in Table 2, we then have a proof tree as in Figure 9 for the partial correctness of $2-\texttt{colouring}$ with respect to c and $c \vee d$.

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

where subtree I is:

$$\dfrac{\dfrac{[\text{ruleapp}]_{\text{slp}}\ \dfrac{}{\{f\}\ \texttt{init}\ \{\text{Slp}(f,\texttt{init})\}}}{[\text{cons}]\ \dfrac{\{f\}\ \texttt{init}\ \{f\}}{[\text{comp}]\ \dfrac{\{f\}\ \texttt{init;Colour!}\ \{f\}}{[\text{alap}]\ \dfrac{\{f\}\ (\texttt{init;Colour!})!\ \{f \wedge \text{Fail}(\texttt{init;Colour!})\}}{[\text{cons}]\ \dfrac{\{f\}\ (\texttt{init;Colour!})!\ \{e\}}{\{f\}\ (\texttt{init;Colour!})!\ \{c \vee d\}}}}}}}$$

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

$$\dfrac{[\text{ruleapp}]_{\text{slp}}\ \dfrac{}{\{f\}\ \texttt{c\_red}\ \{\text{Slp}(f,\texttt{c\_red})\}}}{[\text{cons}]\ \dfrac{\{f\}\ \texttt{c\_red}\ \{f\}}{\{f\}\ \texttt{Colour}\ \{f\}}}$$

and subtree II is:

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

$$\dfrac{[\text{ruleapp}]_{\text{slp}}\ \dfrac{}{\{d\}\ \texttt{skip}\ \{d\})}}{[\text{cons}]\ \dfrac{\{e \wedge \text{Fail}(\texttt{Illegal})\}\ \texttt{skip}\ \{c \vee d\}}{}}$$

Figure 9: Proof tree for partial correctness of `2colouring`

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

For an example of constructing Slp, let us consider the rule $r = \texttt{init}$ of program $2-\texttt{colouring}$ 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.

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

In the proof tree of Figure 9, 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 a 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 $\models \{c\}\ P\ \{d\}$. Moreover, if c and d are first-order formulas, $\vdash_{\mathsf{SYN}} \{c\}\ P\ \{d\}$ implies $\models \{c\}\ P\ \{d\}$.

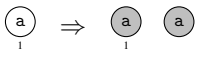
A proof calculus is complete if every partially correct triple can be proved by the calculus. Neither SEM nor SYN are complete because GP 2's expressions include Peano arithmetic which is known to be incomplete [12]. However, the notion of relative completeness allows to separate the incompleteness in proving valid assertions from the power of the inference rules for programming constructs [5]. 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, $\models \{c\}\ P\ \{d\}$ implies $\vdash_{\mathsf{SEM}} \{c\}\ P\ \{d\}$.

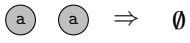
The proof of Theorem 4 can be seen in [18]. The proof relies on the existence of WLP(P,c) for arbitrary programs P and assertions c. Even if we omit [ruleapp$_{\text{slp}}$] from the calculus, SEM is still relative complete. However, for SYN to be relative complete, it would be necessary to express WLP(P,c) or SLP (c,P) as first-order formulas. 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 $\models \{c\}$ `duplicate!;delete!` $\{d\}$ holds: `duplicate!` duplicates the number of nodes while marking the nodes grey, hence its result graph consists of an even number of isolated grey nodes. Then `delete!` deletes pairs of grey nodes as long as possible, so the overall result is the empty graph. Note that "consists of an even number of isolated grey nodes" is both the strongest postcondition with respect to c and `duplicate!`, and the weakest precondition with respect to `delete!` and d.

Using SYN one can prove $\vdash \{c\}$ `duplicate!` $\{e\}$ where e expresses that all nodes are grey and isolated. However, we 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 [11] 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 with attributed rules was introduced in [17, 15], using E-conditions which generalise the nested graph conditions of Habel and Pennemann [9, 13]. E-conditions do not cover rooted rules or the `break` command, which are considered in our first-order formulas. More importantly, the approach of [17, 15] can only handle programs in which the conditions of branching commands and loop bodies are rule set calls. Our syntactic calculus SYN covers a larger class of graph

programs, viz. programs where the condition of each branching command is a loop-free program, and each loop body is an iteration command. This allows us, in particular, to verify many programs with nested loops. Besides this increased power, we believe that assertions in the form of first-order formulas are easier to comprehend by programmers than nested graph conditions of some form.

As argued at the end of the previous section, we cannot express $\mathrm{SLP}(c,P)$ or $\mathrm{WLP}(P,c)$ for arbitrary assertions c and graph programs P as first-order formulas. In [9, 13], there is a construction of $\mathrm{Wlp}(c,P!)$ by using an infinite formula. Here, we do not use a similar trick but stick to standard finitary logic. The papers [7, 10] do not give constructions for syntactic strongest liberal postconditions or weakest liberal postconditions either. Instead, similar to the consequent of our inference rule [alap], the conjunction of a loop invariant and a negated loop condition is considered as an "approximate" strongest liberal postcondition.

In [3], the authors design an imperative programming language for manipulating graphs and give a Hoare calculus based on weakest preconditions. Programs manipulate the graph structure only and do not contain arithmetic. Assertions are formulas of the so-called guarded fragment of first-order logic, which is decidable. This relatively weak logic makes the correctness of programs decidable.

Our goal is different in that we want a powerful assertion language that can specify many practical algorithms on graphs. (In fact, we plan to extend our logic to monadic second-order logic in order to express non-local properties such as connectedness, colourability, etc.) In our setting, it is easily seen that correctness is undecidable in general, even for trivial programs. For example, consider Hoare triples of the form $\{\texttt{true}\}\texttt{skip}\{d\}$ where d is an arithmetic formula (without references to nodes or edges). Such a triple is partially (and totally) correct if and only if d is true on the integers. But our formulas include Peano arithmetic and hence are undecidable in general [12]. Thus, even for triples of the restricted form above, correctness is undecidable.

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 $\mathrm{SUCCESS}(C)$ for a loop-free program C. Moreover, we can construct a first-order formula for $\mathrm{FAIL}(P)$ 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 [6].

Another limitation in current approaches to graph program verification is the inability to specify isomorphisms between the initial and final graphs [19]. Monadic second-order transductions can link initial and final states by expressing the final state through elements of the initial state [6]. We plan to adopt this technique for graph program verification in the future.

References

[1] Christopher Bak (2015): *GP 2: Efficient Implementation of a Graph Programming Language*. Ph.D. thesis, Department of Computer Science, University of York. Available at `http://etheses.whiterose.ac.uk/12586/`.

[2] Christopher Bak & Detlef Plump (2012): *Rooted Graph Programs*. In: *Proc. Int. Workshop on Graph Based Tools (GraBaTs 2012)*, *Electronic Communications of the EASST* 54, doi:10.14279/tuj.eceasst.54.780.

[3] Jon Haël Brenas, Rachid Echahed & Martin Strecker (2018): *Verifying Graph Transformations with Guarded Logics*. In: *Proc. Int. Symposium on Theoretical Aspects of Software Engineering (TASE 2018)*, IEEE, pp. 124–131, doi:10.1109/TASE.2018.00024.

[4] Graham Campbell, Jack Romö & Detlef Plump (2020): *The Improved GP 2 Compiler*. *ArXiv e-prints* arXiv:2010.03993 [cs.PL]. Available at `https://arxiv.org/abs/2010.03993`.

[5] Stephen A. Cook (1978): *Soundness and Completeness of an Axiom System for Program Verification*. *SIAM Journal on Computing* 7(1), pp. 70–90, doi:10.1137/0207005.

[6] Bruno Courcelle & Joost Engelfriet (2012): *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, doi:10.1017/CBO9780511977619.

[7] Edsger W. Dijkstra & Carel S. Scholten (1990): *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer, doi:10.1007/978-1-4612-3228-5.

[8] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange & Gabriele Taentzer (2006): *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/3-540-31188-2.

[9] Annegret Habel & Karl-Heinz Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Math. Struct. Comput. Sci.* 19(2), pp. 245–296, doi:10.1017/S0960129508007202.

[10] Clifford B. Jones, A.W. Roscoe & Kenneth R. Wood, editors (2010): *Reflections on the Work of C.A.R. Hoare*. Springer, doi:10.1007/978-1-84882-912-1.

[11] Leonid Libkin (2004): *Elements of Finite Model Theory*. Texts in Theoretical Computer Science, Springer, doi:10.1007/978-3-662-07003-1.

[12] James Donald Monk (1976): *Mathematical Logic*. *Graduate Texts in Mathematics* 37, Springer, doi:10.1007/978-1-4684-9452-5.

[13] Karl-Heinz Pennemann (2009): *Development of Correct Graph Transformation Systems*. Ph.D. thesis, Department of Computing Science, University of Oldenburg. Available at `http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/diss_pennemann.pdf`.

[14] Detlef Plump (2012): *The Design of GP 2*. In: *Proc. Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, *EPTCS* 82, pp. 1–16, doi:10.4204/EPTCS.82.1.

[15] Christopher M. Poskitt (2013): *Verification of Graph Programs*. Ph.D. thesis, The University of York. Available at `http://etheses.whiterose.ac.uk/4700/`.

[16] Christopher M. Poskitt & Detlef Plump (2010): *A Hoare Calculus for Graph Programs*. In: *Proc. Int. Conference on Graph Transformation (ICGT 2010)*, *LNCS* 6372, Springer, pp. 139–154, doi:10.1007/978-3-642-15928-2_10.

[17] Christopher M. Poskitt & Detlef Plump (2012): *Hoare-Style Verification of Graph Programs*. *Fundamenta Informaticae* 118(1-2), pp. 135–175, doi:10.3233/FI-2012-708.

[18] Gia Wulandari & Detlef Plump (2020): *Verifying Graph Programs with First-Order Logic (Extended Version)*. *ArXiv e-prints* arXiv:2010.14549 [cs.LO]. Available at `https://arxiv.org/abs/2010.14549`.

[19] Gia S. Wulandari & Detlef Plump (2018): *Verifying a Copying Garbage Collector in GP 2*. In: *Software Technologies: Applications and Foundations – STAF 2018 Collocated Workshops, Revised Selected Papers*, *LNCS* 11176, Springer, pp. 479–494, doi:10.1007/978-3-030-04771-9_34.